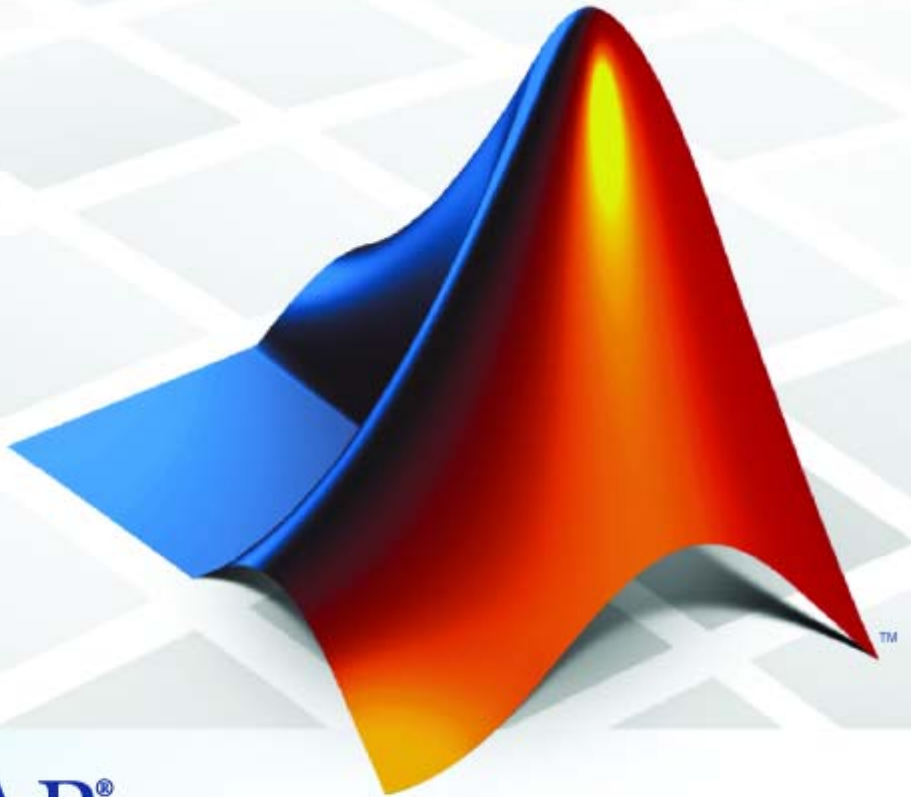


Simulink® 7

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The 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 1990–2009 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

The 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)

1

Simulink Basics

Starting Simulink Software	1-2
Opening a Model	1-4
Introduction	1-4
Opening an Existing Model	1-4
Opening Models with Different Character Encodings	1-4
Avoiding Initial Model Open Delay	1-5
Loading a Model	1-7
Saving a Model	1-8
Introduction	1-8
How to Tell If a Model Needs Saving	1-8
Techniques for Saving Models	1-9
Saving Models with Different Character Encodings	1-10
Saving a Model in an Earlier Simulink Version	1-11
Saving from One Earlier Simulink Version to Another ...	1-13
Using the Model Editor	1-14
Editor Overview	1-14
Toolbar	1-15
Menu Bar	1-19
Canvas	1-19
Context Menus	1-20
Status Bar	1-20
Undoing a Command	1-22
Zooming Block Diagrams	1-22
Panning Block Diagrams	1-23

Viewing Command History	1-25
Bringing the MATLAB Software Desktop Forward	1-25
Copying Models to Third-Party Applications	1-26
Updating a Block Diagram	1-27
Printing a Block Diagram	1-29
About Printing	1-29
Print Dialog Box	1-29
Specifying Paper Size and Orientation	1-31
Positioning and Sizing a Diagram	1-31
Tiled Printing	1-32
Print Sample Time Legend	1-36
Print Command	1-36
Generating a Model Report	1-39
Model Report Options	1-40
Ending a Simulink Session	1-42
Summary of Mouse and Keyboard Actions	1-43
Model Viewing Shortcuts	1-43
Block Editing Shortcuts	1-44
Line Editing Shortcuts	1-45
Signal Label Editing Shortcuts	1-45
Annotation Editing Shortcuts	1-46

How Simulink Works

2

Introduction	2-2
Modeling Dynamic Systems	2-3
Block Diagram Semantics	2-3
Creating Models	2-4

Time	2-5
States	2-5
Block Parameters	2-9
Tunable Parameters	2-9
Block Sample Times	2-10
Custom Blocks	2-11
Systems and Subsystems	2-12
Signals	2-16
Block Methods	2-17
Model Methods	2-18
Simulating Dynamic Systems	2-19
Model Compilation	2-19
Link Phase	2-20
Simulation Loop Phase	2-20
Solvers	2-22
Zero-Crossing Detection	2-24
Algebraic Loops	2-35

Working with Sample Times

3

What Is Sample Time?	3-2
How to Specify the Sample Time	3-3
Designating Sample Times	3-3
Specifying Block-Based Sample Times Interactively	3-5
Specifying Port-Based Sample Times Interactively	3-6
Specifying Block-Based Sample Times Programmatically	3-7
Specifying Port-Based Sample Times Programmatically ..	3-7
Accessing Sample Time Information Programmatically ...	3-8
Specifying Sample Times for a Custom Block	3-8
Determining Sample Time Units	3-8
Changing the Sample Time After Simulation Start Time	3-8
How to View Sample Time Information	3-9
Viewing Sample Time Display	3-9

Managing the Sample Time Legend	3-10
How to Print Sample Time Information	3-13
Types of Sample Time	3-15
Discrete Sample Time	3-15
Continuous Sample Time	3-16
Fixed in Minor Step	3-16
Inherited Sample Time	3-16
Constant Sample Time	3-17
Variable Sample Time	3-18
Triggered Sample Time	3-19
Asynchronous Sample Time	3-19
Determining the Compiled Sample Time of a Block ...	3-20
Managing Sample Times in Subsystems	3-21
Managing Sample Times in Systems	3-22
Purely Discrete Systems	3-22
Hybrid Systems	3-25
Resolving Rate Transitions	3-30
How Propagation Affects Inherited Sample Times	3-31
Process for Sample Time Propagation	3-31
Simulink Rules for Assigning Sample Times	3-31
Monitoring Backpropagation in Sample Times	3-33

Creating a Model

4

Creating an Empty Model	4-2
Creating a Model Template	4-2

Populating a Model	4-4
About the Library Browser	4-4
Opening the Library Browser	4-4
Browsing Block Libraries	4-5
Searching Block Libraries	4-5
Cloning Blocks to Models	4-6
Selecting Objects	4-7
Selecting an Object	4-7
Selecting Multiple Objects	4-7
Specifying Block Diagram Colors	4-9
How to Specify Block Diagram Colors	4-9
Choosing a Custom Color	4-10
Defining a Custom Color	4-10
Specifying Colors Programmatically	4-11
Displaying Sample Time Colors	4-12
Connecting Blocks	4-15
Automatically Connecting Blocks	4-15
Manually Connecting Blocks	4-18
Disconnecting Blocks	4-23
Aligning, Distributing, and Resizing Groups of Blocks	
Automatically	4-24
Annotating Diagrams	4-26
How to Annotate Diagrams	4-26
Annotations Properties Dialog Box	4-27
Annotation Callback Functions	4-30
Associating Click Functions with Annotations	4-31
Annotations API	4-33
Using TeX Formatting Commands in Annotations	4-33
Creating Annotations Programmatically	4-35
Creating Subsystems	4-37
Why Subsystems are Advantageous	4-37
Creating a Subsystem by Adding the Subsystem Block ...	4-38
Creating a Subsystem by Grouping Existing Blocks	4-38
Model Navigation Commands	4-40
Window Reuse	4-40

Labeling Subsystem Ports	4-41
Controlling Access to Subsystems	4-42
Interconverting Subsystems and Block Diagrams	4-43
Emptying Subsystems and Block Diagrams	4-43
Modeling Control Flow Logic	4-44
Equivalent C Language Statements	4-44
Modeling Conditional Control Flow Logic	4-44
Modeling While and For Loops	4-47
Using Callback Functions	4-54
About Callback Functions	4-54
Tracing Callbacks	4-55
Creating Model Callback Functions	4-55
Creating Block Callback Functions	4-58
Port Callback Parameters	4-62
Example Callback Function Tasks	4-63
Using Model Workspaces	4-66
About Model Workspaces	4-66
Simulink.ModelWorkspace Data Object Class	4-67
Changing Model Workspace Data	4-68
Model Workspace Dialog Box	4-70
Resolving Symbols	4-75
About Symbol Resolution	4-75
Hierarchical Symbol Resolution	4-76
Specifying Numeric Values with Symbols	4-77
Specifying Other Values with Symbols	4-77
Limiting Signal Resolution	4-78
Explicit and Implicit Symbol Resolution	4-78
Programmatic Symbol Resolution	4-79
Consulting the Model Advisor	4-80
About the Model Advisor	4-80
Starting the Model Advisor	4-80
Overview of the Model Advisor Window	4-82
Running Model Advisor Checks	4-84
Fixing a Warning or Failure	4-87
Reverting Changes Using Restore Points	4-92
Viewing and Saving Model Advisor Reports	4-94
Running the Model Advisor Programmatically	4-97

Managing Model Versions	4-99
How Simulink Helps You Manage Model Versions	4-99
Model File Change Notification	4-100
Specifying the Current User	4-101
Model Properties Dialog Box	4-103
Creating a Model Change History	4-111
Version Control Properties	4-112
Model Discretizer	4-114
What Is the Model Discretizer?	4-114
Requirements	4-114
How to Discretize a Model from the Model Discretizer	
GUI	4-115
Viewing the Discretized Model	4-124
How to Discretize Blocks from the Simulink Model	4-127
How to Discretize a Model from the MATLAB Command	
Window	4-138

Creating Conditional Subsystems

5

About Conditional Subsystems	5-2
Enabled Subsystems	5-4
What Are Enabled Subsystems?	5-4
Creating an Enabled Subsystem	5-5
Blocks an Enabled Subsystem Can Contain	5-9
Using Blocks with Constant Sample Times in Enabled	
Subsystems	5-11
Triggered Subsystems	5-14
What Are Triggered Subsystems?	5-14
Creating a Triggered Subsystem	5-15
Blocks That a Triggered Subsystem Can Contain	5-18
Triggered and Enabled Subsystems	5-19
What Are Triggered and Enabled Subsystems?	5-19
Creating a Triggered and Enabled Subsystem	5-20
A Sample Triggered and Enabled Subsystem	5-21

Creating Alternately Executing Subsystems	5-21
Function-Call Subsystems	5-24
Conditional Execution Behavior	5-25
What Is Conditional Execution Behavior?	5-25
Propagating Execution Contexts	5-27
Behavior for Switch Blocks	5-28
Displaying Execution Contexts	5-28
Disabling Conditional Execution Behavior	5-29
Displaying Execution Context Bars	5-29

Referencing a Model

6

Overview of Model Referencing	6-2
About Model Referencing	6-2
Referenced Model Advantages	6-4
Model Referencing Demos	6-5
Model Referencing Resources	6-6
Creating a Model Reference	6-7
Converting a Subsystem to a Referenced Model	6-10
Referenced Model Simulation Modes	6-12
About Referenced Model Simulation Modes	6-12
Specifying the Simulation Mode	6-13
Mixing Simulation Modes	6-14
Accelerating a Freestanding or Top Model	6-14
Viewing a Model Reference Hierarchy	6-16
Displaying Version Numbers	6-16
Model Reference Simulation Targets	6-17
About Simulation Targets	6-17
Building Simulation Targets	6-18

Simulink Model Referencing Requirements	6-20
About Model Referencing Requirements	6-20
Name Length Requirement	6-20
Configuration Parameter Requirements	6-20
Model Structure Requirements	6-25
Parameterizing Model References	6-27
Introduction	6-27
Global Nontunable Parameters	6-27
Global Tunable Parameters	6-28
Using Model Arguments	6-28
Defining Function-Call Models	6-34
About Function-Call Models	6-34
Function-Call Model Demo	6-34
Creating a Function-Call Model	6-34
Referencing a Function-Call Model	6-35
Function-Call Model Requirements	6-36
Using Model Reference Variants	6-38
Adapting Models to Different Contexts	6-38
About Model Reference Variants	6-39
Model Reference Variant Components	6-41
Model Reference Variant Example	6-42
Model Reference Variant Requirements	6-46
Implementing Model Reference Variants	6-47
Saving Variant Referenced Models	6-57
Changing Variant Model Definitions	6-57
Parameterizing Variant Models	6-58
Reusing Variant Objects and Models	6-59
Overriding Variant Conditions	6-59
Variant Models and Enumerated Types	6-60
Generating Code for Variant Models	6-61
Model Reference Variant Limitations	6-61
Protecting Referenced Models	6-63
About Protected Models	6-63
The Model Protection Facility	6-64
Model Protection Requirements	6-65
Protecting a Referenced Model	6-65
Packaging a Protected Model	6-69
Testing a Protected Model	6-70

Using a Protected Model	6-70
Model Protection Limitations	6-72
Inheriting Sample Times	6-73
Blocks That Depend on Absolute Time	6-74
Blocks Whose Outputs Depend on Inherited Sample Time	6-75
Refreshing Model Blocks	6-77
Simulink Model Referencing Limitations	6-78
Introduction	6-78
Limitations on All Model Referencing	6-78
Limitations on Normal Mode Referenced Models	6-81
Limitations on Accelerator Mode Referenced Models	6-81
Limitations on PIL Mode Referenced Models	6-84

Working with Blocks

7

About Blocks	7-2
What Are Blocks?	7-2
Block Data Tips	7-2
Virtual Blocks	7-2
Adding Blocks	7-4
Ways to Add Blocks	7-4
Adding Blocks by Browsing or Searching with the Library Browser	7-5
Copying Blocks from a Model	7-5
Adding Frequently Used Blocks	7-6
Adding Blocks Programmatically	7-8
Editing Blocks	7-9
Copying and Moving Blocks from One Window to Another	7-9
Moving Blocks in a Model	7-10
Copying Blocks in a Model	7-13

Deleting Blocks	7-14
Working with Block Parameters	7-15
About Block Parameters	7-15
Mathematical Versus Configuration Parameters	7-15
Setting Block Parameters	7-16
Specifying Numeric Parameter Values	7-17
Checking Parameter Values	7-19
Changing the Values of Block Parameters During Simulation	7-23
Inlining Parameters	7-25
Block Properties Dialog Box	7-27
State Attributes Pane of Block Parameters Dialog Box ...	7-33
Changing a Block's Appearance	7-34
Changing a Block's Orientation	7-34
Resizing a Block	7-37
Displaying Parameters Beneath a Block	7-38
Using Drop Shadows	7-38
Manipulating Block Names	7-39
Specifying a Block's Color	7-40
Displaying Block Outputs	7-41
Block Output Data Tips	7-41
Setting Block Output Data Tip Options	7-42
Enabling Block Output Display	7-42
Controlling the Block Output Display	7-43
Port Value Display Limitations	7-44
Controlling and Displaying the Sorted Order	7-46
What Is Sorted Order?	7-46
Displaying the Sorted Order	7-46
Sorted Order Notation	7-48
How Simulink Determines the Sorted Order	7-51
Assigning Block Priorities	7-52
Rules for Priorities	7-53
Accessing Block Data During Simulation	7-56
About Block Run-Time Objects	7-56
Accessing a Run-Time Object	7-56
Listening for Method Execution Events	7-57

Synchronizing Run-Time Objects and Simulink Execution	7-58
---	------

Working with Block Libraries

8

About Block Libraries	8-2
Working with Reference Blocks	8-3
About Reference Blocks	8-3
Creating a Reference Block	8-3
Updating a Reference Block	8-4
Modifying Reference Blocks	8-4
Finding a Reference Block's Library Block Prototype	8-5
Getting Information About Library Blocks Referenced by a Model	8-5
Working with Library Links	8-6
Displaying Library Links	8-6
Disabling Links to Library Blocks	8-7
Restoring Disabled or Parameterized Links	8-7
Determining Link Status	8-10
Breaking a Link to a Library Block	8-11
Fixing Unresolved Library Links	8-12
Creating Block Libraries	8-14
Creating a Library	8-14
Creating a Sublibrary	8-15
Modifying a Library	8-15
Locking Libraries	8-16
Making Backward-Compatible Changes to Libraries	8-16
Adding Libraries to the Library Browser	8-24
How to Display a Library in the Library Browser	8-24
Example of a Minimal sblocks.m File	8-24
Adding More Descriptive Information in sblocks.m	8-25

What Are Masks?	9-2
Why Use a Mask?	9-3
Masked Subsystem Example	9-5
Roadmap for Masking Blocks	9-8
Mask Terminology	9-10
Creating a Block Mask	9-11
Introduction	9-11
Opening the Mask Editor	9-12
Defining a Mask Icon	9-13
Understanding Mask Parameters	9-16
Defining Mask Parameters	9-18
Defining Mask Documentation	9-21
Using a Block Mask	9-25
Masking a Model Block	9-28
Masks on Blocks in User Libraries	9-29
About Masks and User-Defined Libraries	9-29
Masking a Block for Inclusion in a User Library	9-29
Masking a Block that Resides in a User Library	9-29
Masking a Block Copied from a User Library	9-30
Operating on Existing Masks	9-31
Changing a Block Mask	9-31
Viewing Mask Parameters	9-31
Looking Under a Block Mask	9-31
Removing and Caching a Mask	9-32
Restoring a Cached Mask	9-33
Permanently Deleting a Mask	9-33
Roadmap for Dynamic Masks	9-34

Calculating Values Used Under the Mask	9-35
Controlling Masks Programmatically	9-39
Predefined Masked Dialog Parameters	9-40
Notes on Mask Parameter Storage	9-43
Understanding Mask Code Execution	9-44
Specifying Mask Initialization	9-44
Callback	9-48
Mask Initialization Code	9-50
Debugging Masks that Use M-Code	9-50
Creating Dynamic Mask Dialog Boxes	9-51
Setting Nested Masked Block Parameters	9-51
About Dynamic Masked Dialog Boxes	9-51
Show parameter	9-52
Enable parameter	9-52
Setting Masked Block Dialog Parameters	9-52
Creating Dynamic Masked Subsystems	9-55
Allow library block to modify its contents	9-55
Creating Self-Modifying Masks for Library Blocks	9-55
Best Practices for Using M-Code in Masks	9-59

Working with Signals

10

Signal Basics	10-2
About Signals	10-2
Creating Signals	10-3
Naming Signals	10-3
Displaying Signal Values	10-5
Signal Line Styles	10-6
Signal Labels	10-7
Signal Data Types	10-8
Signal Dimensions	10-8
Complex Signals	10-12

Virtual Signals	10-12
Mux Signals	10-15
Control Signals	10-18
Composite Signals	10-18
Signal Glossary	10-19
Validating Signal Connections	10-20
Displaying Signal Sources and Destinations	10-21
About Signal Highlighting	10-21
Highlighting Signal Sources	10-21
Highlighting Signal Destinations	10-22
Removing Highlighting	10-23
Resolving Incomplete Highlighting to Library Blocks	10-23
Determining Output Signal Dimensions	10-25
About Signal Dimensions	10-25
Determining the Output Dimensions of Source Blocks	10-25
Determining the Output Dimensions of Nonsource Blocks	10-26
Signal and Parameter Dimension Rules	10-26
Scalar Expansion of Inputs and Parameters	10-28
Checking Signal Ranges	10-30
About Signal Range Checking	10-30
Blocks That Allow Signal Range Specification	10-30
Specifying Ranges for Signals	10-31
Checking for Signal Range Errors	10-32
Introducing the Signal and Scope Manager	10-37
What is the Signal & Scope Manager?	10-37
Displaying the Signal and Scope Manager User Interface	10-38
Understanding the Signal and Scope Manager User Interface	10-38
Using the Signal and Scope Manager	10-43
Introduction	10-43
Attaching a New Viewer or Generator	10-43
Creating a Multiple Axes Viewer	10-44
Adding Additional Signals to an Existing Viewer	10-45

Viewing Test Point Data	10-45
Adding Custom Viewers and Generators	10-46
The Signal Selector	10-48
About the Signal Selector	10-48
Port/Axis Selector	10-49
Model Hierarchy	10-50
Inputs/Signals List	10-50
Initializing Signals and Discrete States	10-53
About Initialization	10-53
Using Block Parameters to Initialize Signals and Discrete States	10-54
Using Signal Objects to Initialize Signals and Discrete States	10-54
Using Signal Objects to Tune Initial Values	10-55
Example: Using a Signal Object to Initialize a Subsystem Output	10-57
Initialization Behavior Summary for Signal Objects	10-58
Working with Test Points	10-61
About Test Points	10-61
Designating a Signal as a Test Point	10-61
Displaying Test Point Indicators	10-62
Displaying Signal Properties	10-64
Port/Signal Displays Menu	10-64
Port Data Types	10-65
Signal Dimensions	10-65
Signal Resolution Indicators	10-66
Wide Nonscalar Lines	10-67
Working with Signal Groups	10-69
About the Signal Groups	10-69
Creating a Signal Group Set	10-69
Signal Builder Dialog Box	10-71
Editing Signal Groups	10-73
Editing Signals	10-73
Editing Waveforms	10-76
Signal Builder Time Range	10-82
Exporting Signal Group Data	10-83
Printing, Exporting, and Copying Waveforms	10-83

Simulating with Signal Groups	10-84
Simulation Options Dialog Box	10-85

Working with Variable-Size Signals

11

Variable-Size Signal Basics	11-2
About Variable-Size Signals	11-2
Creating Variable-Size Signals	11-2
How Variable-Size Signals Propagate	11-3
Empty Signals	11-4
Subsystem Initialization of Variable-Size Signals	11-4
Simulink Models Using Variable-Size Signals	11-6
Demo of Variable-Size Signal Generation and Operations	11-6
Demo of Variable-Size Signal Length Adaptation	11-10
Demo of Mode-Dependent Variable-Size Signals	11-13
S-Functions Using Variable-Size Signals	11-19
Demo of Level-2 M-File S-Function with Variable-Size Signals	11-19
Demo of C-File S-Function with Variable-Size Signals ...	11-20
Simulink Block Support for Variable-Size Signals	11-22
Discrete	11-22
Logic and Bit Operations	11-22
Math Operations	11-22
Subsystem Blocks	11-23
Conditionally Executed Subsystem Blocks	11-24
Signal Attributes	11-25
Signal Routing	11-25
Switching Blocks	11-25
Source Blocks	11-26
Sink Blocks	11-26
User-Defined Function Blocks	11-27
Signal Processing Blockset	11-27
Video and Image Processing Blockset	11-28

Using Composite Signals

12

About Composite Signals	12-2
Composite Signal Terminology	12-2
Types of Simulink Buses	12-3
Buses and Muxes	12-3
Bus Objects	12-4
Bus Code	12-4
Creating and Accessing a Bus	12-5
Nesting Buses	12-7
Circular Bus Definitions	12-8
Bus-Capable Blocks	12-9
Using Bus Objects	12-10
About Bus Objects	12-10
Bus Object Capabilities	12-11
Associating Bus Objects with Simulink Blocks	12-11
Using the Bus Editor	12-13
Introduction	12-13
Opening the Bus Editor	12-14
Displaying Bus Objects	12-15
Creating Bus Objects	12-17
Creating Bus Elements	12-20
Nesting Bus Definitions	12-23
Changing Bus Entities	12-25
Exporting Bus Objects	12-30
Importing Bus Objects	12-32
Closing the Bus Editor	12-32
Filtering Displayed Bus Objects	12-34
Filtering by Name	12-35

Filtering by Relationship	12-36
Changing Filtered Objects	12-38
Clearing the Filter	12-39
Customizing Bus Object Import and Export	12-40
Prerequisites for Customization	12-41
Writing a Bus Object Import Function	12-41
Writing a Bus Object Export Function	12-42
Viewing the Customization Manager	12-43
Registering Customizations	12-44
Changing Customizations	12-45
Using the Bus Object API	12-47
Virtual and Nonvirtual Buses	12-48
Introduction	12-48
Creating Nonvirtual Buses	12-48
Nonvirtual Bus Sample Times	12-49
Automatic Bus Conversion	12-50
Connecting Buses to Inports and Outports	12-51
Connecting Buses to Root Level Inports	12-51
Connecting Buses to Root Level Outports	12-51
Connecting Buses to Nonvirtual Inports	12-52
Connecting Buses to Model, Stateflow, and Embedded MATLAB Blocks	12-54
Connecting Multi-Rate Buses to Referenced Models	12-54
Buses and Libraries	12-56
Avoiding Mux/Bus Mixtures	12-57
Introduction	12-57
Using Diagnostics for Mux/Bus Mixtures	12-58
Using the Model Advisor for Mux/Bus Mixtures	12-60
Correcting Buses Used as Muxes	12-62
Bus to Vector Block Backward Compatibility	12-63
Avoiding Mux/Bus Mixtures When Developing Models ...	12-63
Buses in Generated Code	12-64

Working with Data

13

Working with Data Types	13-2
About Data Types	13-2
Data Types Supported by Simulink	13-3
Fixed-Point Data	13-4
Enumerated Data	13-6
Block Support for Data and Numeric Signal Types	13-6
Creating Signals of a Specific Data Type	13-6
Specifying Block Output Data Types	13-6
Using the Data Type Assistant	13-14
Displaying Port Data Types	13-23
Data Type Propagation	13-24
Data Typing Rules	13-24
Typecasting Signals	13-25
Working with Data Objects	13-26
About Data Object Classes	13-26
About Data Object Methods	13-27
Using the Model Explorer to Create Data Objects	13-29
About Object Properties	13-31
Changing Object Properties	13-31
Handle Versus Value Classes	13-33
Comparing Data Objects	13-35
Saving and Loading Data Objects	13-35
Using Data Objects in Simulink Models	13-36
Creating Persistent Data Objects	13-36
Data Object Wizard	13-36
Subclassing Simulink Data Classes	13-40
About Packages and Data Classes	13-40
Working with Packages	13-41
Working with Classes	13-45
Enumerated Property Types	13-53
Enabling Custom Storage Classes	13-56

Using Enumerated Data

14

Enumerated Data in Simulink	14-2
Demos of Enumerated Data Types	14-3
Defining an Enumerated Data Type	14-4
Workflow for Defining Enumerated Data	14-4
Creating a Class Definition File	14-5
Enumerated Class Definition Syntax	14-5
Enumerated Class Definition Example	14-5
Defining Enumerated Values	14-6
Overriding Default Methods (Optional)	14-7
Defining Additional Customizations	14-8
Saving the M-File on the MATLAB Path	14-8
Changing an Enumerated Data Type	14-10
Using Enumerated Data in a Simulink Model	14-11
Simulating with Enumerated Types	14-11
Referencing an Enumerated Type	14-14
Instantiating an Enumerated Type	14-15
Displaying Enumerated Values	14-17
Enumerated Values in Computation	14-18
Simulink Constructs that Support Enumerated Types	14-22
Overview	14-22
Block Support	14-22
Class Support	14-23
Logging Enumerated Data	14-24
Importing Enumerated Data	14-24
Simulink Enumerated Type Limitations	14-25
Enumerated Types and Bus Initialization	14-25

Enumerated Types on Scopes	14-26
Enumerated Types for Switch Blocks	14-26
Nonsupport of Enumerated Types	14-27

Importing and Exporting Data

15

Introduction	15-2
Logging Signals	15-3
About Signal Logging	15-3
Globally Enabling and Disabling Logging	15-4
Enabling Logging for a Signal	15-4
Displaying Logging Indicators	15-5
Specifying a Logging Name	15-5
Limiting the Data Logged for a Signal	15-6
Logging Virtual Signals	15-6
Logging Multidimensional Signals	15-6
Logging Composite Signals	15-7
Logging Referenced Model Signals	15-7
Viewing Logged Signal Data	15-8
Accessing Logged Signal Data	15-9
Handling Spaces and Newlines in Logged Names	15-9
Extracting Partial Data from a Running Simulation	15-12
Example: Logging Signal Data in the F14 Model	15-12
Signal Logging Limitations	15-15
Importing Data from a Workspace	15-16
Enabling Data Import	15-16
Importing Time-Series Data	15-17
Importing Data Arrays	15-18
Using a MATLAB Time Expression to Import Data	15-19
Importing Data Structures	15-20
Specifying Time Vectors for Discrete Systems	15-22
Exporting Data to the MATLAB Workspace	15-24
Enabling Data Export	15-24
Format Options	15-25

Importing and Exporting States	15-29
Introduction	15-29
Saving Final States	15-29
Loading Initial States	15-30
Limiting Output	15-32
Specifying Output Options	15-33
Introduction	15-33
Refining Output	15-33
Producing Additional Output	15-34
Producing Specified Output Only	15-34
Comparing Output Options	15-35

Working with Data Stores

16

About Data Stores	16-2
Introduction	16-2
When to Use a Data Store	16-3
Creating Data Stores	16-3
Accessing Data Stores	16-4
Workflow for Configuring Data Stores	16-4
Defining Data Stores with Data Store Memory	
Blocks	16-6
Creating the Data Store	16-6
Specifying Data Store Memory Block Attributes	16-6
Defining Data Stores with Signal Objects	16-9
Creating the Data Store	16-9
Local and Global Data Stores	16-9
Specifying Signal Object Data Store Attributes	16-9
Accessing Data Stores with Simulink Blocks	16-11
Data Store Examples	16-13
Overview	16-13

Local Data Store Example	16-13
Global Data Store Example	16-14
Ordering Data Store Access	16-16
About Data Store Access Order	16-16
Ordering Access Using Function Call Subsystems	16-16
Ordering Access Using Block Priorities	16-17
Using Data Store Diagnostics	16-19
About Data Store Diagnostics	16-19
Detecting Access Order Errors	16-19
Detecting Multitasking Access Errors	16-22
Detecting Duplicate Name Errors	16-24
Data Store Diagnostics in the Model Advisor	16-26
Data Stores and Software Verification	16-27

Working with Lookup Tables

17

About Lookup Table Blocks	17-2
Anatomy of a Lookup Table	17-5
Lookup Tables Block Library	17-6
Guidelines for Choosing a Lookup Table	17-8
Data Set Dimensionality	17-8
Data Set Numeric and Data Types	17-8
Data Accuracy and Smoothness	17-8
Dynamics of Table Inputs	17-9
Efficiency of Performance	17-9
Summary of Lookup Table Block Features	17-10
Entering Breakpoints and Table Data	17-11
Entering Data in a Block Parameter Dialog Box	17-11
Entering Data in the Lookup Table Editor	17-13

Entering Data Using the Lookup Table Dynamic Block's Inports	17-16
Characteristics of Lookup Table Data	17-18
Sizes of Breakpoint Data Sets and Table Data	17-18
Monotonicity of Breakpoint Data Sets	17-19
Formulation of Evenly-Spaced Breakpoints	17-20
Representation of Discontinuities in Lookup Tables	17-20
Methods for Estimating Missing Points	17-23
About Estimating Missing Points	17-23
Interpolation Methods	17-23
Extrapolation Methods	17-24
Rounding Methods	17-25
Example Output for Lookup Methods	17-26
Lookup Table Editor	17-28
When to Use the Lookup Table Editor	17-28
Layout of the LUT Editor	17-28
Browsing LUT Blocks	17-30
Editing Table Values	17-31
Adding and Removing Rows and Columns in a Table	17-31
Displaying N-Dimensional Tables in the Editor	17-32
Plotting LUT Tables	17-35
Editing Custom LUT Blocks	17-36
Example of a Logarithm Lookup Table	17-39
Examples for Prelookup and Interpolation Blocks	17-43
Lookup Table Glossary	17-44

Modeling with Simulink

18

General Considerations when Building Simulink

Models	18-2
Avoiding Invalid Loops	18-2

Shadowed Files	18-4
Model Building Tips	18-6
Modeling a Continuous System	18-8
Best-Form Mathematical Models	18-11
Series RLC Example	18-11
Solving Series RLC Using Resistor Voltage	18-12
Solving Series RLC Using Inductor Voltage	18-13
Example: Converting Celsius to Fahrenheit	18-15

Exploring, Searching, and Browsing Models

19

The Model Explorer	19-2
Introduction to the Model Explorer	19-2
Model Hierarchy Pane	19-3
Contents Pane	19-6
Dialog Pane	19-11
Main Toolbar	19-12
Search Bar	19-14
Setting the Model Explorer's Font Size	19-19
The Finder	19-20
About the Finder	19-20
Filter Options	19-22
Search Criteria	19-23
The Model Browser	19-26
About the Model Browser	19-26
Navigating with the Mouse	19-28
Navigating with the Keyboard	19-28
Showing Library Links	19-28
Showing Masked Subsystems	19-28
Model Dependencies	19-29
What Are Model Dependencies?	19-29

Generating Manifests	19-30
Command-Line Dependency Analysis	19-35
Editing Manifests	19-38
Comparing Manifests	19-41
Exporting Files in a Manifest	19-42
Scope of Dependency Analysis	19-44
Best Practices for Dependency Analysis	19-47
Using the Model Manifest Report	19-48
Using the Model Dependency Viewer	19-52

Managing Configuration Sets

20

Setting Up Configuration Sets	20-2
About Configuration Sets	20-2
Configuration Set Components	20-3
The Active Set	20-3
Displaying Configuration Sets	20-3
Activating a Configuration Set	20-4
Opening Configuration Sets	20-4
Copying, Deleting, and Moving Configuration Sets	20-5
Copying Configuration Set Components	20-6
Creating Configuration Sets	20-6
Setting Values in Configuration Sets	20-7
Configuration Set API	20-7
Model Configuration Dialog Box	20-10
Model Configuration Preferences Dialog Box	20-11
Referencing Configuration Sets	20-12
Overview of Configuration References	20-12
Creating a Freestanding Configuration Set	20-16
Creating and Attaching a Configuration Reference	20-17
Obtaining a Configuration Reference Handle	20-21
Attaching a Configuration Reference to Other Models	20-22
Changing a Configuration Reference	20-23
Activating a Configuration Reference	20-24
Unresolved Configuration References	20-24
Getting Values from a Referenced Configuration Set	20-25
Changing Values in a Referenced Configuration Set	20-25
Replacing a Referenced Configuration Set	20-27

Building Models and Generating Code	20-28
Configuration Reference Limitations	20-28
Configuration References for Models with Older Simulation Target Settings	20-29

Running Simulations

21

Simulation Basics	21-2
Controlling Execution of a Simulation	21-3
Starting a Simulation	21-3
Pausing or Stopping a Simulation	21-5
Using Blocks to Stop or Pause a Simulation	21-5
Specifying a Simulation Start and Stop Time	21-8
Choosing a Solver	21-9
What Is a Solver?	21-9
Choosing a Solver Type	21-9
Choosing a Fixed-Step Solver	21-11
Choosing a Variable-Step Solver	21-15
Interacting with a Running Simulation	21-19
Saving and Restoring the Simulation State as the SimState	21-20
Overview of the SimState	21-20
How to Save the SimState	21-21
How to Restore the SimState	21-22
How to Change the States of a Block within the SimState	21-24
Using the SimState Interface Checksum Diagnostic	21-24
Limitations of the SimState	21-25
Using SimState within S-Functions	21-26
Diagnosing Simulation Errors	21-27
Response to Run-Time Errors	21-27

Simulation Diagnostics Viewer	21-27
Creating Custom Simulation Error Messages	21-30

Running a Simulation Programmatically

22

About Programmatic Simulation	22-2
Using the sim Command	22-3
Single-Output Syntax for the sim Command	22-3
Examples of Implementing the sim Command	22-4
Calling sim from within parfor	22-5
Using the set_param Command	22-5
Running a Simulation from an S-Function	22-6
Running Parallel Simulations	22-7
Overview of Calling sim from within parfor	22-7
sim in parfor with Rapid Accelerator Mode	22-7
Workspace Access Issues	22-9
Resolving Workspace Access Issues	22-9
Data Concurrency Issues	22-10
Resolving Data Concurrency Issues	22-11
Error Handling in Simulink Using MSLErrorException	22-14
Error Reporting in a Simulink Application	22-14
The MSLErrorException Class	22-14
Methods of the MSLErrorException Class	22-14
Capturing Information about the Error	22-14

Improving Simulation Performance and Accuracy

23

About Improving Performance and Accuracy	23-2
---	-------------

Speeding Up the Simulation	23-2
Comparing Performance	23-4
Performance of the Simulation Modes	23-4
Measuring Performance	23-6
Improving Acceleration Mode Performance	23-8
Techniques	23-8
C Compilers	23-9
Improving Simulation Accuracy	23-10

Visualizing Simulation Results

24

About Scope Blocks, Viewers, Signal Logging, and Test Points	24-2
What Are Scope Blocks, Signal Viewers, Test Points and Data Logging?	24-2
How Scope Blocks and Signal Viewers Differ	24-3
Why Use Generators and Signal Viewers Instead of Source and Scope Blocks?	24-4
Methods for Attaching a Generator or Viewer	24-6
Displaying a Scope Viewer	24-7
Things to Know When Using Viewers	24-9
About Viewers	24-9
How the Viewer Determines Trace Color Coding and Line Styles	24-9
How Scope Viewer Parameter Settings Can Affect Performance	24-10
Changing Viewer Characteristics	24-12
The Scope Viewer Toolbar	24-12
Scope Viewer Parameters Dialog Box	24-13

Scope Viewer Context Menu	24-17
Performing Common Viewer Tasks	24-18
Viewing Scope Viewer Help	24-18
Attaching a Scope Viewer	24-18
Adding Multiple Signals to a Scope Viewer	24-19
Adding a Legend	24-19
Zooming In On Graph Regions	24-19
Displaying Multiple Axes	24-20
How to Save Data to MATLAB Workspace	24-22
Saving the Viewer Data to a File	24-22
Plotting the Viewer Data	24-23
Performing Common Generator Tasks	24-24
Attaching a Generator	24-24
Removing a Generator	24-24

Analyzing Simulation Results

25

Viewing Output Trajectories	25-2
Using the Scope Block	25-2
Using Return Variables	25-2
Using the To Workspace Block	25-3
Linearizing Models	25-4
About Linearizing Models	25-4
Linearization with Referenced Models	25-6
Linearization Using the 'v5' Algorithm	25-8
Finding Steady-State Points	25-10

Introduction to the Debugger	26-2
Using the Debugger's Graphical User Interface	26-3
Displaying the Graphical Interface	26-3
Toolbar	26-4
Breakpoints Pane	26-6
Simulation Loop Pane	26-7
Outputs Pane	26-8
Sorted List Pane	26-9
Status Pane	26-10
Using the Debugger's Command-Line Interface	26-11
Controlling the Debugger	26-11
Method ID	26-11
Block ID	26-11
Accessing the MATLAB Workspace	26-12
Getting Online Help	26-13
Starting the Debugger	26-14
Starting a Simulation	26-15
Running a Simulation Step by Step	26-19
Introduction	26-19
Block Data Output	26-20
Stepping Commands	26-21
Continuing a Simulation	26-22
Running a Simulation Nonstop	26-24
Debug Pointer	26-25
Setting Breakpoints	26-27
About Breakpoints	26-27
Setting Unconditional Breakpoints	26-27
Setting Conditional Breakpoints	26-30

Displaying Information About the Simulation	26-33
Displaying Block I/O	26-33
Displaying Algebraic Loop Information	26-35
Displaying System States	26-36
Displaying Solver Information	26-36
Displaying Information About the Model	26-38
Displaying a Models Sorted Lists	26-38
Displaying a Block	26-39

Accelerating Models

27

What Is Acceleration?	27-2
How the Acceleration Modes Work	27-3
Overview	27-3
Normal Mode	27-3
Accelerator Mode	27-4
Rapid Accelerator Mode	27-5
Code Regeneration in Accelerated Models	27-7
Structural Changes That Cause Rebuilds	27-7
Determining If the Simulation Will Rebuild	27-7
Choosing a Simulation Mode	27-10
Tradeoffs	27-10
Comparing Modes	27-11
Decision Tree	27-12
Designing Your Model for Effective Acceleration	27-14
Selecting Blocks for Accelerator Mode	27-14
Selecting Blocks for Rapid Accelerator Mode	27-15
Controlling S-Function Execution	27-15
Accelerator and Rapid Accelerator Mode Data Type Considerations	27-16
Using Scopes and Viewers with Rapid Accelerator Mode ..	27-16
Factors Inhibiting Acceleration	27-17

Performing Acceleration	27-20
Customizing the Build Process	27-20
Running Acceleration Mode from the User Interface	27-21
Making Run-Time Changes	27-23
Interacting with the Acceleration Modes	
Programmatically	27-24
Why Interact Programmatically?	27-24
Building Accelerator Mode MEX-files	27-24
Controlling Simulation	27-24
Simulating Your Model	27-25
Customizing the Acceleration Build Process	27-26
Using the Accelerator Mode with the Simulink	
Debugger	27-27
Advantages of Using Accelerator Mode with the Debugger	27-27
How to Run the Debugger	27-27
When to Switch Back to Normal Mode	27-28
Capturing Performance Data	27-29
What Is the Profiler?	27-29
How the Profiler Works	27-29
Enabling the Profiler	27-31
How to Save Simulink Profiler Results	27-34

Customizing the Simulink User Interface

28

Adding Items to Model Editor Menus	28-2
About Adding Items to the Model Editor Menus	28-2
Code Example	28-2
Defining Menu Items	28-4
Registering Menu Customizations	28-9
Callback Info Object	28-10
Debugging Custom Menu Callbacks	28-10
About Menu Tags	28-10
Disabling and Hiding Model Editor Menu Items	28-13

About Disabling and Hiding Model Editor Menu Items . . .	28-13
Example: Disabling the New Model Command on the Simulink Editor's File Menu	28-13
Creating a Filter Function	28-13
Registering a Filter Function	28-14
Disabling and Hiding Dialog Box Controls	28-15
About Disabling and Hiding Controls	28-15
Example: Disabling a Button on a Simulink Dialog Box . .	28-16
Writing Control Customization Callback Functions	28-17
Dialog Box Methods	28-17
Dialog Box and Widget IDs	28-18
Registering Control Customization Callback Functions . . .	28-19
Customizing the Library Browser	28-21
Reordering Libraries	28-21
Disabling and Hiding Libraries	28-21
Customizing the Library Browser's Menu	28-22
Registering Customizations	28-24
About Registering User Interface Customizations	28-24
Customization Manager	28-24

Creating Custom Blocks

29

When to Create Custom Blocks	29-2
Types of Custom Blocks	29-3
MATLAB Function Blocks	29-3
Subsystem Blocks	29-4
S-Function Blocks	29-4
Comparison of Custom Block Functionality	29-7
Custom Block Considerations	29-7
Modeling Requirements	29-10
Speed and Code Generation Requirements	29-13

Expanding Custom Block Functionality	29-17
Tutorial: Creating a Custom Block	29-18
How to Design a Custom Block	29-18
Defining Custom Block Behavior	29-20
Deciding on a Custom Block Type	29-21
Placing Custom Blocks in a Library	29-26
Adding a Graphical User Interface to a Custom Block	29-28
Adding Block Functionality Using Block Callbacks	29-37
Custom Block Examples	29-44
Creating Custom Blocks from Masked Library Blocks	29-44
Creating Custom Blocks from MATLAB Functions	29-44
Creating Custom Blocks from S-Functions	29-45

Using the Embedded MATLAB Function Block

30

Introduction to Embedded MATLAB Function	
Blocks	30-3
What Is an Embedded MATLAB Function Block?	30-3
Why Use Embedded MATLAB Function Blocks?	30-6
Creating an Example Embedded MATLAB Function ..	30-8
Adding an Embedded MATLAB Function Block to a Model	30-8
Programming the Embedded MATLAB Function	30-10
Building the Function and Checking for Errors	30-15
Defining Inputs and Outputs	30-19
Debugging an Embedded MATLAB Function Block ...	30-23
How Debugging Affects Simulation Speed	30-23
Enabling and Disabling Debugging	30-23
Debugging the Function in Simulation	30-24
Watching Function Variables During Simulation	30-31
Checking for Data Range Violations	30-34
Debugging Tools	30-34

Embedded MATLAB Function Editor	30-37
Customizing the Embedded MATLAB Editor	30-37
Embedded MATLAB Editor Tools	30-37
Editing and Debugging Embedded MATLAB Code	30-38
Ports and Data Manager	30-42
Working with Compilation Reports	30-66
About Compilation Reports	30-66
Location of Compilation Reports	30-67
Opening Compilation Reports	30-67
Description of Compilation Reports	30-68
Viewing Your Embedded MATLAB Function Code	30-69
Viewing Call Stack Information	30-70
Viewing the Compilation Summary Information	30-71
Viewing Error and Warning Messages	30-71
Viewing Variables in Your M-Code	30-73
Keyboard Shortcuts for the Compilation Report	30-77
Compilation Report Limitations	30-78
Typing Function Arguments	30-81
About Function Arguments	30-81
Specifying Argument Types	30-81
Inheriting Argument Data Types	30-84
Built-In Data Types for Arguments	30-86
Specifying Argument Types with Expressions	30-86
Specifying Simulink® Fixed Point Data Properties	30-87
Sizing Function Arguments	30-93
Specifying Argument Size	30-93
Inheriting Argument Sizes from Simulink	30-93
Specifying Argument Sizes with Expressions	30-95
Parameter Arguments in Embedded MATLAB	
Functions	30-97
Resolving Signal Objects for Output Data	30-98
Implicit Signal Resolution	30-98
Eliminating Warnings for Implicit Signal Resolution in the Model	30-98
Disabling Implicit Signal Resolution for an Embedded MATLAB Function Block	30-99

Forcing Explicit Signal Resolution for an Output Data Signal	30-99
Working with Structures and Bus Signals	30-100
About Structures in Embedded MATLAB Function	
Blocks	30-100
Example of Structures in an Embedded MATLAB Function	
Block	30-101
How Structure Inputs and Outputs Interface with Bus	
Signals	30-105
Rules for Defining Structures in Embedded MATLAB	
Function Blocks	30-106
Workflow for Creating Structures in Embedded MATLAB	
Function Blocks	30-106
Indexing Substructures and Fields	30-107
Assigning Values to Structures and Fields	30-108
Working with Non-Tunable Structure Parameters in	
Embedded MATLAB Function Blocks	30-110
Limitations of Structures in Embedded MATLAB Function	
Blocks	30-113
 Using Variable-Size Data in Embedded MATLAB	
Function Blocks	30-114
What Is Variable-Size Data?	30-114
How Embedded MATLAB Function Blocks Implement	
Variable-Size Data	30-114
Enabling Support for Variable-Size Data	30-115
Declaring Variable-Size Inputs and Outputs	30-115
Declaring Variable-Size Data Locally	30-115
Simple Example: Defining and Using Variable-Size Data in	
Embedded MATLAB Function Blocks	30-116
Limitations of Variable-Size Support in Embedded	
MATLAB Function Blocks	30-122
 Using Enumerated Data in Embedded MATLAB	
Function Blocks	30-123
Enumerated Data in Embedded MATLAB Function	
Blocks	30-123
When to Use Enumerated Data	30-124
Simple Example: Defining and Using Enumerated Types in	
Embedded MATLAB Function Blocks	30-125
Using Enumerated Data in Embedded MATLAB Function	
Blocks	30-129

How to Define Enumerated Data Types for Embedded MATLAB Function Blocks	30-130
How to Add Enumerated Data to Embedded MATLAB Function Blocks	30-130
How to Instantiate Enumerated Data in Embedded MATLAB Function Blocks	30-132
Operations on Enumerated Data	30-133
Limitations of Enumerated Types	30-133
Working with Frame-Based Signals	30-134
About Frame Based Signals	30-134
Supported Types for Frame-Based Data	30-135
Adding Frame-Based Data in Embedded MATLAB Function Blocks	30-135
Examples of Frame-Based Signals in Embedded MATLAB Function Blocks	30-136
Using Traceability in Embedded MATLAB Function Blocks	30-141
Extent of Traceability in Embedded MATLAB Function Blocks	30-141
Traceability Requirements	30-141
Basic Workflow for Using Traceability	30-141
Tutorial: Using Traceability in an Embedded MATLAB Function Block	30-143
Enhancing Readability of Generated Code for Embedded MATLAB Function Blocks	30-148
Requirements for Using Readability Optimizations	30-148
Converting If-Elseif-Else Code to Switch-Case Statements	30-148
Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements	30-151
Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library	30-157
How Embedded MATLAB Function Blocks Use the BLAS Library	30-157
When to Disable BLAS Library Support	30-157
How to Disable BLAS Library Support	30-158
Supported Compilers	30-158

Controlling Runtime Checks	30-159
Types of Runtime Checks	30-159
When to Disable Runtime Checks	30-160
How to Disable Runtime Checks	30-160

PrintFrame Editor

31

PrintFrame Editor Overview	31-2
About the Print Frame Editor	31-2
What PrintFrames Are	31-3
Starting the PrintFrame Editor	31-6
Getting Help for the PrintFrame Editor	31-7
Closing the PrintFrame Editor	31-7
Print Frame Process	31-7
Designing the Print Frame	31-8
Before You Begin	31-8
Variable and Static Information	31-8
Single Use or Multiple Use Print Frames	31-8
Specifying the Print Frame Page Setup	31-9
Creating Borders (Rows and Cells)	31-11
First Steps	31-11
Adding and Removing Rows	31-11
Adding and Removing Cells	31-12
Resizing Rows and Cells	31-12
Print Frame Size	31-12
Adding Information to Cells	31-14
Procedure for Adding Information to Cells	31-14
Text Information	31-15
Variable Information	31-15
Multiple Entries in a Cell	31-16
Changing Information in Cells	31-18
Aligning the Information in a Cell	31-18

Editing Text Strings	31-18
Removing and Copying Entries	31-19
Changing the Font Characteristics	31-20
Saving and Opening Print Frames	31-22
Saving a Print Frame	31-22
Opening a Print Frame	31-22
Printing Block Diagrams with Print Frames	31-23
Example	31-26
About the Example	31-26
Create the Print Frame	31-27
Print the Block Diagram with the Print Frame	31-30

Glossary

Examples

A

Simulink Basics	A-2
How Simulink Works	A-2
Creating a Model	A-2
Executing Commands From Models	A-2
Working with Blocks	A-3
Creating Block Masks	A-3

Working with Lookup Tables	A-3
Creating Custom Simulink Blocks	A-3

Index


Simulink Basics

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

- “Starting Simulink Software” on page 1-2
- “Opening a Model” on page 1-4
- “Loading a Model” on page 1-7
- “Saving a Model” on page 1-8
- “Using the Model Editor” on page 1-14
- “Undoing a Command” on page 1-22
- “Zooming Block Diagrams” on page 1-22
- “Panning Block Diagrams” on page 1-23
- “Viewing Command History” on page 1-25
- “Bringing the MATLAB Software Desktop Forward” on page 1-25
- “Copying Models to Third-Party Applications” on page 1-26
- “Updating a Block Diagram” on page 1-27
- “Printing a Block Diagram” on page 1-29
- “Generating a Model Report” on page 1-39
- “Ending a Simulink Session” on page 1-42
- “Summary of Mouse and Keyboard Actions” on page 1-43

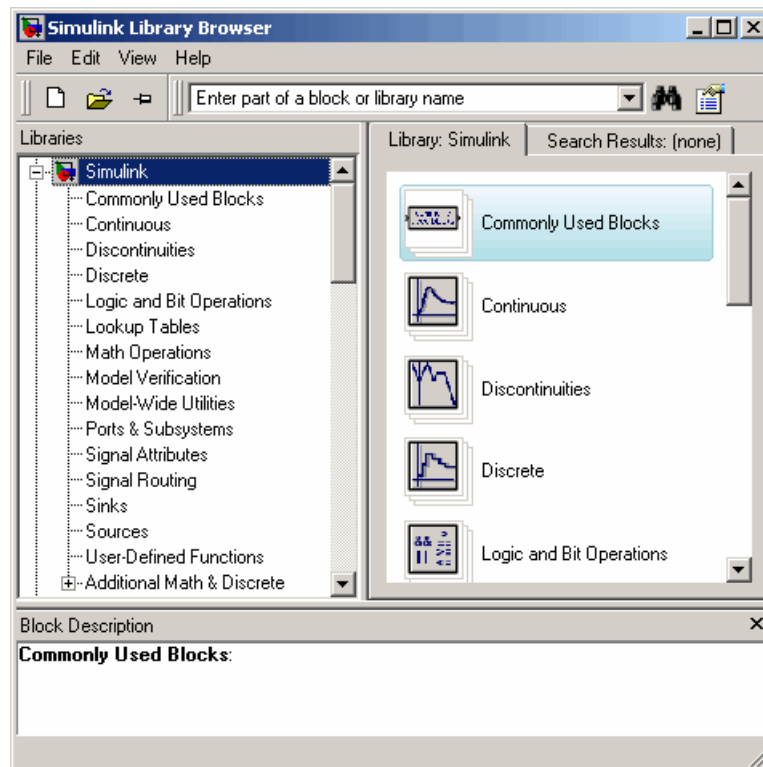
Starting Simulink Software

To start the Simulink software, you must first start the MATLAB® technical computing environment. Consult your MATLAB documentation for more information. You can then start the Simulink software in two ways:

- On the toolbar, click the Simulink icon. 
- Enter the `simulink` command at the MATLAB prompt.

The Library Browser appears. It displays a tree-structured view of the Simulink block libraries installed on your system. You build models by copying blocks from the Library Browser into a model window (see “Editing Blocks”).

The Simulink library window displays icons representing the pre-installed block libraries. You can create models by copying blocks from the library into a model window.



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.

Opening a Model

In this section...

“Introduction” on page 1-4

“Opening an Existing Model” on page 1-4

“Opening Models with Different Character Encodings” on page 1-4

“Avoiding Initial Model Open Delay” on page 1-5

Introduction

Opening a model both brings the model into memory and displays the model graphically. You can also bring a model into memory without displaying it, as described in “Loading a Model” on page 1-7.

Opening an Existing Model

To open an existing model diagram, either:

- Click the **Open** button on the Library Browser’s toolbar (Windows operating systems only) or select **Open** from the Simulink library window’s **File** menu and then choose or enter the file name for the model to edit.
- Enter the name of the model (without the `.mdl` extension) in the MATLAB software Command Window. The model must be in the current directory 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, you must first use the later version to save the model in a format compatible with the earlier version. You can then open the model in the earlier version. See “Saving a Model in an Earlier Simulink Version” on page 1-11 for details.

Opening 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`, the Simulink software displays a warning or an error message, depending on whether it can or cannot encode the model, using the current character encoding, respectively. The warning or error message specifies the encoding of the current session and the encoding used to create the model. To avoid corrupting the model (see “Saving Models with Different Character Encodings” on page 1-10) and ensure correct display of the model’s text, you should:

- 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 safely edit and save the model.

Avoiding Initial Model Open Delay

You may notice that the first model that you open in a MATLAB technical computing environment session takes longer to open than do subsequent models. This is because to reduce its own startup time and to avoid unnecessary consumption of your system’s memory, the MATLAB software does not load the Simulink product into memory until the first time you open a Simulink model. You can cause the MATLAB technical computing environment to load the Simulink software when the MATLAB product starts up, and thus avoid the initial model opening delay. This can be done by using either the `-r` command line option or your MATLAB software `startup.m` file to run either `load_simulink` (loads the Simulink product) or `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 UNIX® systems, systems:

```
matlab -r load_simulink
```


Loading a Model

Loading a model brings it into memory but does not display it graphically. To both bring a model into memory and display it graphically, open the model as described in “Opening a Model” on page 1-4.

After you load a model (as distinct from opening it) you can work with the model programmatically just as you could if it were visible, but you cannot use any GUI techniques on it.

You cannot load a model using the Simulink GUI. To load a model programmatically, enter the Simulink command `load_system` in the MATLAB command window, specifying the model to be opened.

Saving a Model

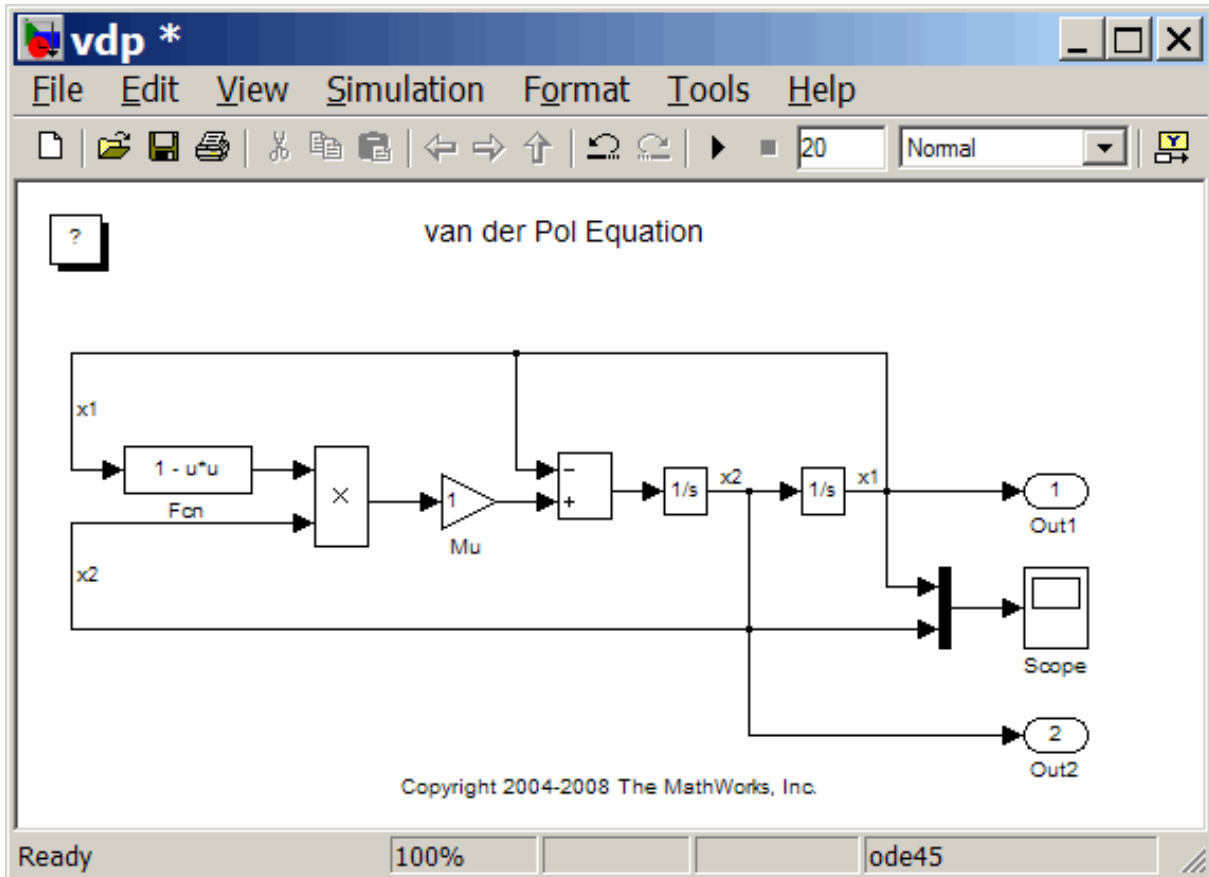
In this section...
“Introduction” on page 1-8
“How to Tell If a Model Needs Saving” on page 1-8
“Techniques for Saving Models” on page 1-9
“Saving Models with Different Character Encodings” on page 1-10
“Saving a Model in an Earlier Simulink Version” on page 1-11
“Saving from One Earlier Simulink Version to Another” on page 1-13

Introduction

Changes that you make to a model with the model editor or via MATLAB commands affect only the in-memory copy of your model. To make the changes permanent you must save the model, as described in this section.

How to Tell If a Model Needs Saving

To tell whether a model needs saving, look at the title bar of the model window. An asterisk appears next to the model’s name if the model needs saving as in the following example:



At the MATLAB command line and in M programs, you can use the model parameter, `Dirty`, to determine whether a model needs saving, for example,

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

Techniques for Saving Models

You can save a model by choosing either the **Save** or **Save As** command from the **File** menu. The model is saved by generating a specially formatted

file called the *model file* (with the .mdl extension) that contains the block diagram and block properties.

If you are saving a model for the first time, use the **Save** command to provide a name and location for the model file. Model file names must start with a letter and can contain letters, numbers, and underscores. The total number must not be greater than a certain maximum, usually 63 characters. You can use the MATLAB software `namelengthmax` command to find out if the maximum is greater than 63 characters for your system. The file name must not be the same as that of a MATLAB software command.

If you are saving a model whose model file was previously saved, use the **Save** command to replace the file's contents or the **Save As** command to save the model with a new name or location. You can also use the **Save As** command to save the model in a format compatible with previous releases of the Simulink product (see "Saving a Model in an Earlier Simulink Version" on page 1-11).

The Simulink software follows this procedure while saving a model:

- 1** If the mdl file for the model already exists, it is renamed as a temporary file.
- 2** All block `PreSaveFcn` callback routines are executed first, then the block diagram's `PreSaveFcn` callback routine are executed.
- 3** The model file is written to a new file using the same name and an extension of `mdl`.
- 4** All block `PostSaveFcn` callback routines are executed, then the block diagram's `PostSaveFcn` callback routine is executed.
- 5** The temporary file is deleted.

If an error occurs during this process, the Simulink software 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, and issues an error message. If an error occurs in step 2, step 3 is omitted and steps 4 and 5 are performed.

Saving Models with Different Character Encodings

When a model is saved, the character encoding in effect when the model was created (the original encoding) is used to encode the text stored in the model's

.mdl file, regardless of the character encoding in effect when the model is saved. This can lead to model corruption if you save a model whose original encoding differs from encoding currently in effect.

For example, it is possible that you introduced characters that cannot be represented in the model's original encoding. If this is the case, the model is saved as **model.err** where **model** is the model's name, leaving the original model file unchanged. The Simulink software 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 without losing all of the changes that you have made to the model in the current session, use the following procedure. First, use a text editor to find the character in the .err file at the position specified by the save error message. Then, find and delete the corresponding character in the open model and resave the model. 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.

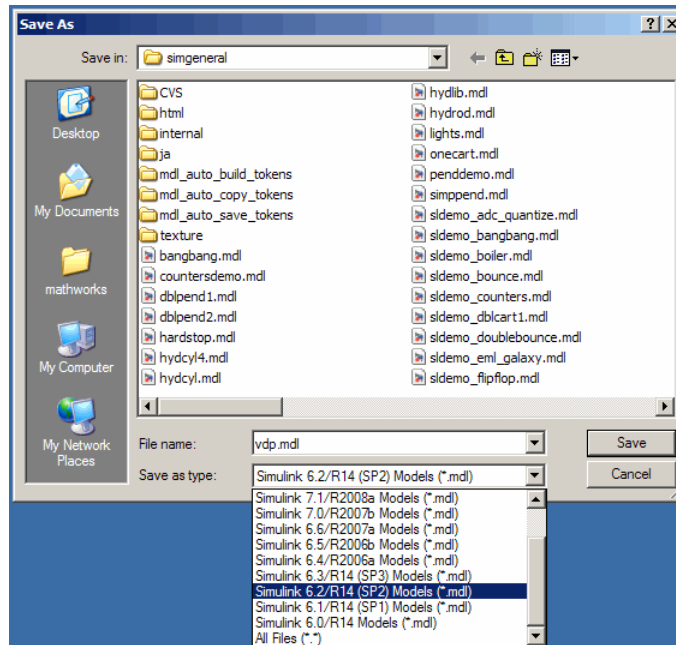
Saving a Model in an Earlier Simulink Version

The **Save As** command allows you to save a model created with the latest version of the Simulink software in formats used by earlier versions, including: Simulink 6 (Release 14, Release 14SP1, Release 14SP2, Release 14SP3, Release 2006a, Release 2006b, and Release 2007a), and Simulink® 7 (Release 2007b, Release 2008a, Release 2008b, Release 2009a, and Release 2009b). You might want to perform such a **Save As** if, for example, you need to make a model available to colleagues who only have access to one of these earlier versions of the Simulink product.

To save a model in an earlier format:

- 1 Select **Save** from the **File** menu. This saves a copy in the latest version of Simulink. This step is necessary to avoid compatibility problems.
- 2 Select **Save As** from the **File** menu.

The **Save As** dialog box is displayed.



- 3 Select a format from the **Save as type** list in the dialog box.
- 4 Click the **Save** button.

When saving a model in an earlier 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 run by the earlier version. In addition, Simulink converts blocks that postdate an earlier version into empty masked Subsystem blocks

colored yellow. For example, if you save a model that contains Polynomial blocks to Release R2007b, Simulink will convert the Polynomial blocks into empty masked Subsystem blocks and will color them yellow.

Saving from One Earlier Simulink Version to Another

You can open a model created in an earlier version of Simulink and use **Save as** to save the model in 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:

- 1** Use the current version of Simulink to open the model created with the earlier version.
- 2** Before making any changes, use **Save** to save the model in the current version.

After saving the model in the current version, you change and resave it as needed.

- 3** Use **Save as** to save the model in the earlier version of Simulink.
- 4** Start the earlier Simulink version and use it to open the saved model.
- 5** Use **Save** to save the model in the earlier version.

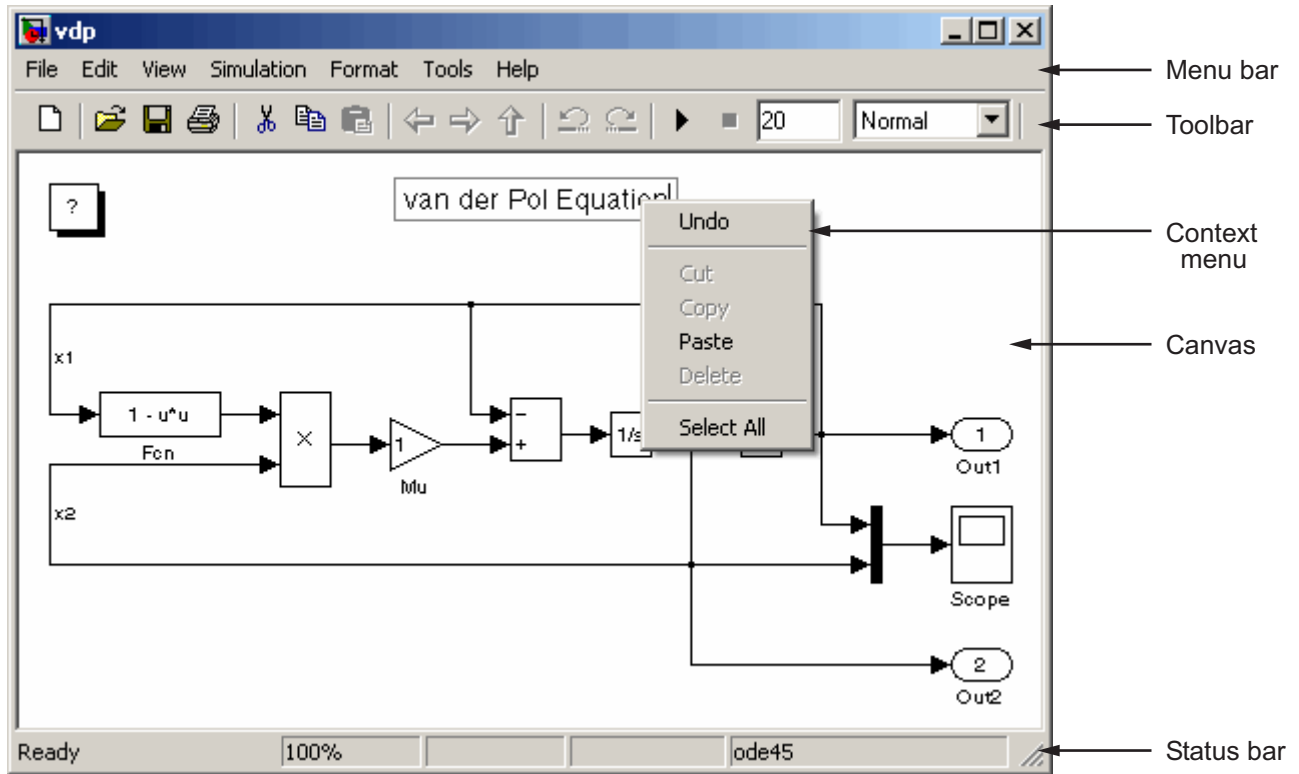
You can now use the model in the earlier version of Simulink exactly as you could if it had been created in that version.

Using the Model Editor

In this section...
“Editor Overview” on page 1-14
“Toolbar” on page 1-15
“Menu Bar” on page 1-19
“Canvas” on page 1-19
“Context Menus” on page 1-20
“Status Bar” on page 1-20

Editor Overview

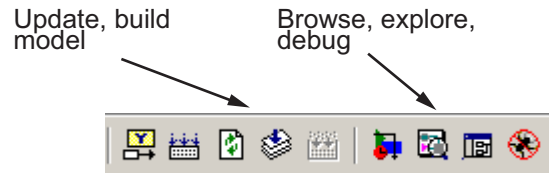
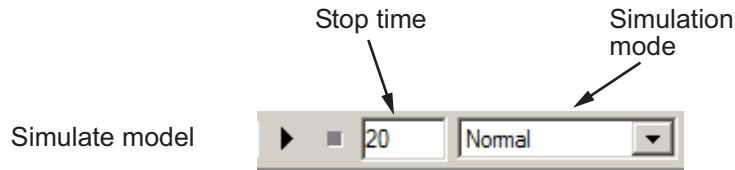
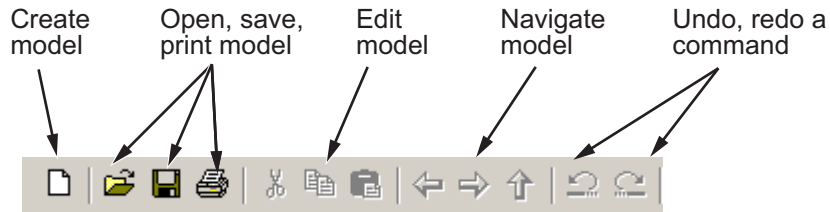
Opening a Simulink model or library displays the model or library in an instance of the Model Editor.



The Model Editor includes the following components.

Toolbar

The editor toolbar allow you to use your mouse to execute the Simulink software's most frequently used model editing, building, and simulation commands.
















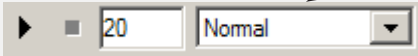





Using the Toolbar







Clicking a button executes the corresponding command. For example, to open a Simulink model, click the open folder icon on the toolbar. Letting the mouse cursor hover over a toolbar button or control causes a tooltip to appear. The tooltip describes the purpose of the button or control. You can hide the toolbar by clearing the **Toolbar** option on the editor **View** menu.

Button Reference

The following table summarizes the usage of each button on the editor toolbar. The buttons appear in the table in the same order as they appear left to right on the toolbar.

Button	Usage
	Create a new model (see “Creating an Empty Model” on page 4-2).
	Open a model (see “Opening a Model” on page 1-4).
	Save the current model (see “Saving a Model” on page 1-8).
	Print the current model (see “Printing a Block Diagram” on page 1-29).
	Cut the diagram object selected in the editor window to the system clipboard .
	Copy the selected diagram object to the system clipboard.
	Paste a diagram object from the system clipboard into the current diagram.
	Go back to the subsystem previously displayed in this editor window. This button works only in window reuse mode (see “Viewing Command History” on page 1-25).
	Go forward to the subsystem displayed after the subsystem currently displayed in this editor window. This button works only in window reuse mode.
	Go to the parent of the subsystem currently displayed in the editor window (see “Model Navigation Commands” on page 4-40).
	Undo the last editor command (see “Undoing a Command” on page 1-22).
	Redo the last undone command.

Button	Usage
	<p>Start simulating the model displayed in this editor window (see “Starting a Simulation” on page 21-3).</p> <hr/> <p>Note The controls to the right of this button enable you to set the end time of the simulation and the simulation mode, respectively.</p> <p>Enter simulation stop time here! Select simulation mode here!</p> <div style="text-align: center;">  </div>
	<p>Pause the current simulation (see “Pausing or Stopping a Simulation” on page 21-5).</p> <hr/> <p>Note This button replaces the Start button while a simulation is in progress.</p>
	<p>Stop the current simulation.</p>
	<p>Display block outputs as tool tips during simulation (see “Displaying Block Outputs” on page 7-41).</p>
	<p>Build the RTW target for this model.</p> <hr/> <p>Note This button is enabled only if the Real-Time Workshop is installed on your system (see “Initiating the Build Process” for more information)..</p>
	<p>Refresh Model blocks residing in this model (see “Refreshing Model Blocks” on page 6-77).</p>

Button	Usage
	Update this model's diagram (see "Updating a Block Diagram" on page 1-27).
	Build the selected subsystem. Note This button is enabled only if you have selected a subsystem and the Real-Time Workshop is installed on your system (see "Generating Code and Executables from Subsystems" for more information).
	Open the Library Browser (see "Library Browser").
	Open the Model Explorer (see "The Model Explorer" on page 19-2).
	Open the Model Browser (see "The Model Browser" on page 19-26).
	Open the Simulink Debugger (see Chapter 26, "Simulink Debugger").

Menu Bar

The Simulink menu bar contains commands for creating, editing, viewing, printing, and simulating models. The menu commands apply to the model displayed in the editor. See "Creating a Model" and "Running Simulations" for more information.

Canvas

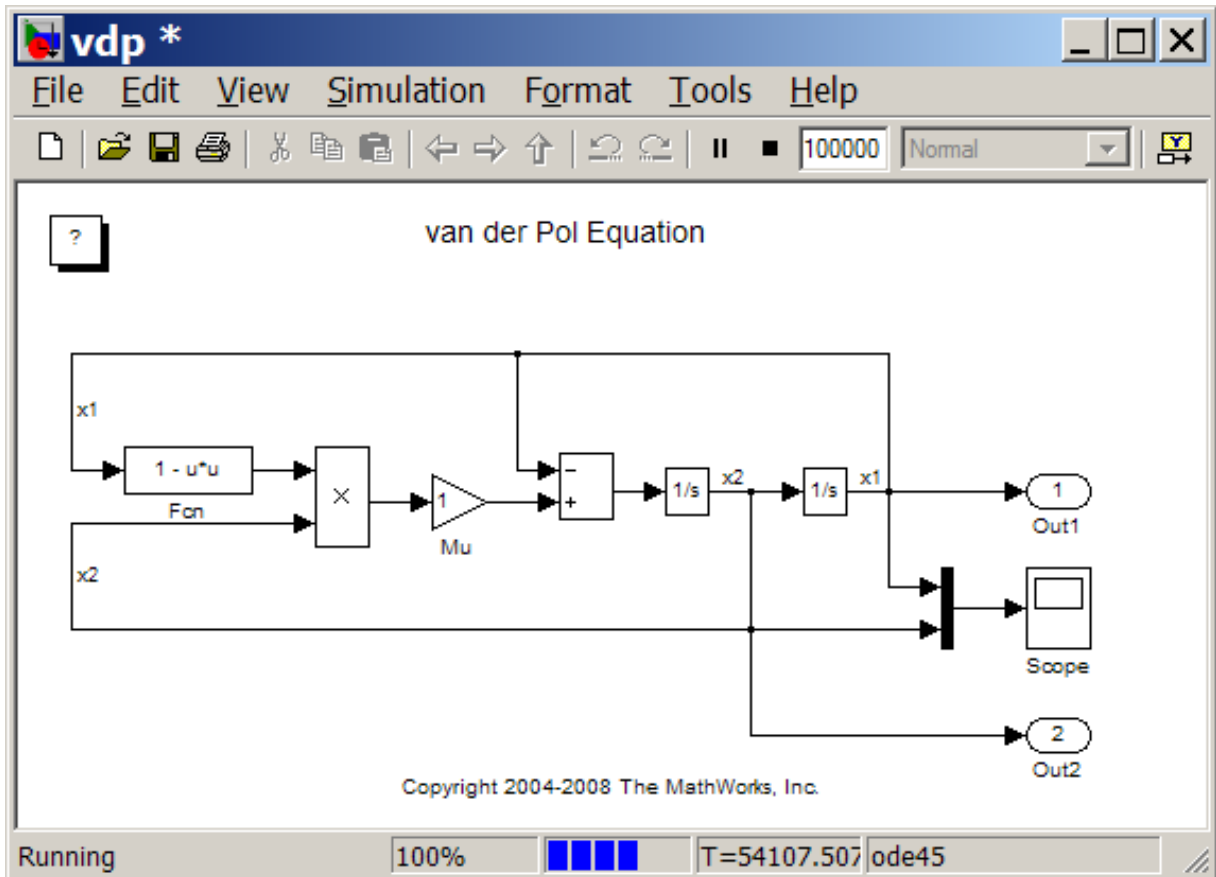
The canvas displays the model's block diagram. The canvas allows you to edit the block diagram. You can use your system's mouse and keyboard to create and connect blocks, select and move blocks, edit block labels, display block dialog boxes, and so on. See "Working with Blocks" for more information.

Context Menu

A context-sensitive menu is displayed when you click the right mouse button over the canvas. The contents of the menu depend on whether a block, line, annotation, or other object is selected. If an object is selected, the menu displays commands that apply only to the selected object. If no object is selected, the menu displays commands that apply to a model or library as a whole.

Status Bar

The status bar appears only in the Windows operating system version of the Model Editor. It displays the update and simulation status of the model.



Status Messages

Zoom Level

Progress Bar

Current Simulation Time

Solver

The status bar includes the following areas:

- status message

Displays the status of a simulation (e.g., ready to run or running) or of a model update.

- zoom level

Displays the zoom level of the model window as a percentage of normal.

- progress bar

Indicates how far a model update or simulation has progressed. The color of the status bar indicates whether Simulink is updating the model (green) or simulating the model (blue).

- current simulation time
- current solver

You can display or hide the status bar by selecting or clearing the **Status Bar** option on the **View** menu.

Undoing a Command

You can cancel the effects of up to 101 consecutive operations by choosing **Undo** from the **Edit** menu. You can undo these operations:

- 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” for more information)

You can reverse the effects of an **Undo** command by choosing **Redo** from the **Edit** menu.

Zooming Block Diagrams

You can enlarge or shrink the view of the block diagram in the current Simulink software window. To zoom a view:

- Select **Zoom In** from the **View** menu (or type **r**) to enlarge the view.
- Select **Zoom Out** from the **View** menu (or type **v**) to shrink the view.
- Select **Fit System To View** from the **View** menu (or press the space bar) to fit the diagram to the view.
- Select **Normal** from the **View** menu (or type **1**) to view the diagram at actual size.

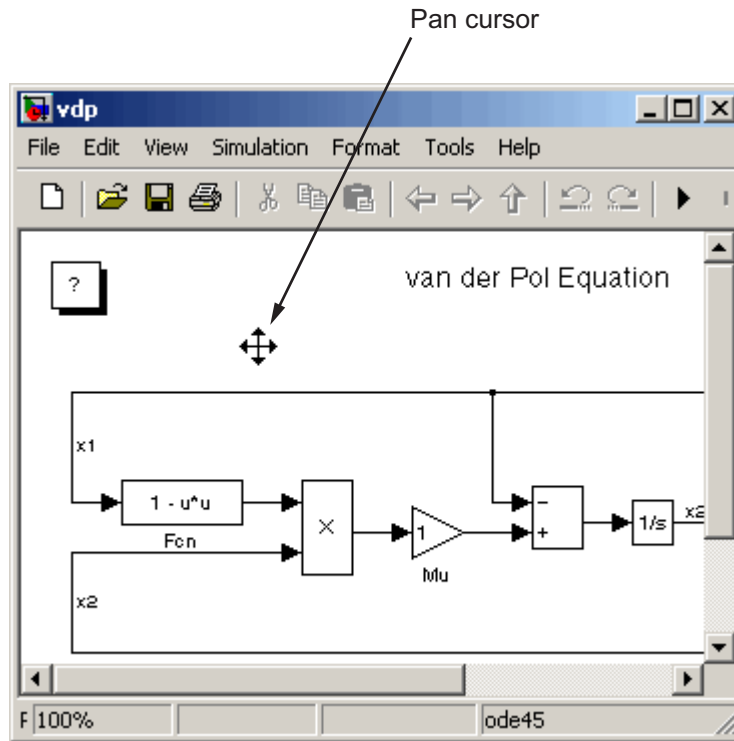
By default, the Simulink software fits a block diagram to view when you open the diagram either in the model browser's content pane or in a separate window. If you change a diagram's zoom setting and save the model containing the diagram, the model editor restores the setting the next time you open the diagram. If you want to restore the default behavior, choose **Fit System To View** from the **View** menu the next time you open the diagram.

Panning Block Diagrams

You can use your keyboard alone (see “Model Viewing Shortcuts” on page 1-43) or in combination with your mouse to pan model diagrams that are too large to fit in the Model Editor's window. To use the keyboard and the mouse, position the mouse over the diagram, hold down the **p** or **q** key on the keyboard, then hold down the left mouse button.

Note You must press and hold down the key first and then the mouse button. The reverse does not work.

A pan cursor appears.



Moving the mouse now pans the model diagram in the editor window.

Viewing Command History

A history of the modeling viewing commands is maintained (such as pan and zoom) that you execute for each model window. The history allows you to quickly return to a previous view in a window, using the following commands, accessible from the Model Editor's **View** menu and tool bar:

- **Back** (←) — Displays the previous view in the view history.
- **Forward** (→) — Displays the next view in the view history.
- **Go To Parent** (↑) — Opens, if necessary, the parent of the current subsystem and brings its window to the top of the desktop.

Note A separate view history is maintained for each model window opened in the current session. As a result, the **View > Back** and **View > Forward** commands cannot cross window boundaries. For example, if window reuse is not on and you open a subsystem in another window, you cannot use the **View > Back** command to go to the window displaying the parent system. You must use the **View > Go To Parent** command in this case. On the other hand, if you enable window reuse and open a subsystem in the current window, you can use **View > Back** to restore the parent view.

Bringing the MATLAB Software Desktop Forward

The Simulink product opens model windows on top of the MATLAB desktop. To bring the MATLAB desktop back to the top of your screen, select **View > MATLAB Desktop** from the Model Editor's menu bar.

Copying Models to Third-Party Applications

On a computer running the Microsoft Windows operating system, you can copy a Simulink product model to the Windows operating system clipboard, then paste it to a third-party application such as word processing software. The Simulink product allows you to 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. See “Exporting to the Windows or Macintosh® Clipboard” for a description of how to set up the figure settings and save a figure to the clipboard.

The following steps give an example of how use the MATLAB software to copy a model to a third-party application:

- 1** Set the figure copying options.
 - a** Select **File > Preferences**. The Preferences dialog box appears.
 - b** Under the **Figure Copy Template** node, select **Copy Options**.
 - c** In the Clipboard format pane on the right, select **Preserve information (metafile if possible)**.

With this setting, the MATLAB software selects the figure format for you, and uses the metafile format whenever possible.
 - d** Click **OK**.
- 2** Click **OK**.
- 3** Open the vdp model.
- 4** In the Model Editor, select **Edit > Copy Model to Clipboard**.
- 5** Open a document in Microsoft Word and paste the contents of the clipboard.

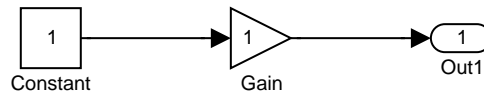
Updating a Block Diagram

You can leave many attributes of a block diagram, such as signal data types and sample times, unspecified. The Simulink product then infers the values of block diagram attributes based on block connectivity and attributes that you do specify, a process known as *updating the diagram*. The Simulink software tries to infer the most appropriate value for an attribute that you do not specify. If an attribute cannot be inferred, it halts the update and displays an error dialog box.

A model's block diagram is updated at the start of every simulation of a model. This assures that the simulation reflects the latest changes that you have made to a model. In addition, you can command the Simulink software to update a diagram at any time by selecting **Edit > Update Diagram** from the Model Editor's menu bar or context menu, or by pressing **Ctrl+D**. This allows you to determine the values of block diagram attributes inferred by the Simulink software immediately after opening or editing a model.

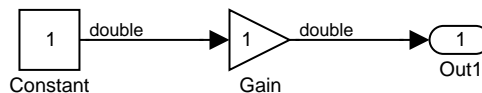
For example:

- 1 Create the following model.



- 2 Select **Format > Port/Signal Displays > Port Data Types** from the Model Editor's menu bar.

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.

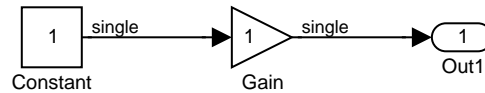


- 3 Set the **Output data type** parameter of the Constant block (see Constant) to `single`.

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

- 4 Select **Edit > Update Diagram** from the Model Editor's menu bar or press **Ctrl-D**.

The block diagram is updated to reflect the change that you made previously.



Note that the Simulink software has inferred a data type for the output of the Gain block. This is because you did not specify a data type for the block. The 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's input is `single`.

Printing a Block Diagram

In this section...

- “About Printing” on page 1-29
- “Print Dialog Box” on page 1-29
- “Specifying Paper Size and Orientation” on page 1-31
- “Positioning and Sizing a Diagram” on page 1-31
- “Tiled Printing” on page 1-32
- “Print Sample Time Legend” on page 1-36
- “Print Command” on page 1-36

About Printing

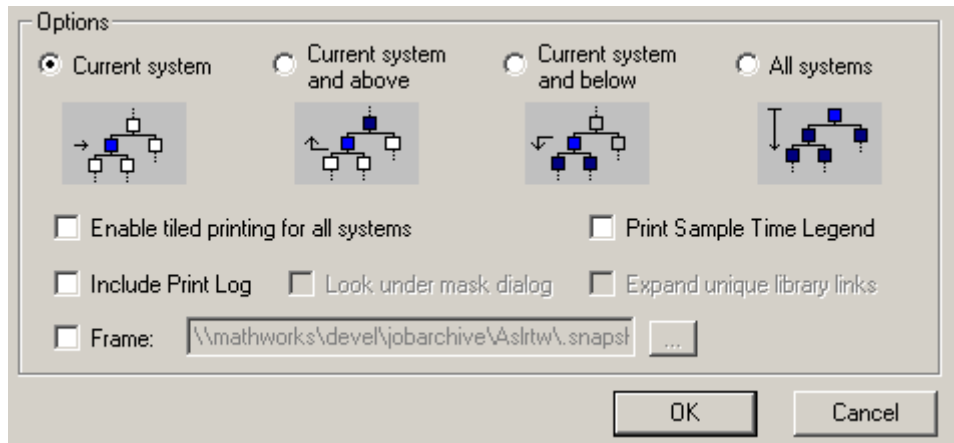
You can print a block diagram by selecting **Print** from the **File** menu or by using the print command in the MATLAB software Command Window.

Print Dialog Box

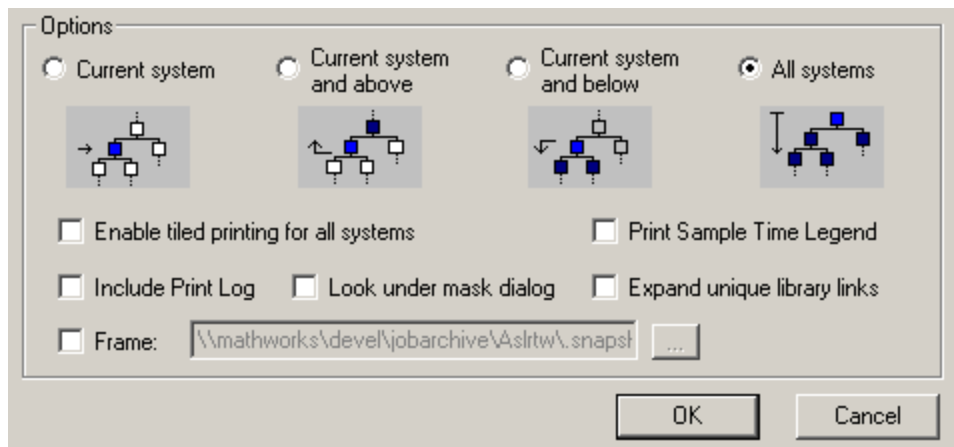
When you select the **Print** menu item, the **Print** dialog box appears. The **Print** dialog box enables you to selectively print systems within your model. Using the dialog box, you can print

- The current system only
- The current system and all systems above it in the model hierarchy
- The current system and all systems below it in the model hierarchy, with the option of looking into the contents of masked and library blocks
- All systems in the model, with the option of looking into the contents of masked and library blocks
- The entire diagram over multiple pages
- An overlay frame on each diagram

The portion of the **Print** dialog box that supports selective printing is similar on supported platforms. This figure shows how it looks on a computer running the Microsoft Windows operating system. In this figure, only the current system is to be printed.



When you select either the **Current system and below** or **All systems** option, two check boxes become enabled. In this figure, **All systems** is selected.



Selecting the **Look under mask dialog** check box prints the contents of masked subsystems when encountered at or below the level of the current block. When you are printing all systems, the top-level system is considered the current block, so the Simulink software looks under any masked blocks encountered.

Selecting the **Expand unique library links** check box prints the contents of library blocks when those blocks are systems. Only one copy is printed regardless of how many copies of the block are contained in the model. For more information about libraries, see Chapter 8, “Working with Block Libraries”.

The print log lists the blocks and systems printed. To print the print log, select the **Include Print Log** check box.

Selecting the **Enable tiled printing for all systems** check box overrides the tiled-print settings for individual subsystems in a model. See “Tiled Printing” on page 1-32 for more information.

Selecting the **Frame** check box prints a title block frame on each diagram. Enter the path to the title block frame in the adjacent edit box. You can create a customized title block frame, using the MATLAB product frame editor. See `frameedit` for information on using the frame editor to create title block frames.

Specifying Paper Size and Orientation

You can specify the type and orientation of the paper used to print a model diagram. You can do this on all platforms by setting the model’s `PaperType` and `PaperOrientation` properties, respectively (see “Model and Block Parameters” in the online reference), using the `set_param` command. You can set the paper orientation alone, using the MATLAB software `orient` command. On computers running the Windows, operating system, the **Print** and **Printer Setup** dialog boxes let you set the page type and orientation properties as well.

Positioning and Sizing a Diagram

You can use a model’s `PaperPositionMode` and `PaperPosition` parameters to position and size the model’s diagram on the printed page. 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 page's bottom-left corner. The last two elements specify the width and height of the rectangle. When the model's `PaperPositionMode` is manual, the Simulink software positions (and scales, if necessary) the model's diagram to fit inside the specified print rectangle. For example, the following commands

```
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
```

print the block diagram of the `vdp` sample model in the lower-left corner of a U.S. letter-size page in landscape orientation.

If `PaperPositionMode` is auto, the Simulink software centers the model diagram on the printed page, scaling the diagram, if necessary, to fit the page.

Tiled Printing

By default, each block diagram is scaled during the printing process such that it fits on a single page. That is, the size of a small diagram is increased or the size of a large diagram is decreased to confine its printed image to one page. In the case of a large diagram, scaling can make the printed image difficult to read.

By contrast, 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. You can control the number of pages over which the Simulink software prints the block diagram, and hence, the total size of the printed diagram.

Moreover, different tiled-print settings are accommodated for each of the systems in your model. Consequently, you can customize the appearance of all printed images to best suit your needs. The following sections describe how to utilize tiled printing.

Enabling Tiled Printing

To enable tiled printing for a particular system in your model, select the **Enable Tiled Printing** item from the **File** menu associated with that system's Model Editor.

Or you can enable tiled printing programmatically using the `set_param` command. Simply set the system's `PaperPositionMode` parameter to `tiled` (see "Model Parameters" in the online Simulink reference). For example, the following commands

```
sldemo_f14
set_param('sldemo_f14/Controller', 'PaperPositionMode', 'tiled')
```

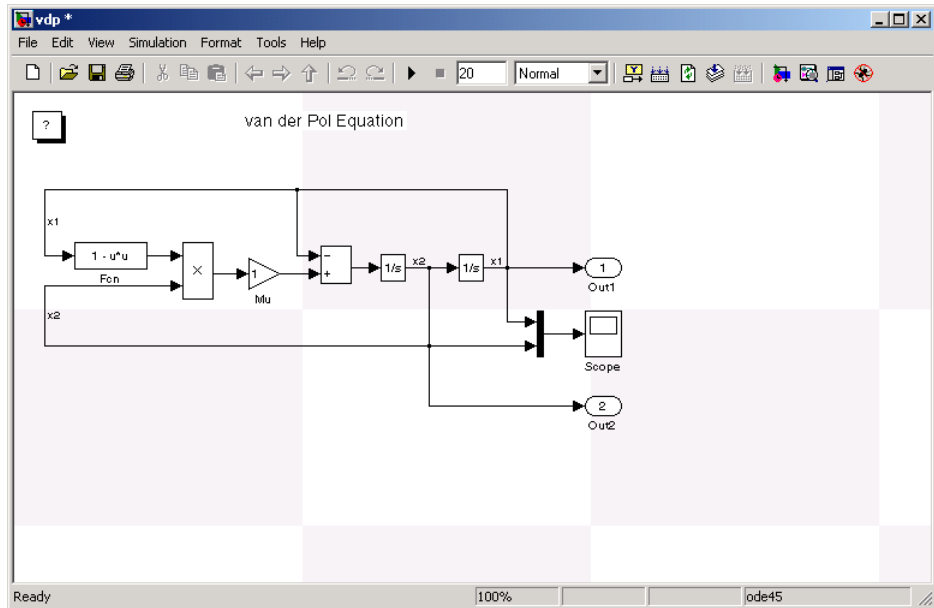
open the `f14` demo model and enable tiled printing for the `Controller` subsystem.

To enable tiled printing for all systems in your model, select the **Enable tiled printing for all systems** check box on the **Print** dialog box (see "Print Dialog Box" on page 1-29). If you select this option, the Simulink software overrides the individual tiled-print settings for all systems in your model.

Displaying Page Boundaries

You can display the page boundaries in the Model Editor to visualize the model's size and layout with respect to the page. To make the page boundaries visible for a particular system in your model, select the **Show Page Boundaries** item from the **View** menu associated with that system's Model Editor. Or you can display the page boundaries programmatically using the `set_param` command. Simply set the system's `ShowPageBoundaries` parameter to `on` (see "Model Parameters" in the online Simulink reference).

The Simulink software renders the page boundaries on the Model Editor's canvas. If tiled printing is enabled, page boundaries are represented by a checkerboard pattern. As illustrated in the following figure, each checkerboard square indicates the extent of a separate page.



If tiled printing is disabled, only a single page is displayed on the Model Editor's canvas.

Specifying Tiled Print Settings

You can use a system's `TiledPageScale` and `TiledPaperMargins` parameters to customize certain aspects of tiled printing. You specify values for these parameters using the `set_param` command.

The `TiledPageScale` parameter scales the block diagram so that more or less of it appears on a single tiled page. By default, its value is 1. Values greater than 1 proportionally scale the diagram such that it occupies a smaller percentage of the tiled page, while values between 0 and 1 proportionally scale the diagram such that it occupies 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` of 2 makes the printed diagram appear half its size on a tiled page.

You can specify the margin sizes associated with tiled pages using the `TiledPaperMargins` parameter. The value of `TiledPaperMargins` is a

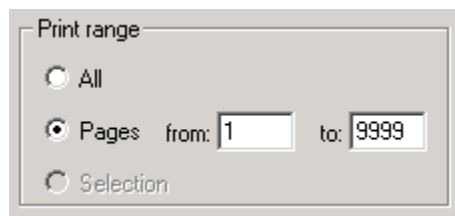
vector of form [left top right bottom]. Each element specifies the size of the margin at a particular edge of the page. The value of the `PaperUnits` parameter is used to determine its units of measurement. Each margin is 0.5 inches by default. By decreasing the margin sizes, you can increase the printable area of the tiled pages.

Printing Tiled Pages

By default, all of a system's tiled pages are printed when you select **Print** from the **File** menu or use the print command at the MATLAB software prompt.

Alternatively, you can specify the range of tiled page numbers that are printed, as follows:

- On a computer running the Microsoft Windows operating system, you can specify a range of tiled page numbers to be printed using the **Print range** portion of the **Print** dialog box. This field is accessible if you select both the **Current system** and **Enable tiled printing for all systems** options (see “Print Dialog Box” on page 1-29).



- On all platforms, you can specify a range of tiled page numbers to be printed using the print command at the MATLAB software prompt. The print command's `tileall` option enables tiled printing for the system, and its `pages` option indicates the range of tiled page numbers to be printed (see “Print Command” on page 1-36). For example, the following commands

```
vdp
set_param('vdp', 'PaperPositionMode', 'tiled')
set_param('vdp', 'ShowPageBoundaries', 'on')
set_param('vdp', 'TiledPageScale', '0.1')
```

open the `vdp` demo model, enable tiled printing, display the page boundaries, and scale the tiled pages such that the block diagram spans

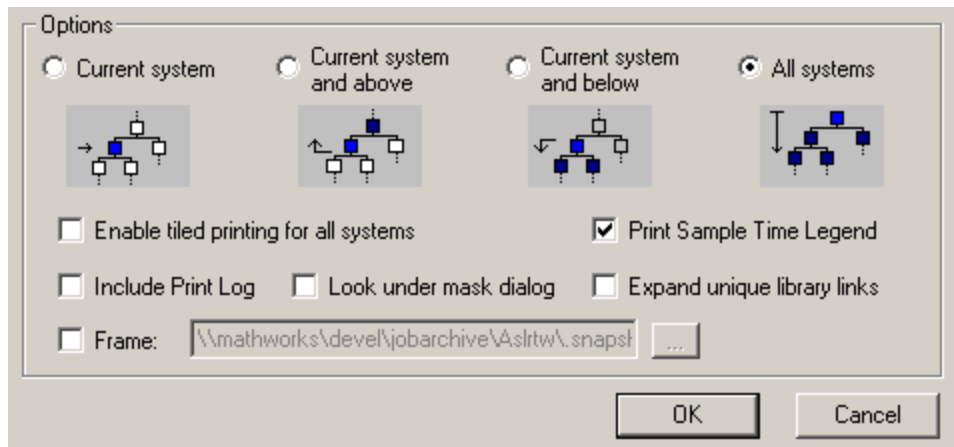
multiple pages. You can print the second, third, and fourth pages by issuing the following command at the MATLAB software prompt:

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

Note The Simulink software 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, etc.

Print Sample Time Legend

If you select the **Print Sample Time Legend** option, the Sample Time Legend will print on a separate page from your model. The legend contains sample time information for your entire system, including any subsystems.



For more information about the contents and the settings of the legend, see “How to View Sample Time Information” on page 3-9.

Print Command

The format of the print command is

```
print -ssys -device -tileall -pagesp filename
```

`sys` is the name of the system to be printed. The system name must be preceded by the `s` switch identifier and is the only required argument. `sys` must be open or must have been open during the current session. If the system name contains spaces or takes more than one line, you need to specify the name as a string. See the examples below.

`device` specifies a device type. For a list and description of device types, see the documentation for the MATLAB software `print` function.

`tileall` specifies the tiled printing option (see “Tiled Printing” on page 1-32).

`p` is a two-element vector specifying the range of tiled page numbers to be printed. The vector must be preceded by the `pages` switch identifier. This option is valid only when you enable tiled printing using the `tileall` switch. For an example of its usage, see “Printing Tiled Pages” on page 1-35.

`filename` is the PostScript® file to which the output is saved. If `filename` exists, it is replaced. If `filename` does not include an extension, an appropriate one is appended.

For example, this command prints a system named `untitled`.

```
print -suntitled
```

This command prints the contents of a subsystem named `Sub1` in the current system.

```
print -sSub1
```

This command prints the contents of a subsystem named `Requisite Friction`.

```
print (['-sRequisite Friction'])
```

The next example prints a system named `Friction Model`, a subsystem whose name appears on two lines. The first command assigns the newline character to a variable; the second prints the system.

```
cr = sprintf('\n');  
print (['-sFriction' cr 'Model'])
```

To print the currently selected subsystem, enter

```
print(['-s', gcb])
```

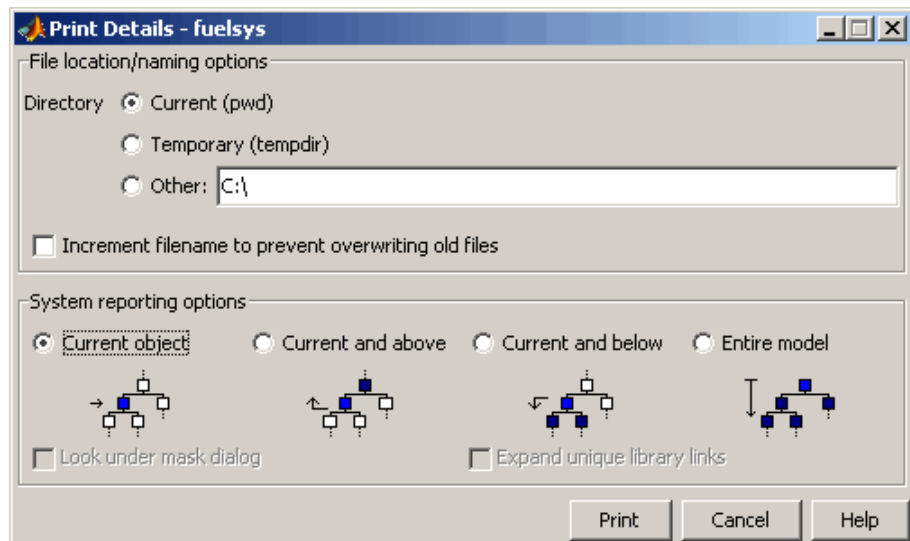
Generating a Model Report

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

To generate a report for the current model:

- 1 Select **Print Details** from the model's **File** menu.

The **Print Details** dialog box appears.

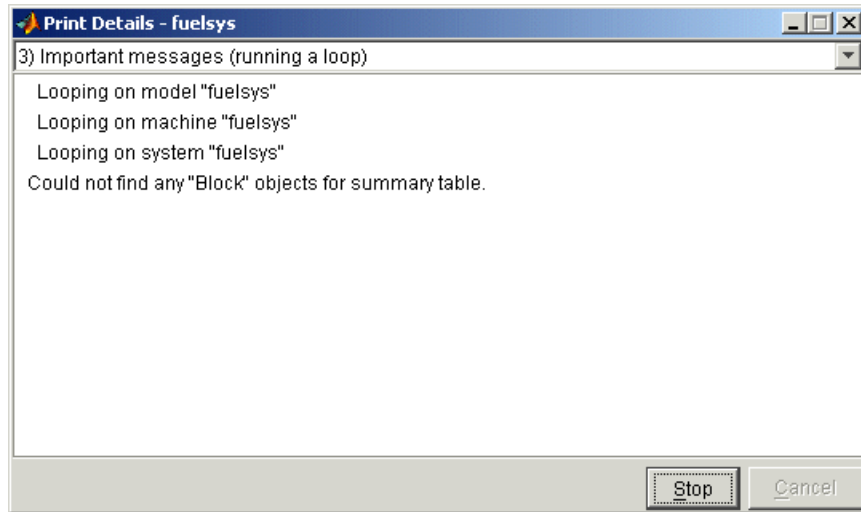


The dialog box allows you to select various report options (see “Model Report Options” on page 1-40).

- 2 Select the desired report options on the dialog box.
- 3 Select **Print**.

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

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



You can 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 on the **Print Details** dialog box changes to a **Stop** button. Clicking this button terminates the report generation. 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

The **Print Details** dialog box allows you to select the following report options.

Directory

The directory where the HTML report is stored. The options include your system's temporary directory (the default), your system's current directory, or another directory 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.

Ending a Simulink Session

Terminate a Simulink software session by closing all Simulink windows.

Terminate a MATLAB software session by choosing one of these commands from the **File** menu:

- On a computer running the Microsoft Windows operating system: **Exit MATLAB**
- On a UNIX system, system: **Quit MATLAB**

Summary of Mouse and Keyboard Actions

In this section...
“Model Viewing Shortcuts” on page 1-43
“Block Editing Shortcuts” on page 1-44
“Line Editing Shortcuts” on page 1-45
“Signal Label Editing Shortcuts” on page 1-45
“Annotation Editing Shortcuts” on page 1-46

Model Viewing Shortcuts

The following table lists keyboard shortcuts for viewing models.

Task	Microsoft Windows Operating System	UNIX System
Zoom in	r	r
Zoom out	v	v
Zoom to normal (100%)	l	l
Pan left	d or Ctrl+Left Arrow	d or Ctrl+Left Arrow
Pan right	g or Ctrl+Right Arrow	g or Ctrl+Right Arrow
Pan up	e or Ctrl+Up Arrow	e or Ctrl+Up Arrow
Pan down	c or Ctrl+Down Arrow	c or Ctrl+Down Arrow
Fit selection to screen	f	f
Fit diagram to screen	Space	Space
Pan with mouse	Hold down p or q and drag mouse	Hold down p or q and drag mouse
Go back in pan/zoom history	b or Shift+Left Arrow	b or Shift+Left Arrow

Task	Microsoft Windows Operating System	UNIX System
Go forward in pan/zoom history	t or Shift+Right Arrow	t or Shift+Right Arrow
Delete selection	Delete or Back Space	Delete or Back Space
Move selection	Use arrow keys	Use arrow keys

Block Editing Shortcuts

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

Task	Microsoft Windows Operating System	UNIX System
Select one block	LMB	LMB
Select multiple blocks	Shift + LMB	Shift + LMB; or CMB alone
Copy block from another window	Drag block	Drag block
Move block	Drag block	Drag block
Duplicate block	Ctrl + LMB and drag; or RMB and drag	Ctrl + LMB and drag; or RMB and drag
Connect blocks	LMB	LMB
Disconnect block	Shift + drag block	Shift + drag block; or CMB and drag
Open selected subsystem	Enter	Return
Go to parent of selected subsystem	Esc	Esc

Line Editing Shortcuts

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

Task	Microsoft Windows Operating System	UNIX System
Select one line	LMB	LMB
Select multiple lines	Shift + LMB	Shift + LMB; or CMB alone
Draw branch line	Ctrl + drag line; or RMB and drag line	Ctrl + drag line; or RMB + drag line
Route lines around blocks	Shift + draw line segments	Shift + draw line segments; or CMB and draw segments
Move line segment	Drag segment	Drag segment
Move vertex	Drag vertex	Drag vertex
Create line segments	Shift + drag line	Shift + drag line; or CMB + drag line

Signal Label Editing Shortcuts

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

Action	Microsoft Windows Operating System	UNIX System
Create signal label	Double-click line, then enter label	Double-click line, then enter label
Copy signal label	Ctrl + drag label	Ctrl + drag label
Move signal label	Drag label	Drag label
Edit signal label	Click in label, then edit	Click in label, then edit
Delete signal label	Shift + click label, then press Delete	Shift + click label, then press Delete

Annotation Editing Shortcuts

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

Action	Microsoft Windows Operating System	UNIX System
Create annotation	Double-click in diagram, then enter text	Double-click in diagram, then enter text
Copy annotation	Ctrl + drag label	Ctrl + drag label
Move annotation	Drag label	Drag label
Edit annotation	Click in text, then edit	Click in text, then edit
Delete annotation	Shift + select annotation, then press Delete	Shift + select annotation, then press Delete

How Simulink Works

- “Introduction” on page 2-2
- “Modeling Dynamic Systems” on page 2-3
- “Simulating Dynamic Systems” on page 2-19

Introduction

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.

Modeling Dynamic Systems

In this section...

“Block Diagram Semantics” on page 2-3

“Creating Models” on page 2-4

“Time” on page 2-5

“States” on page 2-5

“Block Parameters” on page 2-9

“Tunable Parameters” on page 2-9

“Block Sample Times” on page 2-10

“Custom Blocks” on page 2-11

“Systems and Subsystems” on page 2-12

“Signals” on page 2-16

“Block Methods” on page 2-17

“Model Methods” on page 2-18

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 Chapter 4, “Creating a Model”) selected from libraries of block types (see Blocks — Alphabetical List) 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.

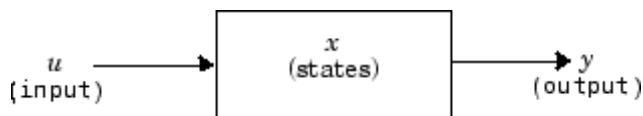
States

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 Chapter 26, "Simulink Debugger").
- The **Data Import/Export** pane of a model's Configuration Parameters dialog box (see Chapter 15, "Importing and Exporting Data") 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 demo 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,

the Runge-Kutta variable-step methods, `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, The MathWorks does not recommend 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.

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 2-19 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.

Note You can not change the values of source block parameters through either a dialog box or the Model Explorer while a simulation is running. Opening the dialog box of a source block with tunable parameters causes a running simulation to pause. While the simulation is paused, you can edit the parameter values displayed on the dialog box. However, you must close the dialog box to have the changes take effect and allow the simulation to continue.

It should be pointed out that parameter changes do not immediately occur, but are queued up and then applied at the start of the next time step during model execution. Returning to our example of the constant block, the function it defines is $\text{signal}(t) = \text{ConstantValue}$ for all time. If we were to allow the constant value to be changed immediately, then the solution at the point in time at which the change occurred would be invalid. Thus we must queue the change for processing at the next time step.

You can use the **Inline parameters** option on the **Optimization** pane of the **Configuration Parameters** dialog box to specify that all parameters in your model are nontunable except for those that you specify. This can speed up execution of large models and enable generation of faster code from your model. See “Configuration Parameters Dialog Box” 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 “How to View Sample Time Information” on page 3-9). 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 “How to View Sample Time Information” on page 3-9).

For a more detailed discussion of sample times, see Chapter 3, “Working with Sample Times”

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 an M-file or a MEX-file that contains the block’s system functions (see *Writing S-Functions*). 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 Chapter 29, “Creating Custom Blocks” 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 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 transition occurs on a triggering, function-call, action, or enabling input (see Chapter 5, “Creating Conditional Subsystems”). The blocks within a nonvirtual subsystem execute only when all subsystem inputs are valid. All nonvirtual subsystems are drawn with a bold border. 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's enable port is greater than zero. You can also configure an enabled subsystem to hold or reset the states of blocks within the enabled subsystem via the `States when enabling` parameter in the enable port block. Each output port of an enabled subsystem can be configured to hold or reset its output via the `Output when disabled` parameter in the output block. You can create an enabled subsystem by placing an enable port block within a subsystem.

Triggered subsystems. A triggered 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. You create a triggered subsystem by placing a trigger port block within a subsystem. A Stateflow chart can also have a trigger port which is defined by using the Stateflow editor. From Simulink's perspective there is no difference 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 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 Simulink's Ports & Subsystems library into your model and connect a function-call initiator to the function-call port displayed on top 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's 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 non-inherited sample time or inherited (-1) sample time. All blocks that specify a non-inherited sample time must specify the same sample time, i.e., 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 with trigger subsystems. An enabled with trigger 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. Simulink limits the types of blocks placed in an enabled with 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. You can create an enabled with triggered subsystem by placing a trigger port block and a enable port block within a subsystem.

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 have one 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 SwitchCase 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. The subsystem icon will automatically adapt to the type of block (i.e., If or SwitchCase 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 on 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 when compared with function-call subsystems. This restriction also means that you can have control over how the states and outputs behave.

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

A while-subsystem is very similar to a function-call subsystem in that it can run for any number of iterations on a given time step. The while-subsystem differs from a function-call subsystem in that there is no separate initiator (e.g., a Stateflow Chart). In addition, a while-subsystem has access to the current iteration number optionally produced by the while-iterator block. A while-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-subsystems. A for-subsystem will run a fixed number of iterations on each model time step. The number of iterations can be an external input to the for-subsystem or specified internally on the for-iterator block. A for-subsystem is created by placing a for-iterator block within a subsystem block.

A for-subsystem has access to the current iteration number that is optionally produced by the for-iterator block. A for-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-subsystem is very similar to a while-subsystem with the restriction that the number of iterations on any given time step is fixed.

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 Chapter 10, "Working with 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 2-19

“Link Phase” on page 2-20

“Simulation Loop Phase” on page 2-20

“Solvers” on page 2-22

“Zero-Crossing Detection” on page 2-24

“Algebraic Loops” on page 2-35

Model Compilation

The first phase of simulation occurs when you choose **Start** from the Model Editor’s **Simulation** menu, with the system’s model open. This 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 2-22).
- Determines the block sorted order (see “Controlling and Displaying the Sorted Order” on page 7-46 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 3-31).

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 “Assigning Block Priorities” on page 7-52). 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 2-22) used to compute the system’s continuous states, the system’s fundamental sample time (see “Managing Sample Times in Systems” on page 3-22), and whether the system’s continuous states have discontinuities (see “Zero-Crossing Detection” on page 2-24).

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 2-22).

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 2-24 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 21-9).

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 21-9).

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 “Managing Sample Times in Systems” on page 3-22 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 more accurate integration results can be obtained by including the derivative input information at each time step. This is done by activating the **Shapes Preservation** option in the Solver pane of the Configuration Parameter dialog.

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 2-29.

Demonstrating Effects of Excessive Zero-Crossing Detection

The Simulink software comes with two demos that illustrate zero-crossing behavior: `sldemo_bounce.mdl` and `sldemo_doublebounce.mdl`.

- Run the bounce demo to see how excessive zero crossings can cause a simulation to halt before the intended completion time.
- Run the double-bounce demo to see how the adaptive algorithm successfully solves a complex system with two distinct zero-crossing requirements.

The Bounce Demo.

- 1** Load the demo by typing `sldemo_bounce` at the MATLAB command prompt.
- 2** Once the block diagram appears, navigate to the Configuration Parameters dialog box. Confirm that the **Algorithm** is set to Nonadaptive.
- 3** Run the model for a simulation time of 20 seconds.
- 4** After the simulation completes, click on the scope to display the results.

You may need to click on **Autoscale** to get a clear display.

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.

- 5** Change the simulation run time to 25 seconds, and run the simulation again.

- 6 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 this limit can be increased by adjusting the **Number of consecutive zero crossings parameter** in the Configuration Parameters dialog box, doing so in this case does not allow the simulation to simulate for 25 seconds.

- 7 Navigate to the Configuration Parameters dialog box and select the **Adaptive Algorithm** from the **Algorithm** pull down.
- 8 Change the simulation time to 25 seconds, and run the simulation again.
- 9 This time the simulation runs to completion because the adaptive algorithm prevented an excessive number of zero crossings from occurring.

The Double-bounce Demo.

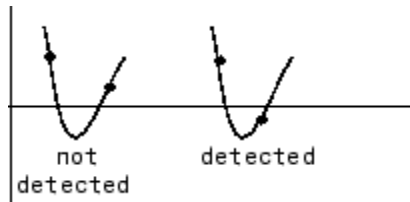
- 1 Load the demo by typing `sldemo_doublebounce` at the MATLAB command prompt.
- 2 In the demo, click the **Nonadaptive** button. This causes the demo to run with the Nonadaptive Algorithm. This is the default setting used by the Simulink software for all models.
- 3 Notice that the two balls hit the ground and recoil at different times.
- 4 The simulation halts after 14 seconds because the ball on the left has exceeded the number of zero crossings limit. The ball on the right is left hanging in mid air.
- 5 Click on the error message to clear 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 when the ground shifts out from underneath the ball on the left after 20 seconds.

How the Simulator Can Miss Zero-Crossing Events

The bounce and double-bounce demos 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...	Rational 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.

Make this change...	How to make this change...	Rational for making this change...
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 .
Disable zero-crossing detection for a specific block	<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. 	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.

Make this change...	How to make this change...	Rational for making this change...
Disable zero-crossing detection for the entire model	Select Disable all from the Zero-crossing control pull down on the Solver pane of the Configuration Parameters dialog box.	This prevents zero crossings from being detected anywhere in your model. A consequence is that your model no longer benefits from the increased accuracy that zero-crossing detection provides.
If using the ode15s solver, consider adjusting the order of the numerical differentiation formulas	Select a value from the Maximum order pull down on the Solver pane of the Configuration Parameters dialog box.	For more information, see “Maximum order”.
Reduce the maximum step size	Enter a value for the Max step size option on the Solver pane of the Configuration Parameters dialog box.	This can insure the solver takes steps small enough to resolve the zero crossing. However, reducing the step size can increase simulation time, and is seldom necessary when using the Adaptive algorithm.

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.

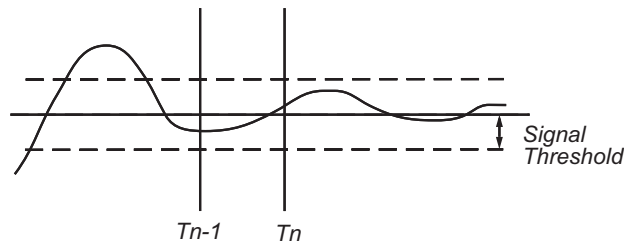
The Adaptive algorithm turns off zero-crossing bracketing (stops iterating) if either of the following are satisfied:

- 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).

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 “Enabled Subsystems” on page 5-4.
From Workspace	One: to detect when the input signal has a discontinuity in either the rising or falling direction
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.

Block	Description of Zero Crossing
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.
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
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: “Triggered Subsystems” on page 5-14.
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: “Triggered and Enabled Subsystems” on page 5-19

Note Zero-crossing detection is also available for a Stateflow® chart that uses continuous-time mode. See “Configuring 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 2-31 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.

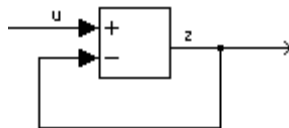
For the Saturation block’s upper limit, the direction of the zero crossing is *either*. This enables the entering and leaving saturation events to be detected using the same zero-crossing signal.

Algebraic Loops

Some Simulink blocks have input ports with *direct feedthrough*. This means that the output of these blocks cannot be computed without knowing the values of the signals entering the blocks at these input ports. Some examples of blocks with direct feedthrough inputs are as follows:

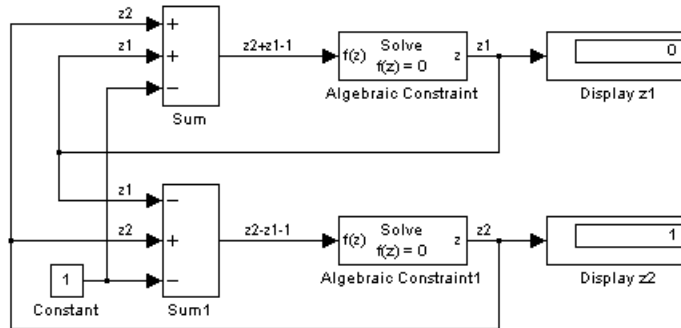
- Math Function block
- Gain block
- Integrator block's initial condition ports
- Product block
- State-Space block when there is a nonzero D matrix
- Sum block
- Transfer Fcn block when the numerator and denominator are of the same order
- Zero-Pole block when there are as many zeros as poles

An *algebraic loop* generally occurs when an input port with direct feedthrough is driven by the output of the same block, either directly, or by a feedback path through other blocks with direct feedthrough. An example of an algebraic loop is this simple scalar loop.



Mathematically, this loop implies that the output of the Sum block is an algebraic state z constrained to equal the first input u minus z (i.e., $z = u - z$). The solution of this simple loop is $z = u/2$, but most algebraic loops cannot be solved by inspection.

It is easy to create vector algebraic loops with multiple algebraic state variables $z1$, $z2$, etc., as shown in this model.



The Algebraic Constraint block is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal $F(z)$ to zero and outputs an algebraic state z . This block outputs the value necessary to produce a zero at the input. The output must affect the input through some direct feedback path, i.e., the feedback path solely contains blocks with direct feedthrough. You can provide an initial guess of the algebraic state value in the block's dialog box to improve algebraic loop solver efficiency.

A scalar algebraic loop represents a scalar algebraic equation or constraint of the form $F(z) = 0$, where z is the output of one of the blocks in the loop and the function F consists of the feedback path through the other blocks in the loop to the input of the block. In the simple one-block example shown on the previous page, $F(z) = z - (u - z)$. In the vector loop example shown above, the equations are

$$\begin{aligned} z2 + z1 - 1 &= 0 \\ z2 - z1 - 1 &= 0 \end{aligned}$$

Algebraic loops arise when a model includes an algebraic constraint $F(z) = 0$. This constraint might arise as a consequence of the physical interconnectivity of the system you are modeling, or it might arise because you are specifically trying to model a differential/algebraic system (DAE).

When a model contains an algebraic loop, a loop solving routine is called at each time step. The loop solver performs iterations to determine the solution to the problem (if it can). As a result, models with algebraic loops run slower than models without them.

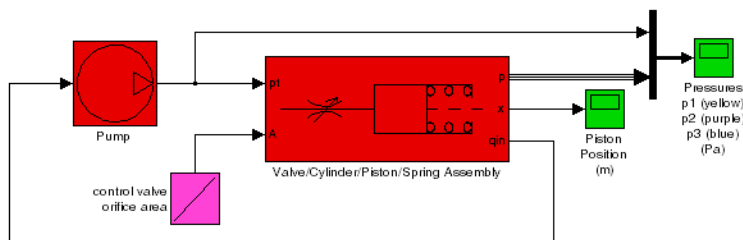
To solve $F(z) = 0$, the Simulink loop solver uses Newton's method with weak line search and rank-one updates to a Jacobian matrix of partial derivatives. Although the method is robust, it is possible to create loops for which the loop solver will not converge without a good initial guess for the algebraic states z . You can specify an initial guess for a line in an algebraic loop by placing an IC block (which is normally used to specify an initial condition for a signal) on that line. As shown above, another way to specify an initial guess for a line in an algebraic loop is to use an Algebraic Constraint block.

Whenever possible, use an IC block or an Algebraic Constraint block to specify an initial guess for the algebraic state variables in a loop.

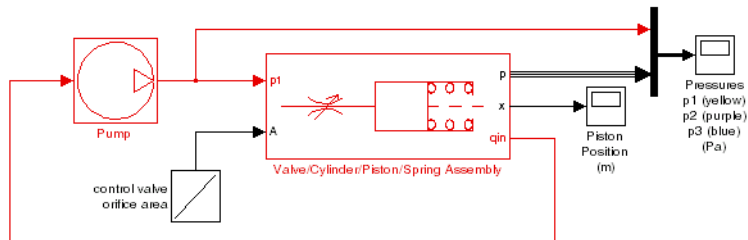
Highlighting Algebraic Loops

You can highlight algebraic loops when you update, simulate, or debug a model. Use the `ashow` command to highlight algebraic loops when debugging a model.

For example, the following figure shows the block diagram of the `hydcyl1` demo model in its original colors.



The following figure shows the diagram after updating when the Algebraic loop diagnostic is set to Error.



Eliminating Algebraic Loops

The Simulink software can eliminate some algebraic loops that include any of the following types of blocks:

- Atomic Subsystem
- Enabled Subsystem
- Model

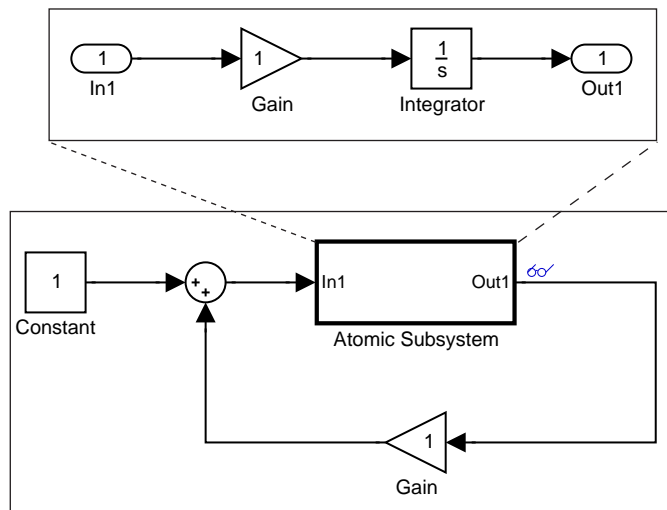
To enable automatic algebraic loop elimination for a loop involving a particular instance of an Atomic Subsystem or Enabled Subsystem block, select the **Minimize algebraic loop occurrences** parameter on the block's parameters dialog box. To enable algebraic loop elimination for a loop involving a Model block, select the **Minimize algebraic loop occurrences** parameter on the **Model Referencing Pane** of the Configuration Parameters dialog box (see "Model Referencing Pane" "The Model Referencing Pane" in the online documentation) of the model referenced by the Model block. If a loop includes more than one instance of these blocks, you should enable algebraic loop elimination for all of them, including nested blocks.

Note The Simulink software does not minimize algebraic loops on signals that are test points, even if you select **Minimize algebraic loop occurrences**

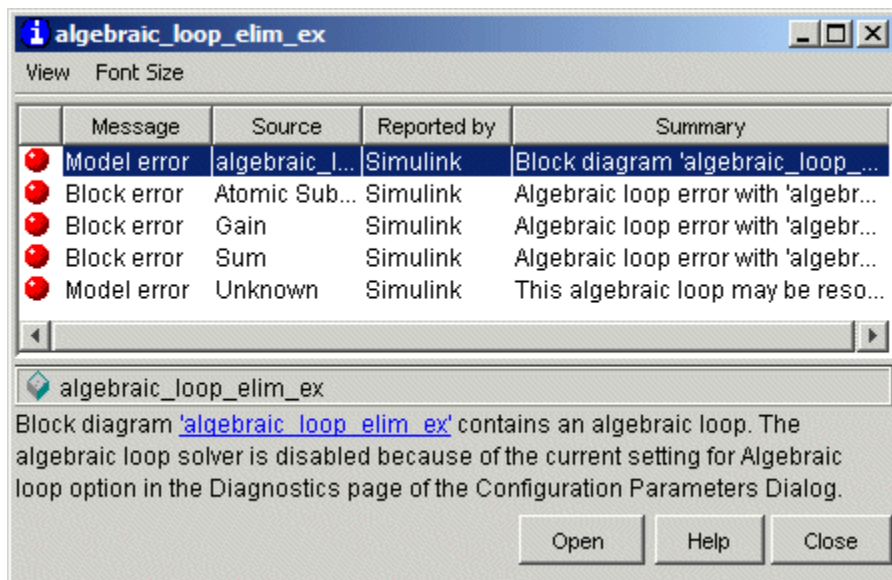
Algebraic loop minimization is off by default because it is incompatible with conditional input branch optimization in Simulink (see “Optimization Pane” “The Optimization Pane” in the online documentation) and with single output/update function optimization in Real-Time Workshop®. If you need these optimizations for an atomic or enabled subsystem or referenced model involved in an algebraic loop, you must eliminate the algebraic loop yourself.

The **Minimize algebraic loop** solver diagnostic allows you to specify the action Simulink should take, for example, display a warning, if it is unable to eliminate an algebraic loop involving a block for which algebraic loop elimination is enabled. See “Diagnostics Pane: Solver” “The Diagnostics Pane: Solver” in the online documentation for more information.

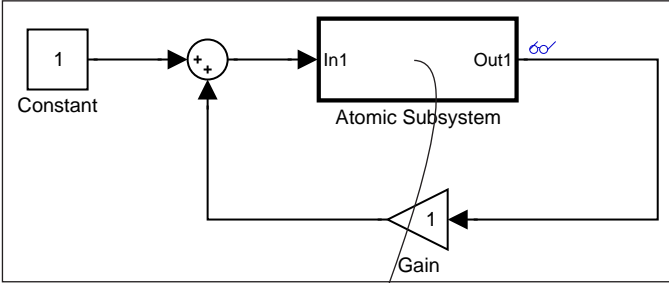
As an example of the ability of the Simulink software to eliminate algebraic loops, consider the following model.



Simulating this model with the solver's Algebraic Loop diagnostic set to error (see “Diagnostics Pane: Solver” “The Diagnostics Pane: Solver” in the online documentation for more information) reveals that this model contains an algebraic loop involving its atomic subsystem.



Checking the atomic subsystem's **Minimize algebraic loop occurrences** parameter eliminates the algebraic loop from the compiled version of the model.



Function Block Parameters: Atomic Subsystem

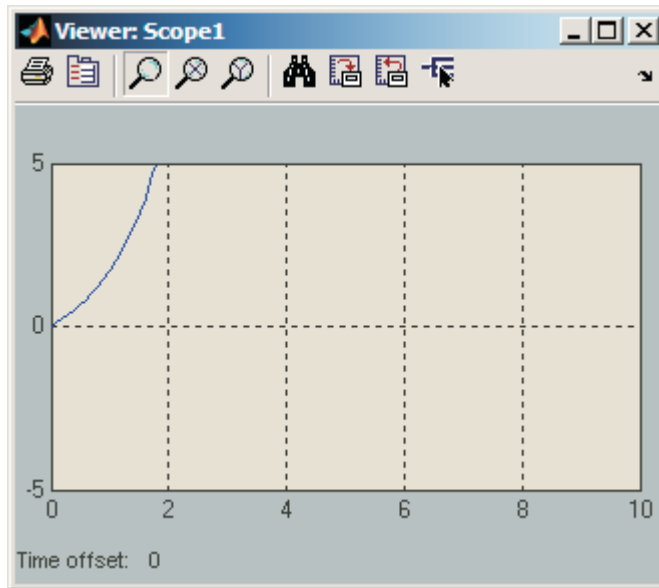
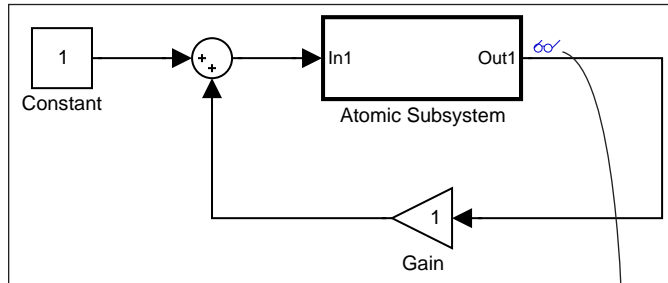
Subsystem
Select the settings for the subsystem block.

Parameters

- Show port labels
- Read/Write permissions:
- Name of error callback function:
- Permit hierarchical resolution:
- Treat as atomic unit
- Minimize algebraic loop occurrences
- Sample time [-1 for inherited]:
- Real-Time Workshop system code:

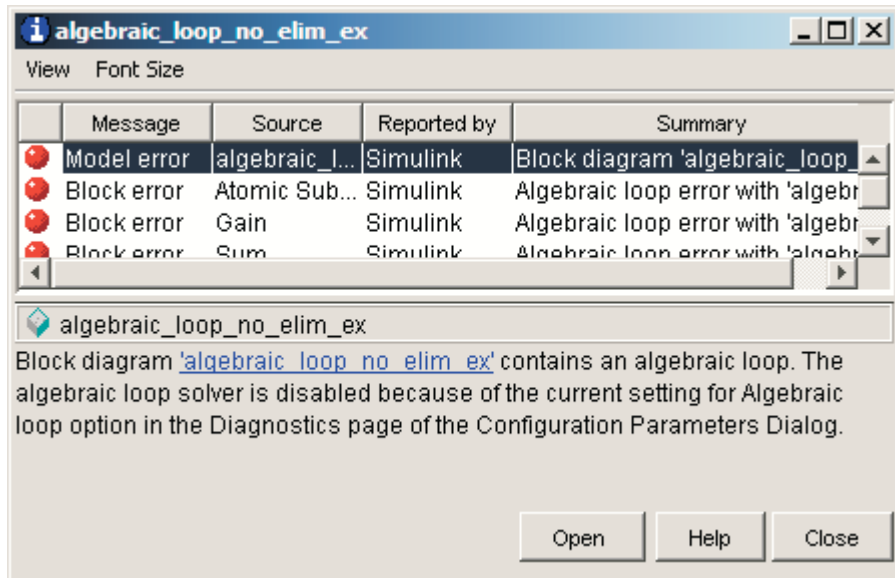
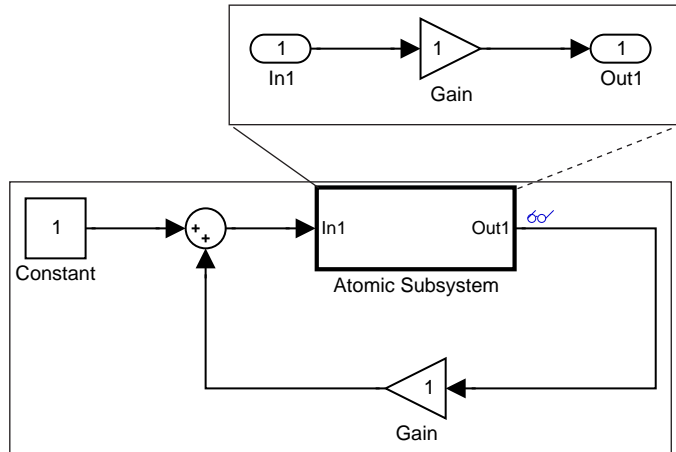
OK Cancel Help Apply

As a result, the model now simulates without error.



Note that the Simulink software is able to eliminate the algebraic loop involving this model's atomic subsystem because the atomic subsystem contains a block with a port that does not have direct feedthrough, i.e., the Integrator block.

If you remove the Integrator block from the atomic subsystem, the algebraic loop cannot be eliminated. Hence, attempting to simulate the model results in an error.



Working with Sample Times

- “What Is Sample Time?” on page 3-2
- “How to Specify the Sample Time” on page 3-3
- “How to View Sample Time Information” on page 3-9
- “How to Print Sample Time Information” on page 3-13
- “Types of Sample Time” on page 3-15
- “Determining the Compiled Sample Time of a Block” on page 3-20
- “Managing Sample Times in Subsystems” on page 3-21
- “Managing Sample Times in Systems” on page 3-22
- “Resolving Rate Transitions” on page 3-30
- “How Propagation Affects Inherited Sample Times” on page 3-31
- “Monitoring Backpropagation in Sample Times” on page 3-33

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.

How to Specify the Sample Time

In this section...

- “Designating Sample Times” on page 3-3
- “Specifying Block-Based Sample Times Interactively” on page 3-5
- “Specifying Port-Based Sample Times Interactively” on page 3-6
- “Specifying Block-Based Sample Times Programmatically” on page 3-7
- “Specifying Port-Based Sample Times Programmatically” on page 3-7
- “Accessing Sample Time Information Programmatically” on page 3-8
- “Specifying Sample Times for a Custom Block” on page 3-8
- “Determining Sample Time Units” on page 3-8
- “Changing the Sample Time After Simulation Start Time” on page 3-8

Designating Sample Times

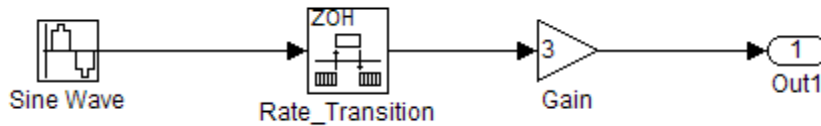
Simulink allows you to specify a block sample time directly as a numerical value or symbolically by defining a sample time vector. In the case of a discrete sample time, the vector is $[T_s, T_o]$ where T_s is the sampling period and T_o is the initial time offset. For example, consider a discrete model 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 3-15. 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 “How to View Sample Time Information” on page 3-9.)

Designations of Sample Time Information

Sample Time Type	Sample Time	Color	Annotation	Explicit
Discrete	$[T_s, T_o]$	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	$[-\infty, 0]$	magenta	Inf	Yes
Variable	$[-2, T_{vo}]$	brown	V1, V2,... Vi	No
Hybrid	N/A	yellow	N/A	No
Triggered	$[-1, -1]$	cyan	T	No
Asynchronous	$[-1, -n]$	purple	A1, A2,... Ai	No

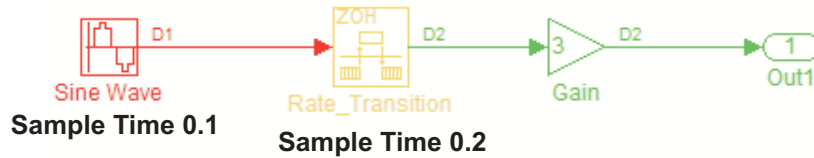
You can use the explicit sample time values in this table to specify sample times interactively or programmatically for either block-based or port-based sample times. The following model, Specify_Sample_Time.mdl, serves as a reference for this section.



Specify_Sample_Time.mdl

In this example, our goal is to set the sample time of the input sine wave signal to 0.1 and 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.)



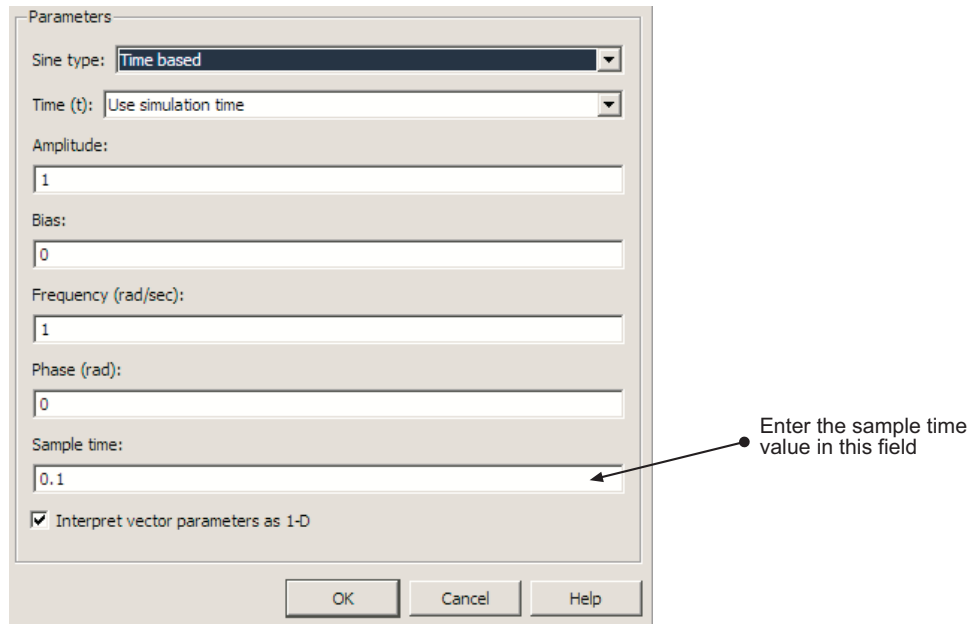
Specify_Sample_Time.mdl after Setting Sample Times

Specifying 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**.

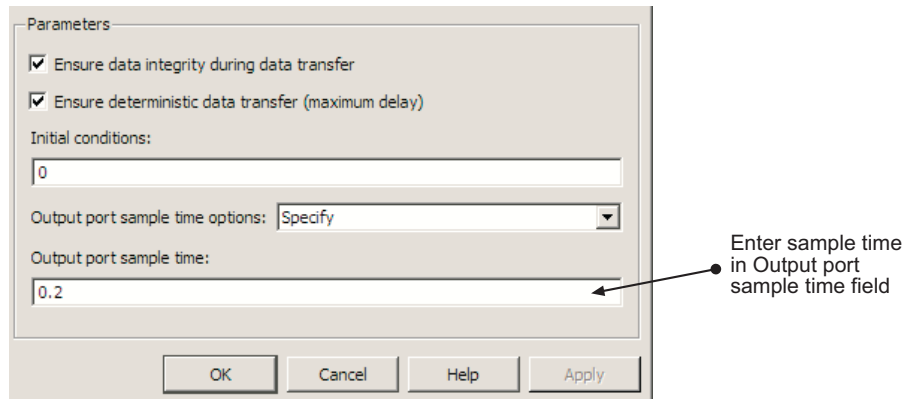
Following is a figure of a parameters dialog box for the Sine Wave block after entering 0.1 in the **Sample time** field.



Specifying 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 **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.

Specifying 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')
```

Specifying 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')
```

Accessing 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`.

Specifying 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.

Changing 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.

How to View Sample Time Information

In this section...
“Viewing Sample Time Display” on page 3-9
“Managing the Sample Time Legend” on page 3-10

Viewing Sample Time Display

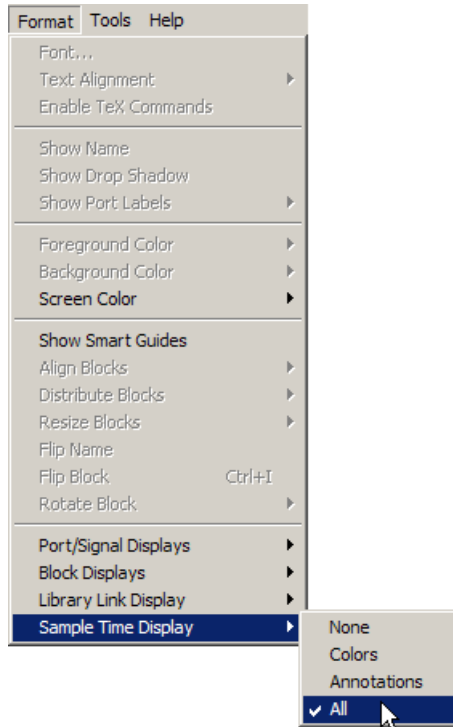
Simulink models can display color coding and annotations that represent specific sample times. As shown in the table Designations of Sample Time Information on page 3-4, each sample time type has one or more colors associated with it. You can display the blocks and signal lines in color, the annotations in black, or both the colors and the annotations. To choose one of these options:

- 1 In the Simulink model window, select **Format > Sample Time Display**.
- 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 **Format > Sample Time Display**.
- 2 Select **None**.

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 3-4 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.

Managing the 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 Display** 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 **Edit > Update Diagram**.
- 2** Select **View > Sample Time Legend** or press **Ctrl +J**.

In addition, whenever you select **Colors**, **Annotations**, or **All** from the **Sample Time Display** 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 **None**, 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. 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, you must repeat the **Update Diagram** to access the latest known information.

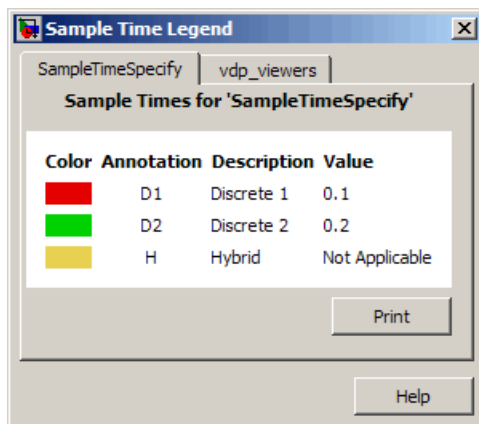
Turning the Legend Off

If you prefer not to view the legend upon selecting **Sample Time Display**:

- 1 In the Simulink model window, select **File > 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

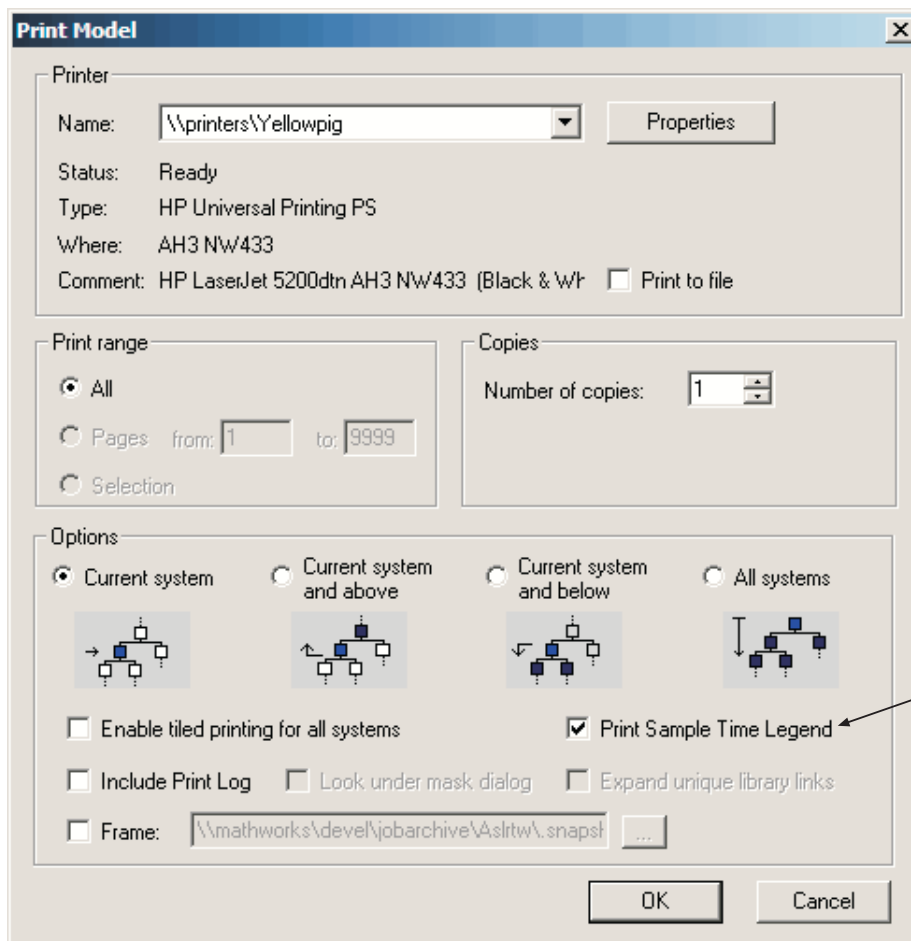
If you have more than one model open and you view the **Sample Time Legend** for each one, a single legend window appears with multiple tabs. Each tab contains the name of the model and the respective legend information. The following figure shows the tabbed legends for SampleTimeSpecify.mdl and vdp_viewers.mdl.



How to 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**.



Select the Print Sample Time Legend option.

Types of Sample Time

In this section...

“Discrete Sample Time” on page 3-15
 “Continuous Sample Time” on page 3-16
 “Fixed in Minor Step” on page 3-16
 “Inherited Sample Time” on page 3-16
 “Constant Sample Time” on page 3-17
 “Variable Sample Time” on page 3-18
 “Triggered Sample Time” on page 3-19
 “Asynchronous Sample Time” on page 3-19

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 2-24). The ODE solver you choose integrates all continuous states from the simulation start time to a given major or minor time step. The solver determines the times of the minor steps and uses the results at the minor time steps to improve the accuracy of the results at the major time steps. However, you see the block output only at the major time steps.

To specify that a block, such as the Derivative block, is continuous, enter [0, 0] or 0 in the **Sample time** field of the block dialog.

Fixed in Minor Step

If the sample time of a block is set to [0, 1], the block becomes *fixed in minor step*. For this setting, Simulink does not execute the block at the minor time steps; updates occur only at the major time steps. This process eliminates unnecessary computations of blocks whose output cannot change between major steps.

While you can explicitly set a block to be fixed in minor step, more typically Simulink sets this condition as either an inherited sample time or as an alteration to a user specification of 0 (continuous). This setting is equivalent to, and therefore converted to, the fastest discrete rate when you use a fixed-step solver.

Inherited Sample Time

If a block sample time is set to [-1, 0] or -1, the sample time is *inherited* and Simulink determines the best sample time for the block based on the block context within the model. Simulink performs this task during the compilation stage; the original inherited setting never appears in a compiled model. Therefore, you never see inherited ([-1, 0]) in the Sample Time Legend. (See “How to View Sample Time Information” on page 3-9.)

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 3-31

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** option is enabled in the **Optimization** pane of the Configuration Parameters dialog box.
- 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 “Enabled Subsystems” on page 5-4).
- 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 S-Function, the Level-2 M-File S-Function, and the Model blocks, 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.

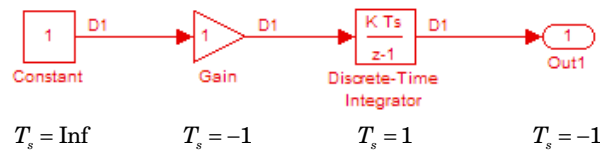
If Simulink cannot honor the request to use the optimization, then Simulink treats the block as if it had specified an inherited sample time (-1).

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 3-31) to determine the block sample time.

An example of a Constant block with tunable parameters is shown below. 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 “Function-Call Subsystems” on page 5-24.)
- 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 “Handling 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”.

Determining the Compiled Sample Time of a Block

During the compilation phase of a simulation, Simulink determines the sample time of a block from its `SampleTime` parameter (if it has an explicit sample time), its block type (if it has an implicit sample time), or by its context within the model. 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.

Managing Sample Times in Subsystems

Subsystems fall into two categories: triggered and nontriggered. 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.

The four nontriggered subsystems are virtual, enabled, atomic, and 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.

Managing Sample Times in Systems

In this section...
“Purely Discrete Systems” on page 3-22
“Hybrid Systems” on page 3-25

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 greatest 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.



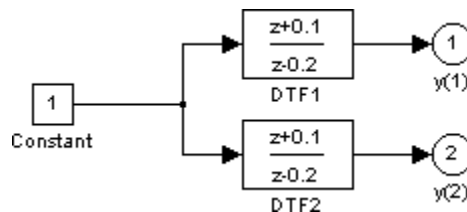
Fixed-Step Solver



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 Real-Time Workshop). 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.

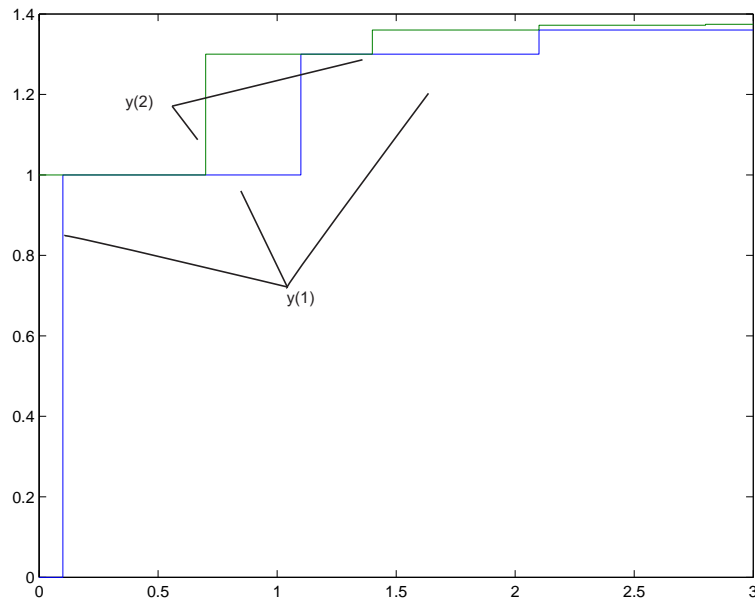
Consider the following example of a simple multirate system. For this example, the DTF1 Discrete Transfer Fcn block's **Sample time** is set to [1 0.1], which gives it an offset of 0.1. The **Sample time** of the DTF2 Discrete Transfer Fcn block is set to 0.7, with no offset.



Running the simulation and plotting the outputs using the stairs function

```
[t,x,y] = sim('multirate', 3);  
stairs(t,y)
```

produces the following plot.



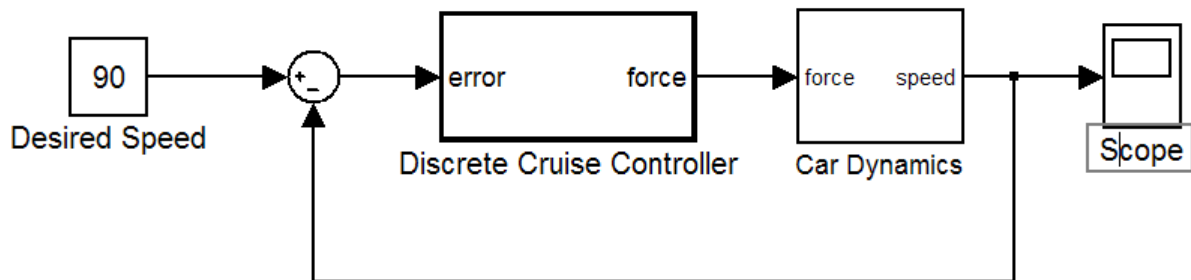
(See Chapter 22, “Running a Simulation Programmatically” for information on the `sim` command.)

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

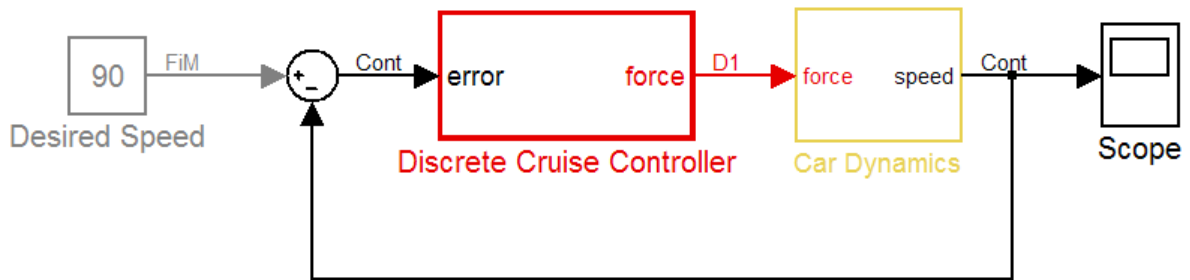
Hybrid systems contain both discrete and continuous blocks and thus have both discrete and continuous states. However, Simulink solvers treat any system that has both continuous and discrete sample times as a hybrid system. For information on modeling hybrid systems, see “Modeling Hybrid Systems” on page 2-8.

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”.

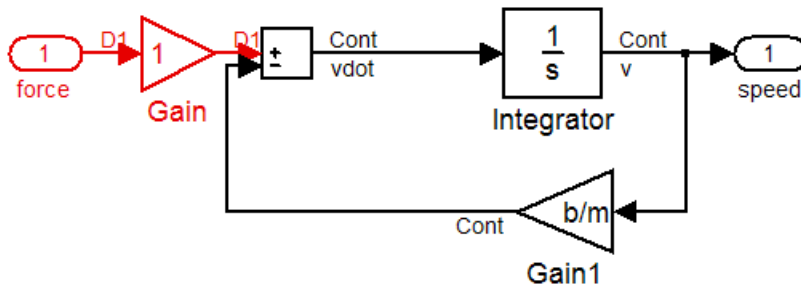


Car Model

With the **Sample Time Display** 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.

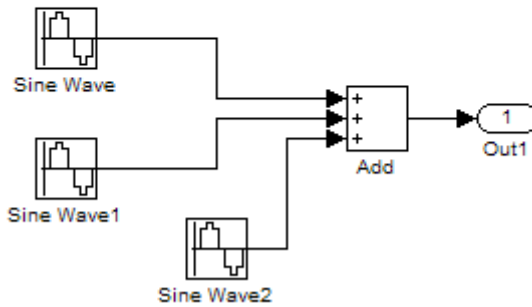
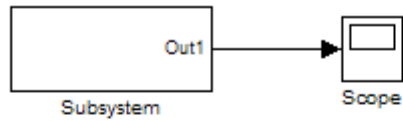


Car Model after an Update Diagram



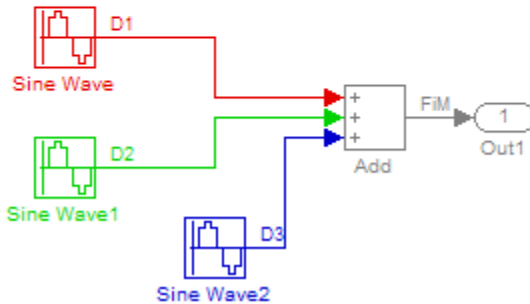
Car Dynamics Subsystem after an Update Diagram

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

An **Update Diagram** turns the subsystem yellow because the subsystem contains more than one sample time. As shown in the Sample Time Legend, the Sine Wave blocks have discrete sample times D1, D2, and D3 and the output signal is fixed in minor step.



Sample Time Legend

MultiRateDiscrete

Sample Times for 'MultiRateDiscrete'

Color	Annotation	Description	Value
Grey	FIM	Fixed in Minor Step	[0,1]
Red	D1	Discrete 1	0.2
Green	D2	Discrete 2	0.3
Blue	D3	Discrete 3	0.4
Yellow	H	Hybrid	Not Applicable

Multirate Subsystem after an Update Diagram

In assessing a system for multiple sample times, Simulink does not consider either constant $[\infty, 0]$ 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.

Resolving 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:

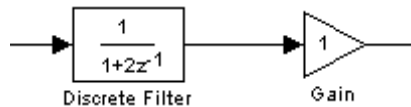
- A constant sample time ($[\text{Inf}, 0]$) never has a rate transition with any other rate.
- A continuous sample time (black) and the fastest discrete rate (red) never has a rate transition if you use a fixed-step solver.
- A variable sample time and fixed in minor step do not have a rate transition.

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 “Handling 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

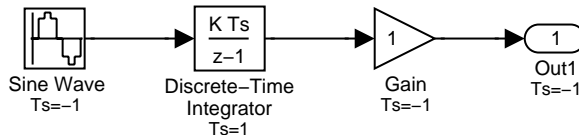
A block having a block-based sample time inherits a sample time based on the sample times of the blocks connected to its inputs, and in accordance with the following rules:

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

If...	then
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.

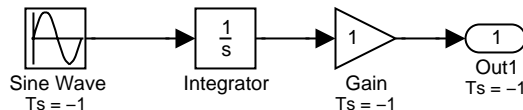
Monitoring Backpropagation in Sample Times

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 Display > Colors** from the Simulink **Format** 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 below, causes the Sine Wave and Gain blocks to change to continuous blocks. You can test this change by selecting **Update Diagram** from the **Edit** menu to update the colors; both blocks now appear black.



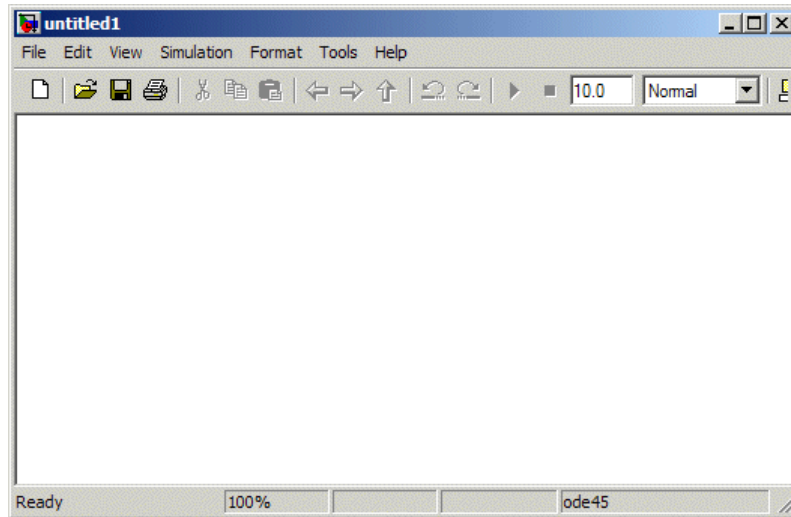
Note Backpropagation makes the sample times of model sources dependent on block connectivity. If you change the connectivity of a model whose sources inherit sample times, you can inadvertently change the source sample times. For this reason, when you update or simulate a model, by default, Simulink displays warnings at the command line if the model contains sources that inherit their sample times.

Creating a Model

- “Creating an Empty Model” on page 4-2
- “Populating a Model” on page 4-4
- “Selecting Objects” on page 4-7
- “Specifying Block Diagram Colors” on page 4-9
- “Connecting Blocks” on page 4-15
- “Aligning, Distributing, and Resizing Groups of Blocks Automatically” on page 4-24
- “Annotating Diagrams” on page 4-26
- “Creating Subsystems” on page 4-37
- “Modeling Control Flow Logic” on page 4-44
- “Using Callback Functions” on page 4-54
- “Using Model Workspaces” on page 4-66
- “Resolving Symbols” on page 4-75
- “Consulting the Model Advisor” on page 4-80
- “Managing Model Versions” on page 4-99
- “Model Discretizer” on page 4-114

Creating an Empty Model

To create an empty model, click the **New** button on the Library Browser's toolbar, or choose **New** from the library window's **File** menu and select **Model**. An empty model is created in memory and it is displayed in a new model editor window.



Creating 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, you can use the Simulink software model construction commands described in “Model Construction” to write a function that creates a model with the defaults you prefer. For example, the following function creates a model that has a green canvas and a hidden toolbar, 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');

% set default toolbar visibility
set_param(modelname, 'Toolbar', 'off');

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

Populating a Model

In this section...

“About the Library Browser” on page 4-4

“Opening the Library Browser” on page 4-4

“Browsing Block Libraries” on page 4-5

“Searching Block Libraries” on page 4-5


“Cloning Blocks to Models” on page 4-6

About the Library Browser

Simulink provides the Simulink Library Browser, which you can use to browse, search, and clone blocks from built-in and user libraries to your model. This section summarizes the techniques for using the Library Browser. For additional details, see “Library Browser”. For information about creating your own libraries and adding them to the Library Browser, see Chapter 8, “Working with Block Libraries”.

Opening the Library Browser

To open the Simulink Library Browser, do one of the following:

- Click the **Library Browser** button  in the toolbar of the MATLAB Desktop or a Simulink Model Editor window
- In the MATLAB Command Window, enter:

```
simulink
```

If you have not already loaded Simulink, a short delay occurs while it loads. The Library Browser opens.

To keep the Library Browser above all other windows on your desktop, select the **Pushpin** button on the browser’s toolbar.

Browsing Block Libraries

The **Libraries** pane on the left shows displays a tree-structured directory of the block libraries installed 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 see what libraries are installed on your system and what blocks they contain.

The contents of the library selected 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. Each member of the selected library is represented by an icon and a name. A library's icon suggests the purpose of the library. A block's library icon is the same as the block's icon when cloned into a model.

You can open a sublibrary by either selecting it in the **Libraries** pane or double-clicking it in the **Library** tab. To see a block's Block parameters dialog, double-click the block in the Library tab. To get Help for a block, right-click it and select Help from its context menu. The Help text, and the context menu itself, are the same that would appear if you right-clicked an instance of that block in a model.

Searching Block Libraries

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

- 1** Enter the character string in the text field of the Library Browser's **Search** field.
- 2** Press Return or click the tool's **Search** button.

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

By default, the search finds any substring and is not case-sensitive. You can change these defaults, or enable use of MATLAB regular expressions in the **Search** field, by clicking the **Library Browser Options** button and selecting appropriate commands. You can work with blocks found by **Search** just as you could with blocks found by selecting a library.

Cloning 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 you selected. The resulting block retains a link to its library, so that updates to the source library automatically propagate to all copies. See Chapter 8, “Working with Block Libraries” for information about library links.

Selecting Objects

In this section...

“Selecting an Object” on page 4-7

“Selecting Multiple Objects” on page 4-7

Selecting an Object

To select an object, click it. Small black square handles appear at the corners of a selected block and near the end points of a selected line. For example, the figure below shows a selected Sine Wave block and a selected line.



When you select an object by clicking it, any other selected objects are deselected.

Selecting Multiple Objects

You can select more than one object either by selecting objects one at a time, by selecting objects located near each other using a bounding box, or by selecting the entire model.

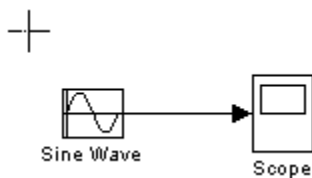
Selecting Multiple Objects One at a Time

To select more than one object by selecting each object individually, hold down the **Shift** key and click each object to be selected. To deselect a selected object, click the object again while holding down the **Shift** key.

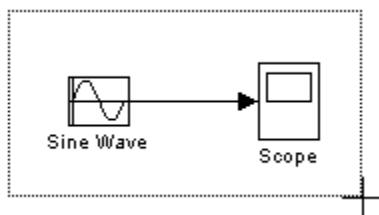
Selecting Multiple Objects Using a Bounding Box

An easy way to select more than one object in the same area of the window is to draw a bounding box around the objects:

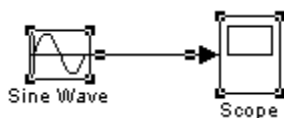
- 1 Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



- 2 Drag the pointer to the opposite corner of the box. A dotted rectangle encloses the selected blocks and lines.



- 3 Release the mouse button. All blocks and lines at least partially enclosed by the bounding box are selected.



Selecting All Objects

To select all objects in the active window, select **Select All** from the **Edit** menu. You cannot create a subsystem by selecting blocks and lines in this way. For more information, see “Creating Subsystems” on page 4-37.

Specifying Block Diagram Colors

In this section...

“How to Specify Block Diagram Colors” on page 4-9

“Choosing a Custom Color” on page 4-10

“Defining a Custom Color” on page 4-10

“Specifying Colors Programmatically” on page 4-11

“Displaying Sample Time Colors” on page 4-12

How to Specify Block Diagram Colors

You can specify the foreground and background colors of any block or annotation in a diagram, as well as the diagram’s background color. To set the background color of a block diagram, select **Screen color** from the **Format** menu. To set the background color of a block or annotation or group of such items, first select the item or items. Then select **Background color** from the **Format** menu. To set the foreground color of a block or annotation, first select the item. Then select **Foreground color** from the **Format** menu.

In all cases, a menu of color choices is displayed. 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 is changed to the selected color.

Choosing a Custom Color

If you choose **Custom**, The Simulink Choose Custom Color dialog box is displayed.

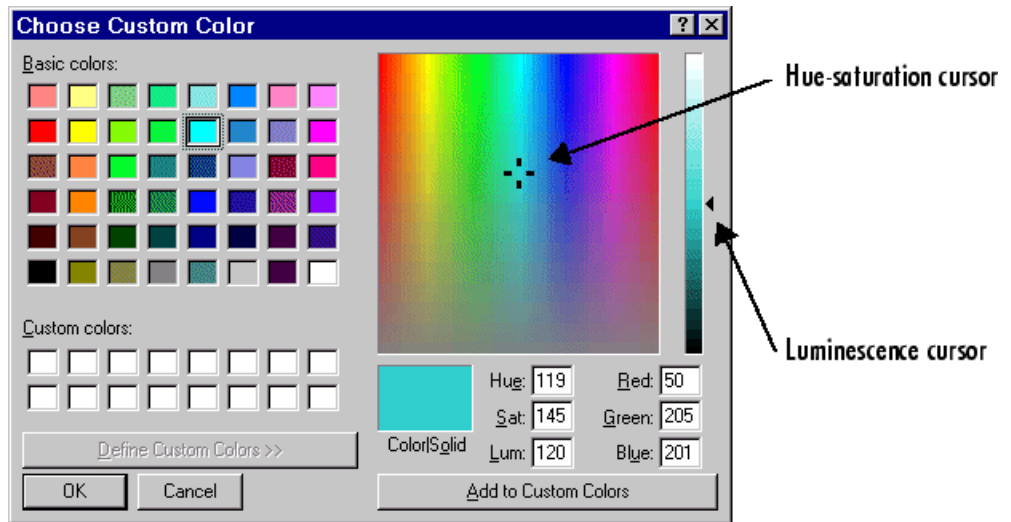


The dialog box displays a palette of basic colors and a palette of custom colors that you previously defined. If you have not previously created any custom colors, the custom color palette is all white. To choose a color from either palette, click the color, and then click the **OK** button.

Defining a Custom Color

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

The dialog box expands to display a custom color definer.



The color definer allows you to specify a custom color by

- Entering the red, green, and blue components of the color as values between 0 (darkest) and 255 (brightest)
- Entering hue, saturation, and luminescence components of the color as values in the range 0 to 255
- Moving 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

The color that you have defined in any of these ways appears in the **Color|Solid** box. To redefine a color in the **Custom colors** palette, select the color and define a new color, using the color definer. Then click the **Add to Custom Colors** button on the color definer.

Specifying Colors Programmatically

You can use the `set_param` command at the MATLAB command line or in an M-file program to set parameters that determine the background color of a

diagram and the background color and foreground color of diagram elements. The following table summarizes the parameters that control block diagram colors.

Parameter	Determines
ScreenColor	Background color of block diagram
BackgroundColor	Background color of blocks and annotations
ForegroundColor	Foreground color of blocks and annotations

You can set these parameters to any of the following values:

- 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'
- '[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]')
```

Displaying Sample Time Colors

The blocks and lines in your model can be color coded to indicate the sample rates at which the blocks operate.

Color	Use
Black	Continuous sample time
Magenta	Constant sample time
Red	Fastest discrete sample time
Green	Second fastest discrete sample time
Blue	Third fastest discrete sample time
Light Blue	Fourth fastest discrete sample time

Color	Use
Dark Green	Fifth fastest discrete sample time
Orange	Sixth, seventh, eighth, etc., fastest discrete sample time
Yellow	Indicates a block with hybrid sample time, e.g., subsystems grouping blocks and Mux or Demux blocks grouping signals with different sample times, Data Store Memory blocks updated and read by different tasks.
Cyan	Blocks in triggered subsystems
Brown	Variable sample time. See the Pulse Generator block and “How to Specify the Sample Time” on page 3-3 for more information
Gray	Fixed in minor step

To enable the sample time colors feature, select **Sample Time Colors** from the **Format** menu.

The Simulink software does not automatically recolor the model with each change you make to it, so you must select **Update Diagram** from the **Edit** menu to explicitly update the model coloration. To return to your original coloring, disable sample time coloration by again choosing **Sample Time Colors**.

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 Chapter 6, “Referencing a Model”.)

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.

Connecting Blocks

In this section...

“Automatically Connecting Blocks” on page 4-15

“Manually Connecting Blocks” on page 4-18

“Disconnecting Blocks” on page 4-23

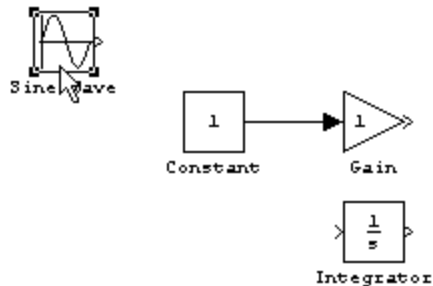
Automatically Connecting Blocks

You can command the Simulink software to connect blocks automatically. This eliminates the need for you to draw the connecting lines yourself. When connecting blocks, the lines are routed around intervening blocks to avoid cluttering the diagram.

Connecting Two Blocks

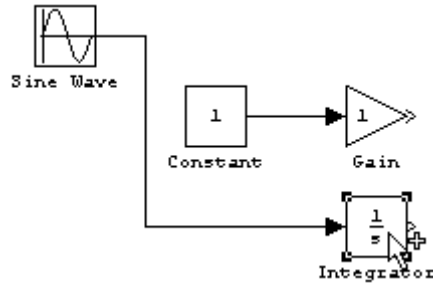
To autoconnect two blocks:

- 1 Select the source block.

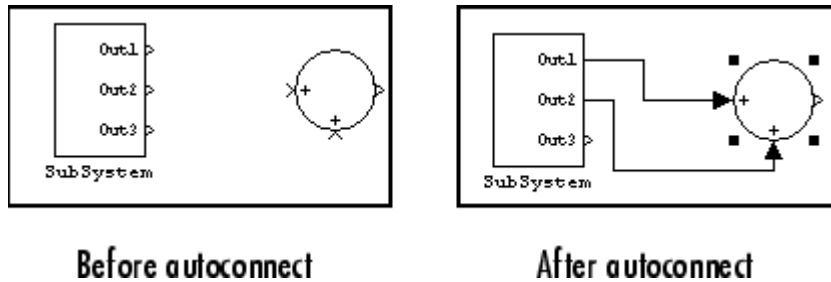


- 2 Hold down **Ctrl** and left-click the destination block.

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



When connecting two blocks, the Simulink software draws as many connections as possible between the two blocks as illustrated in the following example.

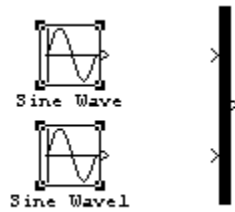


Connecting Groups of Blocks

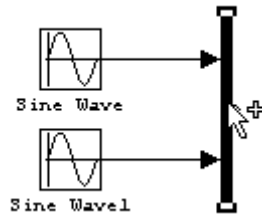
The Simulink software can connect a group of source blocks to a destination block or a source block to a group of destination blocks.

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

- 1 Select the source blocks.

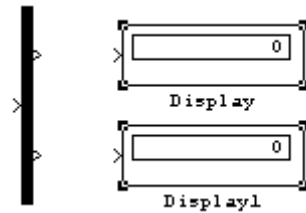


2 Hold down **Ctrl** and left-click the destination block.

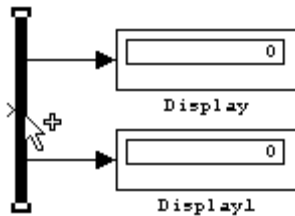


To connect a source block to a group of destination blocks:

1 Select the *destination* blocks.



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



Manually Connecting 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.

Drawing a Line Between Blocks

To connect the output port of one block to the input port of another block:

- 1 Position the cursor over the first block's output port. It is not necessary to position the cursor precisely on the port.

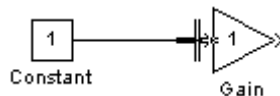
The cursor shape changes to crosshairs.



- 2 Press and hold down the mouse button.

- 3 Drag the pointer to the second block's input port. You can position the cursor on or near the port or in the block. If you position the cursor in the block, the line is connected to the closest input port.

The cursor shape changes to double crosshairs.



- 4 Release the mouse button. The port symbols are replaced by a connecting line with an arrow showing the direction of the signal flow. You can create lines either from output to input, or from input to output.

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

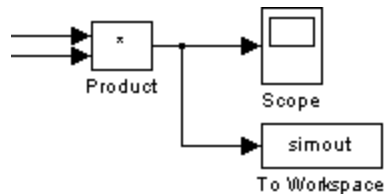


The Simulink software draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

Drawing 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. Using branch lines enables you to connect a signal to more than one block.

This example demonstrates the connecting of the Product block output to both the Scope block and the To Workspace block.



To add a branch line:

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

You can also use the right mouse button instead of holding down the left mouse button and the **Ctrl** key.

Drawing a Line Segment

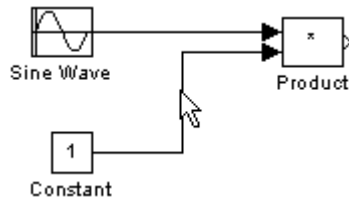
You might want to draw a line with segments exactly where you want them instead of where the Simulink software draws them. Or you might want to draw a line before you copy the block to which the line is connected. You can do either by drawing line segments.

To draw a line segment, you draw a line that ends in an unoccupied area of the diagram. An arrow appears on the unconnected end of the line. To add another line segment, position the cursor over the end of the segment and draw another segment. The segments are drawn as horizontal and vertical lines. To draw diagonal line segments, hold down the **Shift** key while you draw the lines.

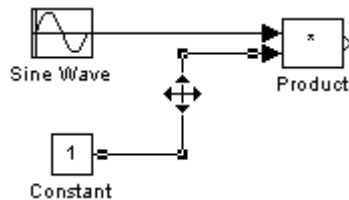
Moving a Line Segment

To move a line segment:

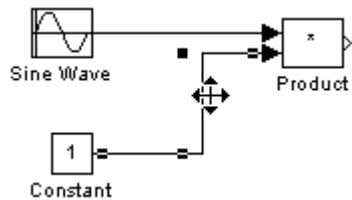
- 1 Position the pointer on the segment you want to move.



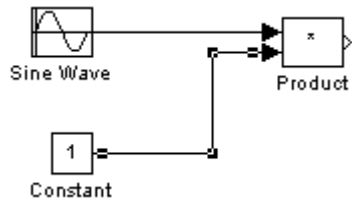
- 2 Press and hold down the left mouse button.



- 3 Drag the pointer to the desired location.



- 4 Release the mouse button.



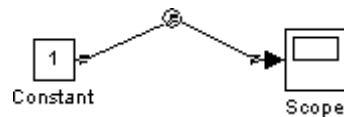
To move the segment connected to an input port, position the pointer over the port and drag the end of the segment to the new location. You cannot move the segment connected to an output port.

Moving a Line Vertex

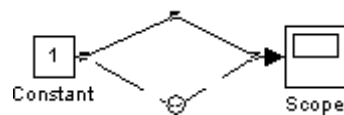
To move a vertex of a line:

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

The cursor changes to a circle that encloses the vertex.



- 2 Drag the pointer to the desired location.



- 3 Release the mouse button.

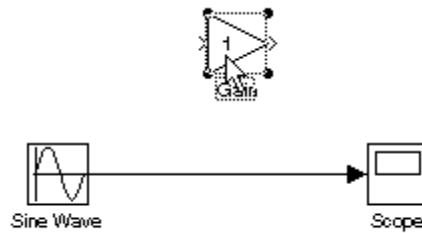


Inserting Blocks in a Line

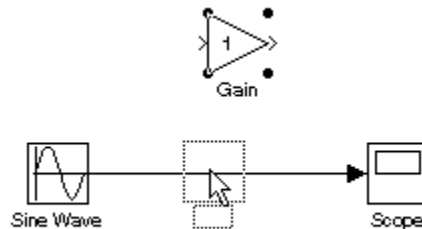
You can insert a block in a line by dropping the block on the line. The Simulink software inserts the block for you at the point where you drop the block. The block that you insert can have only one input and one output.

To insert a block in a line:

- 1 Position the pointer over the block and press the left mouse button.

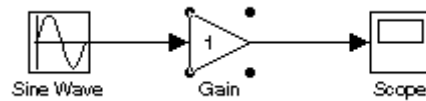


- 2 Drag the block over the line in which you want to insert the block.



- 3 Release the mouse button to drop the block on the line.

The block is inserted where you dropped it.



Disconnecting Blocks

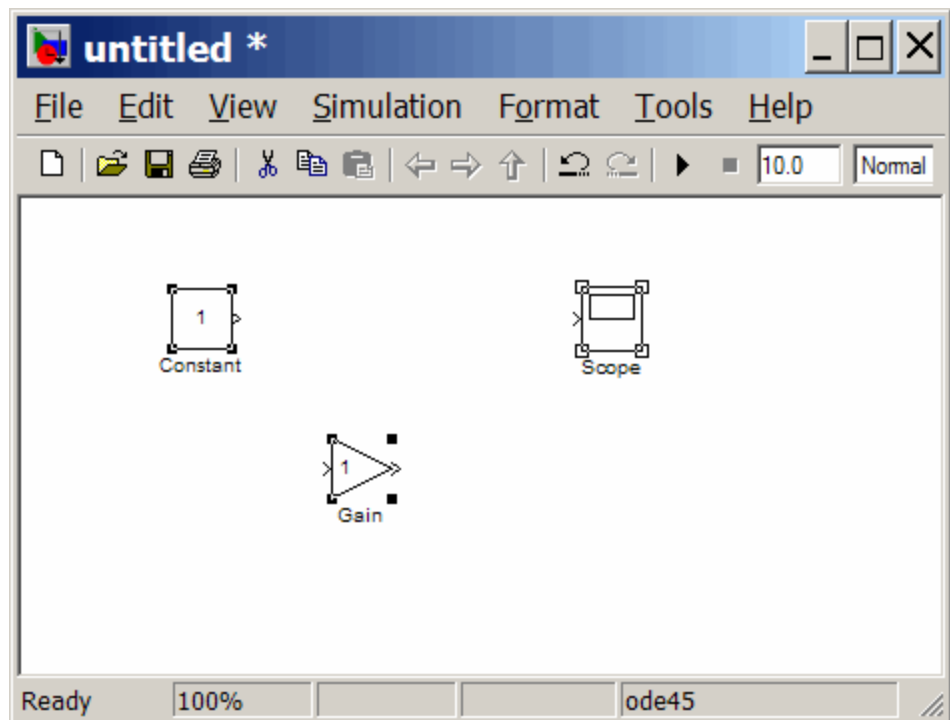
To disconnect a block from its connecting lines, hold down the **Shift** key, then drag the block to a new location.

To disconnect a line from a block's input port, position the mouse pointer over the line's arrowhead. The pointer turns into a circle. Drag the arrowhead away from the block.

Aligning, Distributing, and Resizing Groups of Blocks Automatically

The model editor's **Format** menu includes commands that let you quickly align, distribute, and resize groups of blocks. To align (or distribute or resize) a group of blocks:

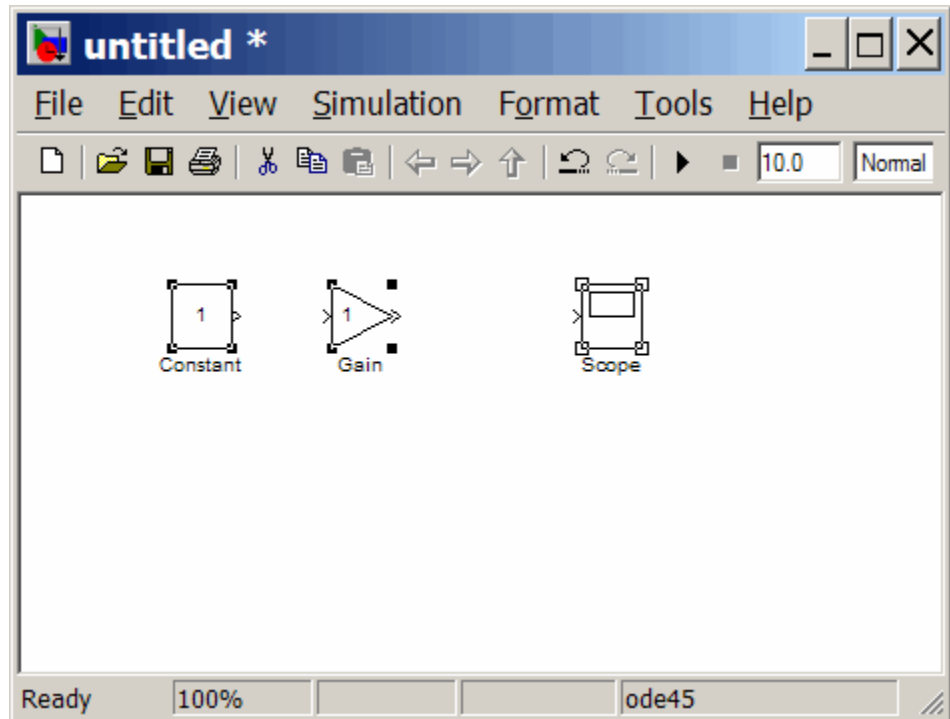
- 1 Select the blocks that you want to align.



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 Select one of the alignment options from the editor's **Format > Align Blocks** menu (or distribution options from the **Format > Distribute**

Blocks or resize options from the **Format > Resize Blocks** menu). For example, selecting **Align Top Edges** aligns the top edges of the selected blocks with the top edge of the reference block.



Annotating Diagrams

In this section...

“How to Annotate Diagrams” on page 4-26

“Annotations Properties Dialog Box” on page 4-27

“Annotation Callback Functions” on page 4-30

“Associating Click Functions with Annotations” on page 4-31

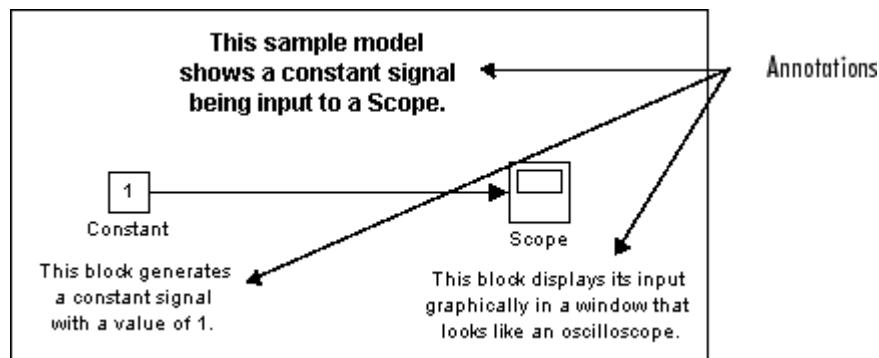
“Annotations API” on page 4-33

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

“Creating Annotations Programmatically” on page 4-35

How to Annotate Diagrams

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



To create a model annotation, double-click an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point. Start typing the annotation contents. Each line is centered in the rectangle that surrounds the annotation.

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

To edit an annotation, select it:

- To replace the annotation, click the annotation, then double-click or drag the cursor to select it. Then, enter the new 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 delete an annotation, hold down the **Shift** key while you select the annotation, then press the **Delete** or **Backspace** key.

To change an annotation's font, select the annotation, then choose **Font** from the **Format** menu. Select a font and size from the dialog box.

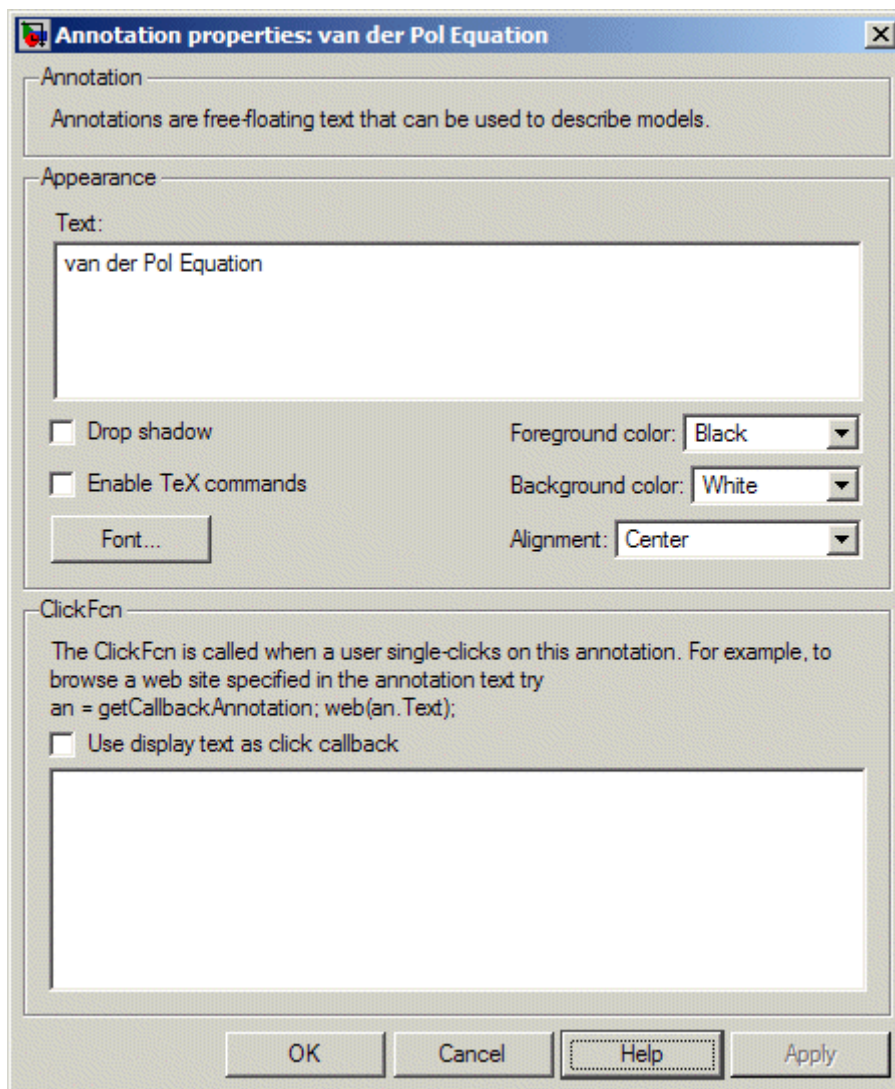
To change the text alignment (e.g., left, center, or right) of the annotation, select the annotation and choose **Text Alignment** from model editor's **Format** or the context menu. Then choose one of the alignment options (e.g., **Center**) from the **Text Alignment** submenu.

Annotations Properties Dialog Box

The Annotation Properties dialog box allows you to specify the contents and format of the currently selected annotation and to associate a click function with the annotation.

To display the Annotation Properties dialog box for an annotation, select the annotation and then select Annotation Properties from model editor's **Edit** or the context menu.

The dialog box appears.



The dialog box includes the following controls.

Text

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

Drop shadow

Checking this option causes a drop shadow to be displayed around the annotation, giving it a 3-D appearance.

Enable TeX commands

Checking this option enables use of TeX formatting commands in this annotation. See “Using TeX Formatting Commands in Annotations” in the online Simulink documentation for more information.

Font

Clicking this button 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 annotation’s bounding box.

Alignment

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

ClickFcn

Specifies MATLAB code to be executed when a user single-clicks this annotation. The Simulink software stores the code entered in this field with the model. See “Associating Click Functions with Annotations” on page 4-31 for more information.

Use display text as click callback

Checking this option causes the text in the **Text** field to be treated as the annotation's 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. See “Associating Click Functions with Annotations” on page 4-31 for more information. Note that selecting this option disables the **ClickFcn** edit field.

Annotation Callback Functions

You can associate the following callback functions with annotations.

Click Function

A click function is an M function that the Simulink software invokes when a user single-clicks an annotation. You can associate a click function with any of a model's annotations (see “Associating Click Functions with Annotations” on page 4-31). The Simulink software uses the color blue to display the text of annotations associated with click functions. This allows the user to see at a glance which annotations are associated with click functions. Click functions allow you to add hyperlinks and custom command “buttons” to your model's block diagram. For example, you can use click functions 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. (See Chapter 6, “Referencing a Model”.)

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).

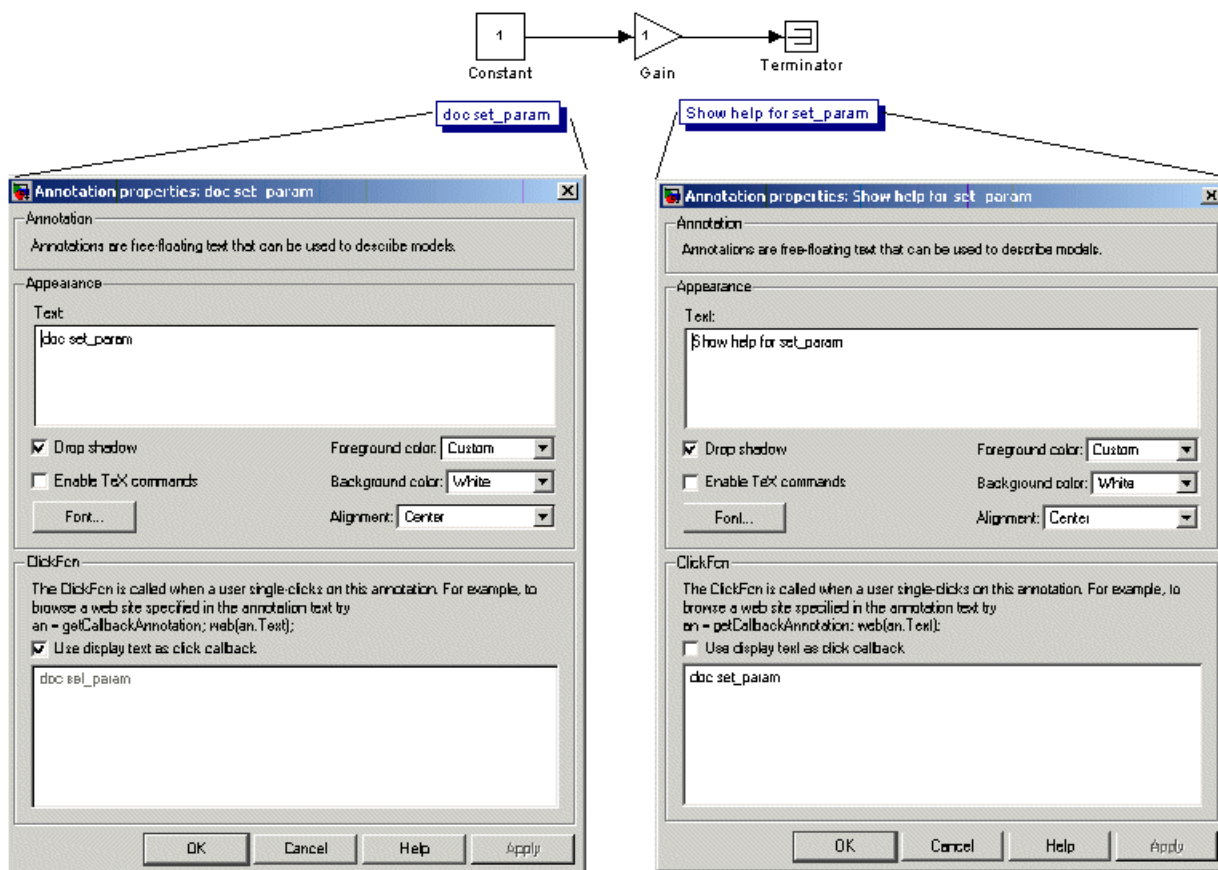
Delete function

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).

Associating Click Functions with Annotations

Two ways are provided to associate a click function with an annotation via the annotation's properties dialog box (see "Annotations Properties Dialog Box" on page 4-27). You can specify either the annotation itself as the click function or a separately defined function. To specify the annotation itself as the click function, enter a valid MATLAB expression in the dialog box's **Text** field and check the **Use display text** as callback option. To specify a separately defined click function, enter the M-code that defines the click function in the dialog box's **ClickFcn** edit field.

The following model illustrates the two ways to associate click functions with an annotation.



Annotation text as the click function

Separately defined click function

Clicking either of the annotations in this model displays help for the `set_param` command.

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

Selecting and Editing Annotations Associated with Click Functions

Associating an annotation with a click function prevents you from selecting the annotation by clicking on it. You must use drag select the annotation. Similarly, you cannot make the annotation editable on the diagram by clicking its text. To make the annotation editable on the diagram, first drag-select it, then select **Edit Annotation Text** from model editor’s **Edit** or the context menu.

Annotations API

An application program interface (API) is provided that enables you to use M programs to get and set the properties of annotations. The API comprises the following elements:

- `Simulink.Annotation` class
Allows M-code, e.g., annotation load functions (see “Load Function” on page 4-30), to set the properties of annotations
- `getCallbackAnnotation` function
Gets the `Simulink.Annotation` object for the annotation associated with the currently executing annotation callback function

Using TeX Formatting Commands in Annotations

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

Linearization of Double Pendulum

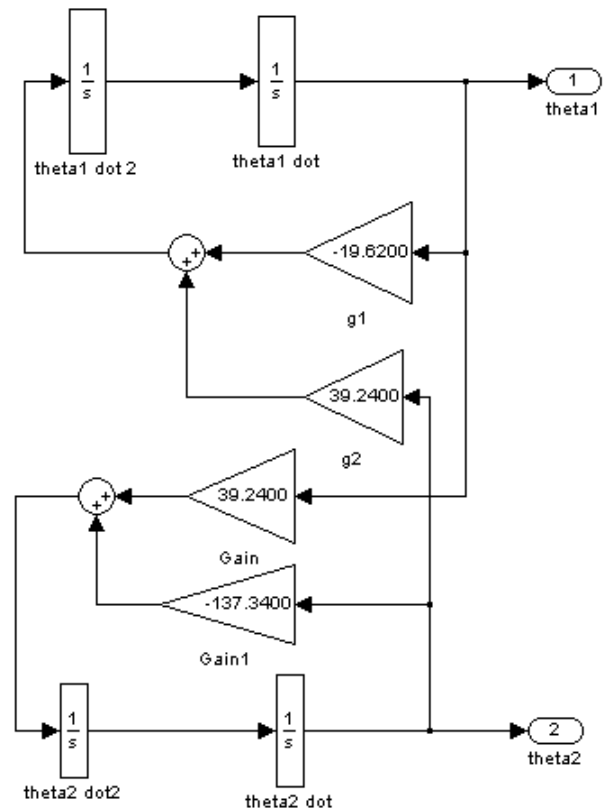
$$\theta_1'' = -19.6200\theta_1 + 39.2400\theta_2$$

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

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 **Enable TeX Commands** from model editor's **Format** menu.
- 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 on the TeX formatting commands which are supported.

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

The formatted text is displayed.

```

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

```

Creating Annotations Programmatically

You can use the `add_block` command to create annotations at the command line or in an M-file program. Use the following syntax to create the annotation:

```

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, the following sequence of commands

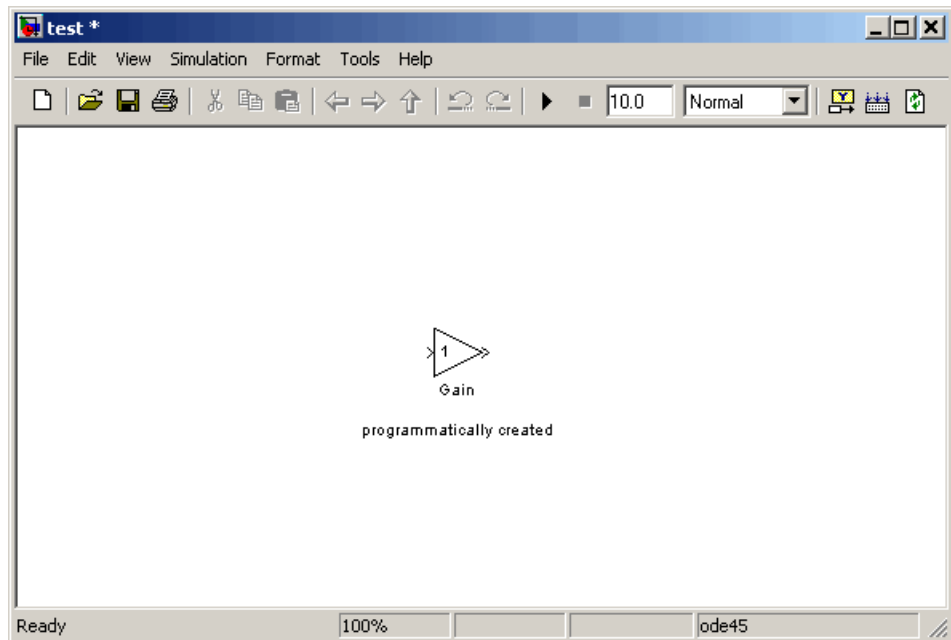
```

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])
```

creates the following model:



To delete an annotation, use the `find_system` command to get the annotation's handle. Then use the `delete` function to delete the annotation, e.g.,

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

Creating Subsystems

In this section...

- “Why Subsystems are Advantageous” on page 4-37
- “Creating a Subsystem by Adding the Subsystem Block” on page 4-38
- “Creating a Subsystem by Grouping Existing Blocks” on page 4-38
- “Model Navigation Commands” on page 4-40
- “Window Reuse” on page 4-40
- “Labeling Subsystem Ports” on page 4-41
- “Controlling Access to Subsystems” on page 4-42
- “Interconverting Subsystems and Block Diagrams” on page 4-43
- “Emptying Subsystems and Block Diagrams” on page 4-43

Why Subsystems are Advantageous

A subsystem is a set of blocks that have been replaced by 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:

- It helps reduce the number of blocks displayed in your model window.
- It allows you to keep functionally related blocks together.
- It enables you to establish a hierarchical block diagram, where a Subsystem block is on one layer and the blocks that make up the subsystem are on another.

You can create a subsystem in two ways:

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

A subsystem can be executed 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 information about conditionally executed subsystems, see Chapter 5, “Creating Conditional Subsystems”.

Creating a Subsystem by Adding the Subsystem Block

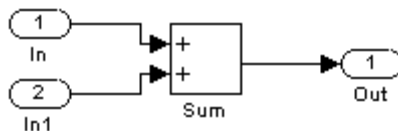
To create a subsystem before adding the blocks it contains, add a Subsystem block to the model, then add the blocks that make up the subsystem:

- 1 Copy the Subsystem block from the Ports & Subsystems library into your model.
- 2 Open the Subsystem block by double-clicking it.

The subsystem is opened in the current or a new model window, depending on the model window reuse mode that you selected (see “Window Reuse” on page 4-40).

- 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, the subsystem shown includes a Sum block and Inport and Outport blocks to represent input to and output from the subsystem.



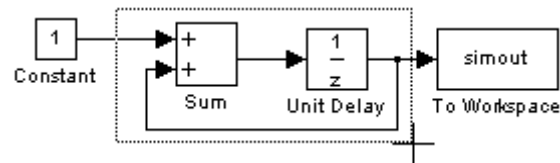
Creating a Subsystem by Grouping Existing Blocks

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

- 1 Enclose the blocks and connecting lines that you want to include in the subsystem within a bounding box. You cannot specify the blocks to be grouped by selecting them individually or by using the **Select All**

command. For more information, see “Selecting Multiple Objects Using a Bounding Box” on page 4-7.

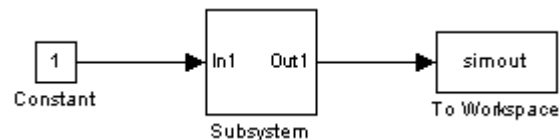
For example, this figure shows a model that represents a counter. The Sum and Unit Delay blocks are selected within a bounding box.



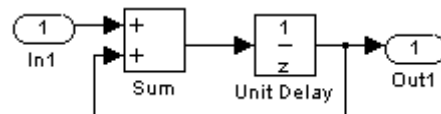
When you release the mouse button, the two blocks and all the connecting lines are selected.

- 2 Choose **Create Subsystem** from the **Edit** menu. The selected blocks are replaced with a Subsystem block.

This figure shows the model after you choose the **Create Subsystem** command (and resize the Subsystem block so the port labels are readable).



If you open the Subsystem block, the underlying system is displayed, as shown below.



Notice that the Simulink software 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. You can also use the masking feature to customize the block's appearance and dialog box. See Chapter 9, “Working with Block Masks”.

Undoing Subsystem Creation

To undo creation of a subsystem by grouping blocks, select **Undo** from the **Edit** menu. You can undo creation of a subsystem that you have subsequently edited. However, the **Undo** command does not undo any nongraphical changes that you made to the blocks, such as changing the value of a block parameter or the name of a block. The Simulink software alerts you to this limitation by displaying a warning dialog box before undoing creation of a modified subsystem.

Model Navigation Commands

Subsystems allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy using the Model Browser (see “The Model Browser” on page 19-26) and/or the following model navigation commands:

- **Open Block**

The **Open Block** command opens the currently selected subsystem. To execute the command, select **Open Block** from either the **Edit** menu or the subsystem’s context (right-click) menu, press **Enter**, or double-click the subsystem.

- **Open Block In New Window**

Opens the currently selected subsystem regardless of the window reuse settings (see “Window Reuse” on page 4-40). To execute the command, select **Open Block In New Window** from the subsystem’s context (right-click) menu.

- **Go To Parent**

The **Go To Parent** command displays the parent of the subsystem displayed in the current window. To execute the command, press **Esc** or select **Go To Parent** from the Simulink software **View** menu.

Window Reuse

You can specify whether the Simulink software model navigation commands use the current window or a new window to display a subsystem or its parent. Reusing windows avoids cluttering your screen with windows. Creating a window for each subsystem allows you to view subsystems side by side with their parents or siblings. To specify your preference regarding window reuse,

select **Preferences** from the **File** menu and then select one of the following **Window reuse type** options listed in the **Preferences** dialog box.

Reuse Type	Open Action	Go to Parent (Esc) Action
none	Subsystem appears in a new window.	Parent window moves to the front.
reuse	Subsystem replaces the parent in the current window.	Parent window replaces subsystem in current window
replace	Subsystem appears in a new window. Parent window disappears.	Parent window appears. Subsystem window disappears.
mixed	Subsystem appears in its own window.	Parent window rises to front. Subsystem window disappears.

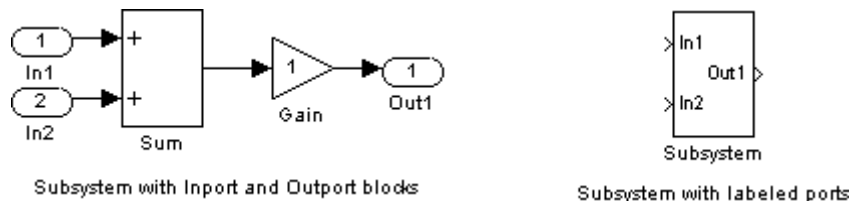
Labeling Subsystem Ports

Simulink labels ports on a Subsystem block. The labels are the names of Inport and Outport blocks that connect the subsystem to blocks outside the subsystem through these ports.

You can hide (or show) the port labels by

- Selecting the Subsystem block, then choosing **Hide Port Labels** (or **Show Port Labels**) from the **Format** menu
- Selecting an Inport or Outport block in the subsystem and choosing **Hide Name** (or **Show Name**) from the **Format** menu
- Selecting the **Show port labels** option in the Subsystem block's parameter dialog

This figure shows two models.



The subsystem on the left contains two Inport blocks and one Outport block. The Subsystem block on the right shows the labeled ports.

Controlling 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.

To restrict access to a library subsystem, open the subsystem's parameter dialog box and set its **Read/Write permissions** parameter to either `ReadOnly` or `NoReadOrWrite`. The first option allows a user to view the contents of the library subsystem but prevents the user from modifying the reference subsystem without first disabling its library link or changing its **Read/Write permissions** parameter to `ReadWrite`. The second option prevents the user from viewing the contents of the library subsystem, modifying the reference subsystem, and changing the reference subsystem's permissions. Note that both options allow a user to use the library subsystem in models by creating links (see Chapter 8, "Working with Block Libraries"). See the Subsystem block in the *Simulink Reference* guide for more information on subsystem access options.

Note You will 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, the Simulink software neither opens the subsystem nor displays any messages.

Interconverting Subsystems and Block Diagrams

These functions are provided that you can use to interconvert subsystems and block diagrams:

`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.

For more information, see the reference documentation for these functions.

Emptying Subsystems and Block Diagrams

These functions are provided to empty subsystems and block diagrams:

`Simulink.SubSystem.deleteContents`

Deletes the contents of a subsystem.

`Simulink.BlockDiagram.deleteContents`

Deletes the contents of a block diagram.

For more information, see the reference documentation for these functions.

Modeling Control Flow Logic

In this section...

“Equivalent C Language Statements” on page 4-44

“Modeling Conditional Control Flow Logic” on page 4-44

“Modeling While and For Loops” on page 4-47

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

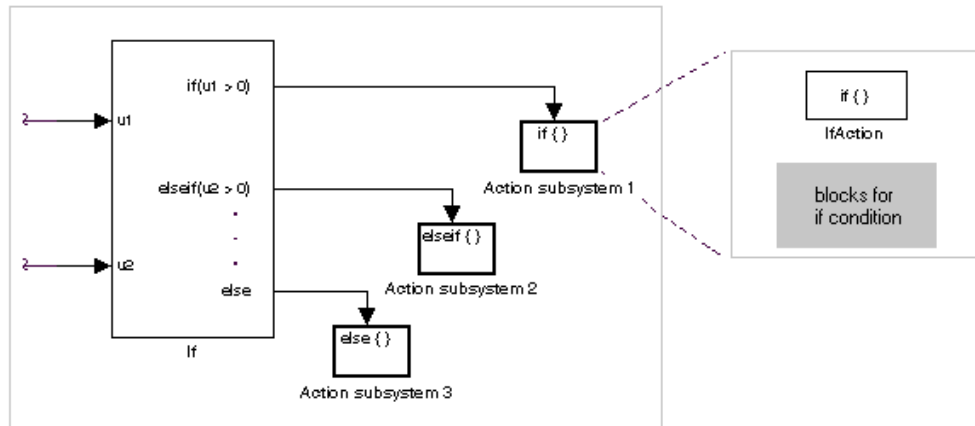
Modeling Conditional Control Flow Logic

You can use the following blocks to model conditional control flow logic.

C Statement	Equivalent Blocks
if-else	If, If Action Subsystem
switch	Switch Case, Switch Case Action Subsystem

Modeling If-Else Control Flow

The following diagram models if-else control flow.



Construct an if-else control flow diagram as follows:

- Provide data inputs to the If block for constructing if-else conditions.

Inputs to the If block are set in the If block properties dialog box. Internally, they are designated as u_1 , u_2 , \dots , u_n and are used to construct output conditions.

- Set output port if-else conditions for the If block.

Output ports for the If block are also set in its properties dialog box. You use the input values u_1 , u_2 , \dots , 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.

- Connect each condition output port to an Action subsystem.

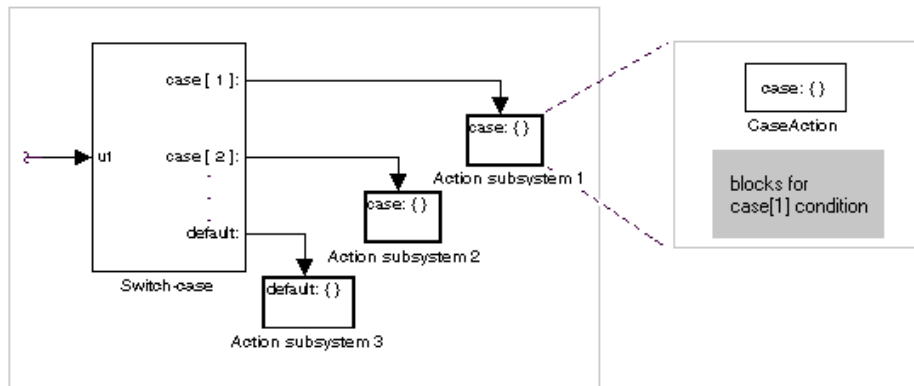
Each if, elseif, and else condition output port on the If block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing an Action Port block in a subsystem. This creates an atomic Action subsystem with a port named Action, which you then connect to 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.

Modeling Switch Control Flow

The following diagram models switch control flow.



Construct a switch control flow statement as follows:

- 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.

- Add cases to the Switch Case block based on the numeric value of the argument input.

You add cases to the Switch Case block through the properties dialog box of 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.

- 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 *Simulink Reference* for the Switch Case and Action Port blocks.

Note After the subsystem for a particular case is executed, an implied break is executed that exits the switch control flow statement altogether. The Simulink software switch control flow statement implementations do not exhibit “fall through” behavior like C switch statements.

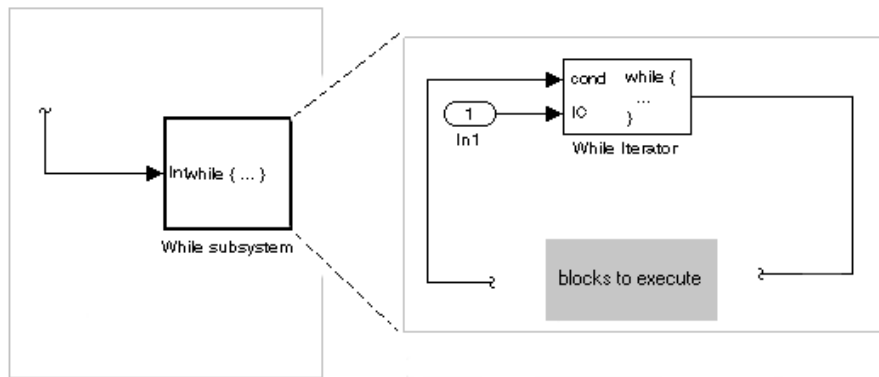
Modeling While and For Loops

The following blocks allow you to model *while* and *for* loops.

C Statement	Equivalent Blocks
do-while	While Iterator Subsystem
for	For Iterator Subsystem
while	While Iterator Subsystem

Modeling While Loops

The following diagram illustrates a *while* loop.



In this example, the Simulink software 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, the Simulink software 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, i.e., a block whose output depends on its previous input, reflects the value of its input at the previous iteration of the *while* loop—not, as one might expect, its 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:

- Place a While Iterator block in a subsystem.

The host subsystem's 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 you want to iterate with the While Iterator block.

- 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.

- 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.

- You can 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.

- You can 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.

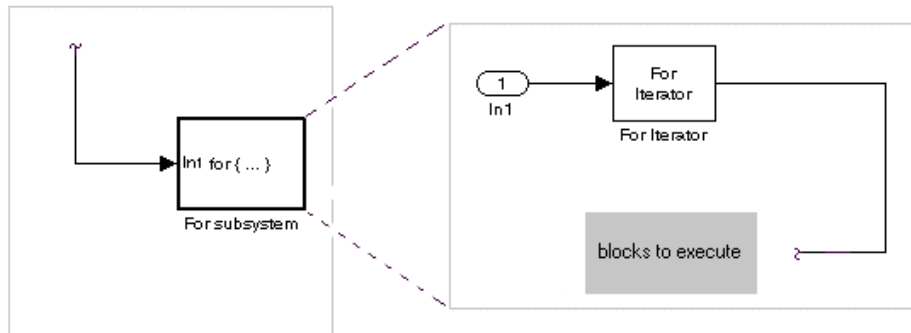
- Create a block diagram in the subsystem that defines the subsystem's outputs.

Note The diagram must not contain blocks with continuous states, e.g., blocks from the Continuous block library, and the sample times of all the blocks must be inherited (-1) or constant (inf).

For more information, see the While Iterator block.

Modeling For Loops

The following diagram models a for loop:



In this example, the Simulink software executes the contents of the For subsystem multiples times at each time step with the number of iterations being specified by the input to the For Iterator block. In particular, for each iteration of the for loop, the Simulink software invokes the update and output methods of all the blocks in the For 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 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, i.e., a block whose output depends on its previous input, reflects the value of its input at the previous iteration of the for loop—not, as one might expect, its 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:

- Drag a For Iterator Subsystem block from the Library Browser or Library window into your model.

- You can 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.

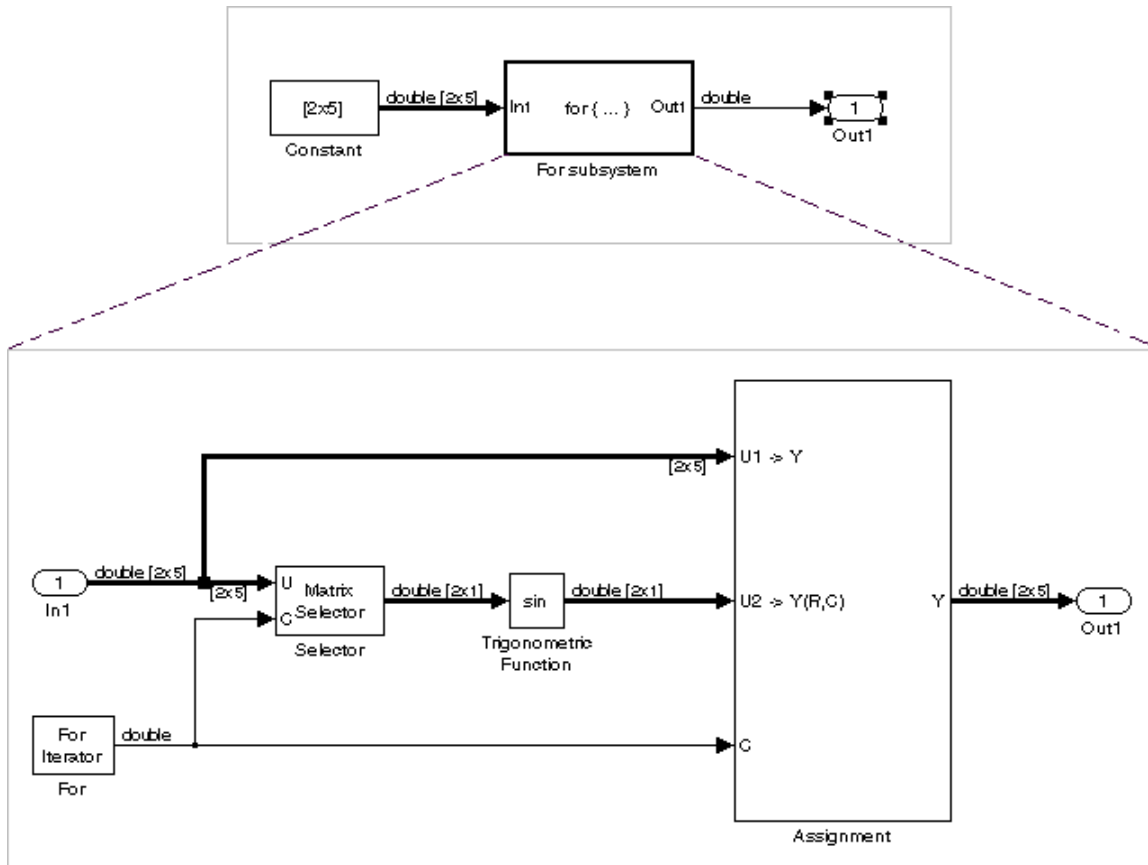
- You can set the For Iterator block to output its iterator value for use in the block programming of the For Iterator Subsystem.

The iterator value is 1 for the first iteration and is incremented by 1 for each succeeding iteration.

- Create a block diagram in the subsystem that defines the subsystem's outputs.

Note The diagram must not contain blocks with continuous states, e.g., blocks from the Continuous block library, and the sample times of all the blocks must be inherited (-1) or constant (inf).

The For Iterator block works well with the Assignment block to reassign values in a vector or matrix. This is demonstrated in the following example. Note the matrix dimensions in the data being passed.



The above example outputs the sine value of an input 2-by-5 matrix (2 rows, 5 columns) using a For subsystem containing an Assignment block. The process is as follows:

- 1** A 2-by-5 matrix is input to the Selector block and the Assignment block.
- 2** The Selector block strips off a 2-by-1 matrix from the input matrix at the column value indicated by the current iteration value of the For Iterator block.
- 3** The sine of the 2-by-1 matrix is taken.

- 4 The sine value 2-by-1 matrix is passed to an Assignment block.
- 5 The Assignment block, which takes the original 2-by-5 matrix as one of its inputs, assigns the 2-by-1 matrix back into the original matrix at the column location indicated by the iteration value.

The rows specified for reassignment in the property dialog for the Assignment block in the above example are [1,2]. Because there are only two rows in the original matrix, you could also have specified -1 for the rows, i.e., all rows.

Note Experienced Simulink software users will note that 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.

Using Callback Functions

In this section...
“About Callback Functions” on page 4-54
“Tracing Callbacks” on page 4-55
“Creating Model Callback Functions” on page 4-55
“Creating Block Callback Functions” on page 4-58
“Port Callback Parameters” on page 4-62
“Example Callback Function Tasks” on page 4-63

About Callback Functions

Callback functions are a powerful way of customizing your Simulink model. A *callback* is a function that 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 an M-file or other MATLAB command. You can use block, port, or model parameters to specify callback functions.

Common tasks you can achieve by using callback functions include:

- Loading variables into the MATLAB workspace automatically when you open your Simulink model
- Executing an M-file by double-clicking on a block
- Executing a series of commands before starting a simulation
- Executing commands when a block diagram is closed

For examples of these tasks, see “Example Callback Function Tasks” on page 4-63.

For related tasks, see

- “Using Model Workspaces” on page 4-66 for loading and modifying variables required by your model
- “Model Dependencies” on page 19-29 for analyzing model and block callbacks, and identifying and packaging files required by your model

Tracing Callbacks

Callback tracing allows you to determine the callbacks the Simulink software invokes and in what order the it invokes them when you open or simulate a model. To enable callback tracing, select the **Callback tracing** option on the Preferences dialog box or execute `set_param(0, 'CallbackTracing', 'on')`. This option causes the callbacks to be listed in the MATLAB Command Window as they are invoked. This option applies to all Simulink models, not just models that are open when the preference is enabled.

Creating Model Callback Functions

You can create model callback functions interactively or programmatically. Use the **Callbacks** pane of the model’s Model Properties dialog box (see “Callbacks Pane” on page 4-104) to create model callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the model parameter corresponding to the callback (see “Model Callback Functions” on page 4-56).

For example, this command evaluates the variable `testvar` when the user double-clicks the Test block in `mymodel`:

```
set_param('mymodel/Test', 'OpenFcn', testvar)
```

You can examine the `clutch` system (`sldemo_clutch.mdl`) for routines associated with many model callbacks. This model defines the following callbacks:

- `PreLoadFcn`
- `PostLoadFcn`
- `StartFcn`
- `StopFcn`

- CloseFcn

Model Callback Functions

The following table describes callback functions associated with models.

Parameter	When Executed
CloseFcn	Before the block diagram is closed. Any ModelCloseFcn and DeleteFcn callbacks set on blocks in the model are called prior to the model's CloseFcn. The DestroyFcn callback of any blocks in the model is called after the model's CloseFcn.
PostLoadFcn	After the model is loaded. Defining a callback routine for this parameter might be useful for generating an interface that requires that the model has already been loaded.
InitFcn	Called at start of model simulation.
PostSaveFcn	After the model is saved.
PreLoadFcn	Before the model is loaded. Defining a callback routine for this parameter might be useful for loading variables used by the model.

Parameter	When Executed
	<hr/> <p>Note In a <code>PreLoadFcn</code> callback routine, the <code>get_param</code> command does not return the model's parameter values because the model is not yet loaded.</p> <p>In a <code>PreLoadFcn</code> routine, <code>get_param</code> returns:</p> <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> <p>In a <code>PostLoadFcn</code> callback routine, however, <code>get_param</code> returns the model's parameter values because the model is loaded.</p> <hr/>
PreSaveFcn	Before the model is saved.
StartFcn	Before the simulation starts.
StopFcn	After the simulation stops. Output is written to workspace variables and files before the <code>StopFcn</code> is executed.

Note Beware of adverse interactions between callback functions of models referenced by other models. (See Chapter 6, “Referencing a Model”.) For example, suppose that model A references model B and that model A's `OpenFcn` creates variables in the MATLAB workspace and model B's `CloseFcn` clears the MATLAB workspace. Now suppose that simulating model A requires rebuilding model B. Rebuilding B entails opening and closing model B and hence invoking model B's `CloseFcn`, which clears the MATLAB workspace, including the variables created by A's `OpenFcn`.

Creating Block Callback Functions

You can create block callback functions interactively or programmatically. Use the **Callbacks** pane of the block's Block Properties dialog box (see "Callbacks Pane" on page 7-31) to create block callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the block parameter corresponding to the callback (see "Block Callback Parameters" on page 4-58).

Note A callback for a masked subsystem cannot directly reference the parameters of the masked subsystem (see). The Simulink software 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, e.g., `get_param(gcb, 'gain')`, where `gain` is the name of a mask parameter of the current block.

Block Callback Parameters

This table lists the parameters for which you can define block callback routines, and indicates when those callback routines are executed. Routines that are executed before or after actions take place occur immediately before or after the action.

Parameter	When Executed
ClipboardFcn	When the block is copied or cut to the system clipboard.
CloseFcn	When the block is closed using the <code>close_system</code> command. The <code>CloseFcn</code> is not called when you interactively close the block, when you interactively close the subsystem or model containing the block, or when you close the subsystem or model containing the block using <code>close_system</code> .

Parameter	When Executed
CopyFcn	After a block is copied. The callback is recursive for Subsystem blocks (that is, if you copy a Subsystem block that contains a block for which the CopyFcn parameter is defined, the routine is also executed). The routine is also executed if an <code>add_block</code> command is used to copy the block.
DeleteChildFcn	After a block or line is deleted in a subsystem. If the block has a DeleteFcn or DestroyFcn, those functions are executed prior to the DeleteChildFcn. Only Subsystem blocks have a DeleteChildFcn callback.
DeleteFcn	After a block is graphically deleted, e.g., when you graphically delete the block, invoke <code>delete_block</code> on the block, or close the model containing the block. When the DeleteFcn is called, the block handle is still valid and can be accessed using <code>get_param</code> . The DeleteFcn callback is recursive for Subsystem blocks. 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 DestroyFcn is called.
DestroyFcn	When the block has been destroyed from memory, e.g., when you invoke <code>delete_block</code> on either the block or a subsystem containing the block or close the model containing the block. If the block was not previously graphically deleted, the block's DeleteFcn is called prior to the DestroyFcn. When the DestroyFcn is called, the block handle is no longer valid.
InitFcn	Before the block diagram is compiled and before block parameters are evaluated.

Parameter	When Executed
ErrorFcn	<p>When an error has occurred in a subsystem. Only Subsystem blocks have a ErrorFcn callback. The callback function should have the following form:</p> <pre data-bbox="718 427 1334 453">errorMsg = errorHandler(subsys, errorType)</pre> <p>where errorHandler is the name of the callback 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, and errorMsg is a string specifying the error message to be displayed to the user. The following command sets the ErrorFcn of the subsystem subsys to call the errorHandler callback function</p> <pre data-bbox="718 782 1348 808">set_param(subsys, 'ErrorFcn', 'errorHandler')</pre> <p>Do not include the callback function's input arguments in the call to set_param. The Simulink software displays the error message errorMsg returned by the callback function.</p>
LoadFcn	<p>After the block diagram is loaded. This callback is recursive for Subsystem blocks.</p>
ModelCloseFcn	<p>Before the block diagram is closed. When the model is closed, the block's ModelCloseFcn is called prior to its DeleteFcn. This callback is recursive for Subsystem blocks.</p>
MoveFcn	<p>When the block is moved or resized.</p>
NameChangeFcn	<p>After a block's name and/or path changes. When a Subsystem block's path is changed, it recursively calls this function for all blocks it contains after calling its own NameChangeFcn routine.</p>

Parameter	When Executed
OpenFcn	When the block is opened. This parameter is generally used with Subsystem blocks. The routine is executed when you double-click the block or when an <code>open_system</code> command is called 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.
ParentCloseFcn	Before closing a subsystem containing the block or when the block is made part of a new subsystem using the <code>new_system</code> command (see <code>new_system</code> in the online Simulink software reference) or the Create Subsystem item in model editor's Edit menu. The <code>ParentCloseFcn</code> of blocks at the root model level is not called when the model is closed.
PostSaveFcn	After the block diagram is saved. This callback is recursive for Subsystem blocks.
PreCopyFcn	Before a block is copied. The callback is recursive for Subsystem blocks (that is, if you copy a Subsystem block that contains a block for which the <code>PreCopyFcn</code> parameter is defined, that routine is also executed). The block's <code>CopyFcn</code> is called after all <code>PreCopyFcn</code> callbacks are executed, unless the <code>PreCopyFcn</code> invokes the <code>error</code> command either explicitly or via a command used in any <code>PreCopyFcn</code> . The <code>PreCopyFcn</code> is also executed if an <code>add_block</code> command is used to copy the block.
PreDeleteFcn	Before a block is graphically deleted, e.g., when the user graphically deletes the block or invokes <code>delete_block</code> on the block. The <code>PreDeleteFcn</code> is not called when the model containing the block is closed. The block's <code>DeleteFcn</code> is called after the <code>PreDeleteFcn</code> unless the <code>PreDeleteFcn</code> invokes the <code>error</code> command either explicitly or via a command used in the <code>PreDeleteFcn</code> .

Parameter	When Executed
PreSaveFcn	Before the block diagram is saved. This callback is recursive for Subsystem blocks.
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. See "S-Function Callback Methods" in the online Simulink software documentation for more information.
StopFcn	At any termination of the simulation. In the case of an S-Function block, StopFcn executes after the block's mdlTerminate function executes. See "S-Function Callback Methods" in the online Simulink software documentation for more information.
UndoDeleteFcn	When a block deletion is undone.

Note Do not call the run command from within model or block callbacks. Doing so can result in unexpected behavior (such as errors or incorrect results) if a Simulink model is loaded, compiled, or simulated from inside an M-function.

Port Callback Parameters

Block input and output ports have a single callback function 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 function must have one input argument that represents the port handle, but 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 fragment sets `foo` as the connection callback on the input port.

```
phs = get_param(gcb, 'PortHandles');  
set_param(phs.Inport, 'ConnectionCallback', 'foo');
```

where, `foo` is defined as:

```
function foo(portHandle)
```

Example Callback Function Tasks

The following sections describe simple examples for commonly used callback routines.

- “Loading Variables Automatically When Opening a Model” on page 4-63
- “Executing an M-file by Double-Clicking a Block” on page 4-64
- “Executing Commands Before Starting Simulation” on page 4-65

Loading Variables Automatically When Opening a Model

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

Some variables might be required by parameters in different parts of the Simulink model. For example, if you have a model which contains a Gain block and the gain is specified as `K`, Simulink looks for the variable `K` to be defined. Using the following technique, you can automatically define `K` every time the model is opened.

You can define variables, such as `K`, in an M-file. You can use the `PreLoadFcn` callback to execute the M-file.

To create model callbacks interactively, open the model’s Model Properties dialog box and use the **Callbacks** pane to edit callbacks (see “Callbacks Pane” on page 4-104).

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

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

where *expression* is a valid MATLAB command or an M-file that exists in your MATLAB search path.

For example, if your model is called *modelname.mdl* and your variables are defined in an M-file 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 will execute. You can see the variables from the *loadvar.m* declared in the MATLAB workspace.

Executing an M-file by Double-Clicking a Block

You can use the `OpenFcn` callback to automatically execute M-files when the user double-clicks a block. M-files can perform many different tasks such as defining variables for a block, making a call to MATLAB which 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** pane to edit callbacks (see "Callbacks Pane" on page 7-31). To create the `OpenFcn` callback programmatically, click the block that you want to add this property to, then enter the following at the MATLAB command prompt:

```
set_param(gcf,'OpenFcn','expression')
```

where *expression* is a valid MATLAB command or an M-file that exists in your MATLAB search path.

The following example shows how to setup the callback to execute an M-file called *myfunction.m* when double clicking a subsystem called *mysubsystem*.

```
set_param('mymodelname/mysubsystem','OpenFcn','myfunction')
```

Executing Commands Before Starting Simulation

You can use the `StartFcn` callback to automatically execute commands before your 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.

Specifically, you can create a simple M-file named `openscopes.m` and save it on your MATLAB search path, as shown in the following example.

```
% 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 have created this M-file, set the `StartFcn` for the model to call the M-file. For example,

```
set_param('mymodelName', 'StartFcn', 'openscopes')
```

Now every time you run the model, all of the Scope blocks should open automatically and be in the forefront.

Using Model Workspaces

In this section...
“About Model Workspaces” on page 4-66
“Simulink.ModelWorkspace Data Object Class” on page 4-67
“Changing Model Workspace Data” on page 4-68
“Model Workspace Dialog Box” on page 4-70

About Model Workspaces

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’s 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 the model’s MDL-file, a MAT-file, or M-code stored in the model file (see “Data source” on page 4-71 for more information).

- You can interactively reload and save MAT-file and M-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 software 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 (Real-Time Workshop 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 (see “Using Model Arguments” on page 6-28 for more information).

Note When resolving references to variables used in a referenced model, the referenced model’s variables 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. (See Chapter 6, “Referencing a Model”.)

Note When you use a workspace variable as a block parameter, the Simulink software 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
- You have a model 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 this class describes a model workspace. Simulink software 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
- Saving the workspace to the MAT-file that the workspace designates as its data source

For more information, see the reference page for the `Simulink.ModelWorkspace` data object class.

Changing Model Workspace Data

The procedure for modifying a workspace depends on the workspace's data source.

Changing Workspace Data Whose Source Is the Model File

If a model workspace's data source is data stored in the model, you can use Model Explorer (see “The Model Explorer” on page 19-2) or MATLAB commands to change the model's workspace (see “Using MATLAB Commands to Change Workspace Data” on page 4-69).

For example, to create a variable in a model workspace, using Model Explorer, first select the workspace in Model Explorer's **Model Hierarchy** pane. Then select **MATLAB Variable** from Model Explorer's **Add** menu or toolbar. You can similarly use the **Add** menu or Model Explorer's toolbar to add a `Simulink.Parameter` object to a model workspace.

To change the value of a model workspace variable, select the workspace, then select the variable in Model Explorer's Contents pane and edit the value displayed in the Contents pane or in Model Explorer's object Dialog pane. To delete a model workspace variable, select the variable in the Contents pane

and select **Delete** from Model Explorer's **Edit** menu or toolbar. To save the changes, save the model.

Changing Workspace Data Whose Source Is a MAT-File

You can also use Model Explorer or MATLAB commands to modify workspace data whose source is a MAT-file. In this case, if you want to make the changes permanent, you must save the changes to the MAT-file, using the **Save To Source** button on the Model Workspace dialog box (see “Model Workspace Dialog Box” on page 4-70). To discard changes to the workspace, use the **Reinitialize From Source** button on the Model Workspace dialog box.

Changing Workspace Data Whose Source Is M-Code

The safest way to change data whose source is M-code is to edit and reload the source, i.e., edit the M-code and then clear the workspace and reexecute the code, using the **Reinitialize From Source** button on the Model Workspace dialog box. You can use the **Export to MAT-File** and **Import From MAT-file** buttons to save and reload alternative versions of the workspace that result from editing the M code source or the workspace variables themselves.

Using 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');
```

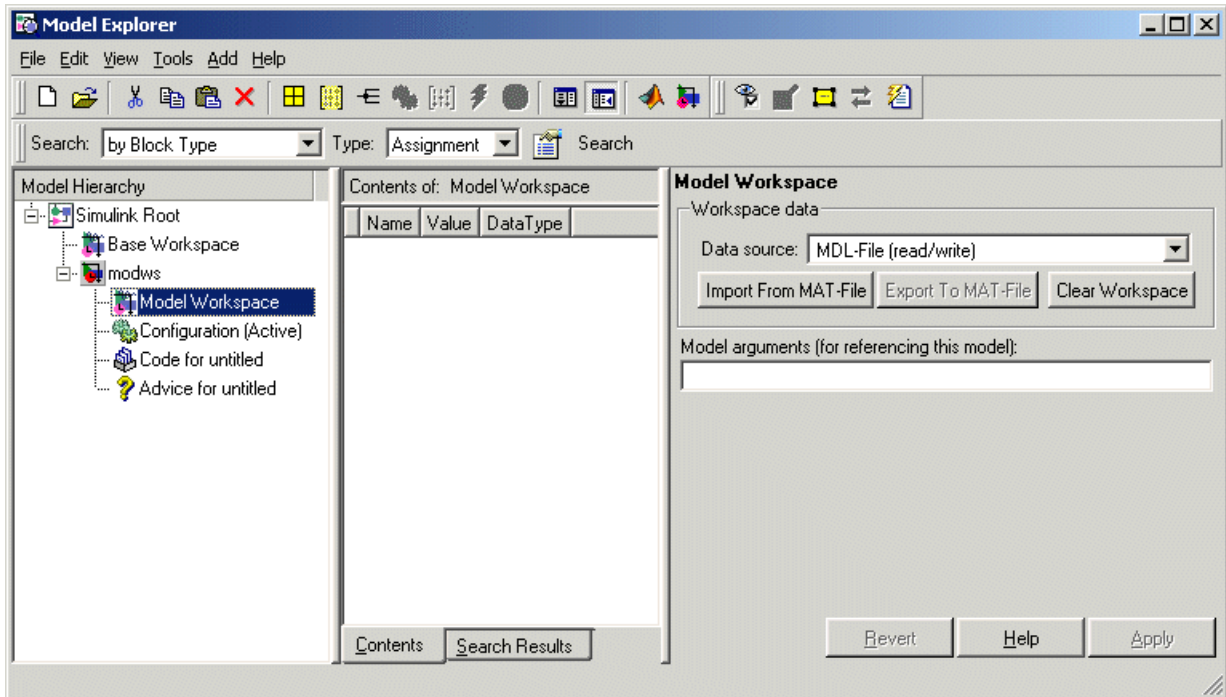
This command returns a handle to a `Simulink.ModelWorkspace` object whose properties specify the source of the data used to initialize the model workspace. Edit the properties to change the data source. Use the workspace's methods to list, set, and clear variables, evaluate expressions in, and save and reload the workspace.

For example, the following MATLAB sequence of commands creates variables specifying model parameters in the model's 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;
```

Model Workspace Dialog Box

The Model Workspace dialog box enables you to specify a model workspace's source and model reference arguments (See Chapter 6, “Referencing a Model”.) To display the dialog box, select the model workspace in Model Explorer's Model Hierarchy pane. To use MATLAB commands to change data in a model workspace, see “Using MATLAB Commands to Change Workspace Data” on page 4-69.



The dialog box contains the following controls.

Data source

Specifies the source of this workspace's data. The options are

- **MDL-File**

Specifies that the data source is the model itself. Selecting this option causes additional controls to appear (see "MDL-File Source Controls" on page 4-72).

- **MAT-File**

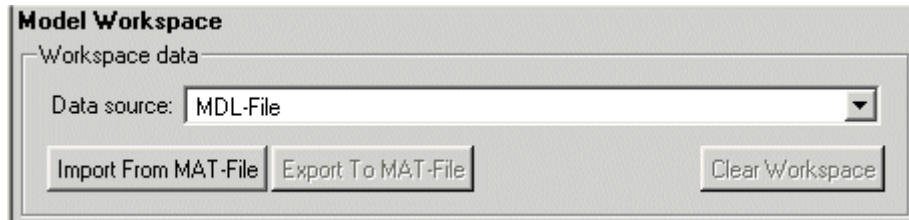
Specifies that the data source is a MAT file. Selecting this option causes additional controls to appear (see "MAT-File Source Controls" on page 4-72).

- **M-code**

Specifies that the data source is M code stored in the model file. Selecting this option causes additional controls to appear (see “M-Code Source Controls” on page 4-73).

MDL-File Source Controls

Selecting Md1-File as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.



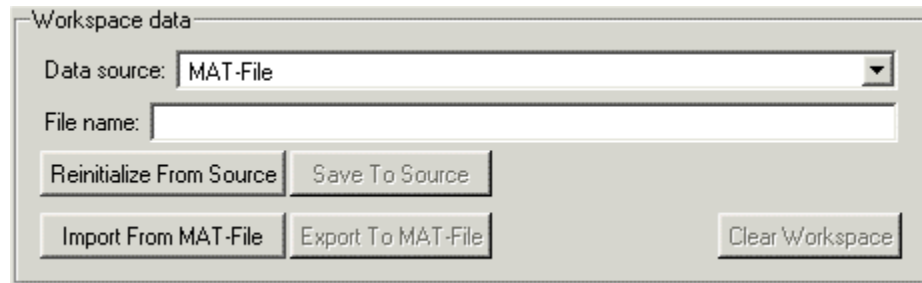
Import From MAT-File. This button lets you import data from a MAT-file. Selecting the button causes a file selection dialog box to be displayed. Use the dialog box to select the MAT file that contains the data you want to import.

Export To MAT-File. This button lets you save the selected workspace as a MAT-file. Selecting the button displays a file selection dialog box. Use the dialog box to select the MAT file to contain the saved data.

Clear Workspace. This button clears all data from the selected workspace.

MAT-File Source Controls

Selecting MAT-File as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.



File name. File name or path name of the MAT file that is the data source for the selected workspace. If 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 specified by the **File name** field.

Save To Source. Save the workspace in the MAT-file specified by the File name field.

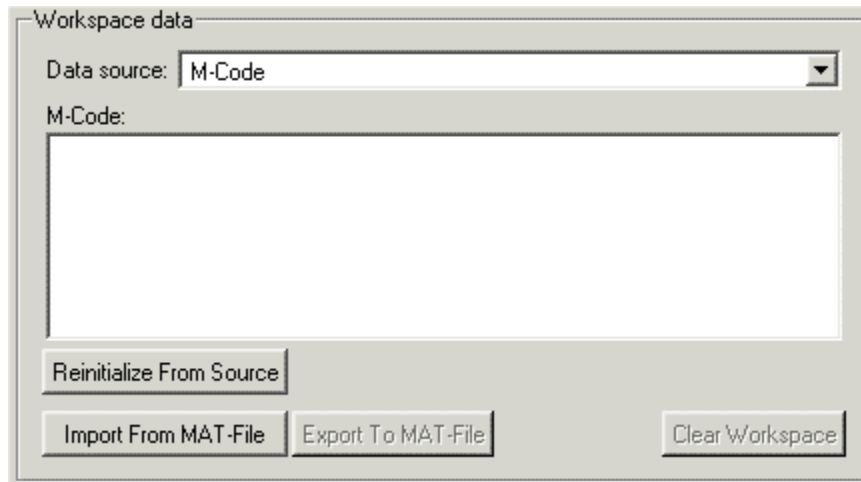
Import From MAT-File. Loads data from a specified MAT file into the selected model workspace without first clearing the workspace. Selecting this option causes a file selection dialog box to be displayed. Use the dialog box to enter the name of the MAT-file that contains the data to be imported.

Export To MAT-File. Saves the data in the selected workspace in a MAT-file. Selecting the button causes a file selection dialog box to be displayed. Use the dialog box to select the MAT file to contain the saved data.

Clear Workspace. Clears the selected workspace.

M-Code Source Controls

Selecting M-Code as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.



M-Code. Specifies M-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 **M-Code** field.

Import From MAT-File. Loads data from a specified MAT file into the selected model workspace without first clearing the workspace. Selecting this option causes a file selection dialog box to be displayed. Use the dialog box to enter the name of the MAT-file that contains the data to be imported.

Export To MAT-File. Saves the data in the selected workspace in a MAT-file. Selecting the button causes a file selection dialog box to be displayed. Use the dialog box to select the MAT file to contain the saved data.

Clear Workspace. Clears the selected workspace.

Model Arguments

This field allows you to specify arguments that can be passed to instances of this model referenced by another model. See Chapter 6, “Referencing a Model” and “Using Model Arguments” on page 6-28 for more information.

Resolving Symbols

In this section...

- “About Symbol Resolution” on page 4-75
- “Hierarchical Symbol Resolution” on page 4-76
- “Specifying Numeric Values with Symbols” on page 4-77
- “Specifying Other Values with Symbols” on page 4-77
- “Limiting Signal Resolution” on page 4-78
- “Explicit and Implicit Symbol Resolution” on page 4-78
- “Programmatic Symbol Resolution” on page 4-79

About Symbol Resolution

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 can be defined 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

The process of searching for and finding an item that corresponds to a symbol is called *resolving* the symbol. The matching item can provide the needed information directly, or it can itself be a symbol, which must then 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 M-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.

Hierarchical Symbol Resolution

The Simulink software 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 the following:

- 1 Any mask workspaces, in order from the block upwards (see “Defining Mask Parameters” on page 9-18)
- 2 The model workspace of the model that contains the block (see “Using Model Workspaces” on page 4-66)
- 3 The MATLAB base workspace (See “MATLAB Workspace”)

If the Simulink software 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, the Simulink software 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 will resolve to the

same item irrespective of whether the model that contains the symbol is a referenced model. See Chapter 6, “Referencing a Model” for information about model referencing.

Specifying Numeric Values with Symbols

Any block parameter that requires a numeric value can be specified 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.

Specifying 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’s **Bus object** parameter can name a bus object (`Simulink.Bus`) that defines the properties of the bus. Symbols can be used when defining data types, can specify input data sources and logged data destinations, and can serve many other purposes.

From the standpoint of 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.

Limiting 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 4-78 for more information.

If the search does not find a match in the workspace, and terminates because the value is `ExplicitOnly` or `None`, the Simulink software 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 Dialog Box” 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. See “Diagnostics Pane: Data Validity” > “Signal resolution” for more information.

Resolution that occurs because an option like **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*.

The MathWorks discourages using implicit symbol resolution except for fast prototyping, because implicit resolution slows performance, complicates model validation, and can have nondeterministic effects.

Programmatic Symbol Resolution

When you use the `sim` command to run a simulation programmatically, you have an option that does not exist with interactive simulation: you can specify a workspace other than the MATLAB base workspace as the last workspace searched in hierarchical symbol resolution.

Most simulation is interactive, so most Simulink documentation does not mention this possibility. For information about substituting another workspace for the base workspace during programmatic simulation, see the `sim` command reference page.

Consulting the Model Advisor

In this section...
“About the Model Advisor” on page 4-80
“Starting the Model Advisor” on page 4-80
“Overview of the Model Advisor Window” on page 4-82
“Running Model Advisor Checks” on page 4-84
“Fixing a Warning or Failure” on page 4-87
“Reverting Changes Using Restore Points” on page 4-92
“Viewing and Saving Model Advisor Reports” on page 4-94
“Running the Model Advisor Programmatically” on page 4-97

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 Real-Time Workshop 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 “Getting Advice About Optimizing Models for Code Generation” in the Real-Time Workshop 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 “Model Advisor Checks” in the Simulink, Real-Time Workshop, and Simulink Verification and Validation Reference documentation.

Starting the Model Advisor

You can use any of the following methods to start the Model Advisor.

Note Before starting the Model Advisor, ensure that the current directory is writable. If the directory is not writable, you see an error message when you start the Model Advisor.

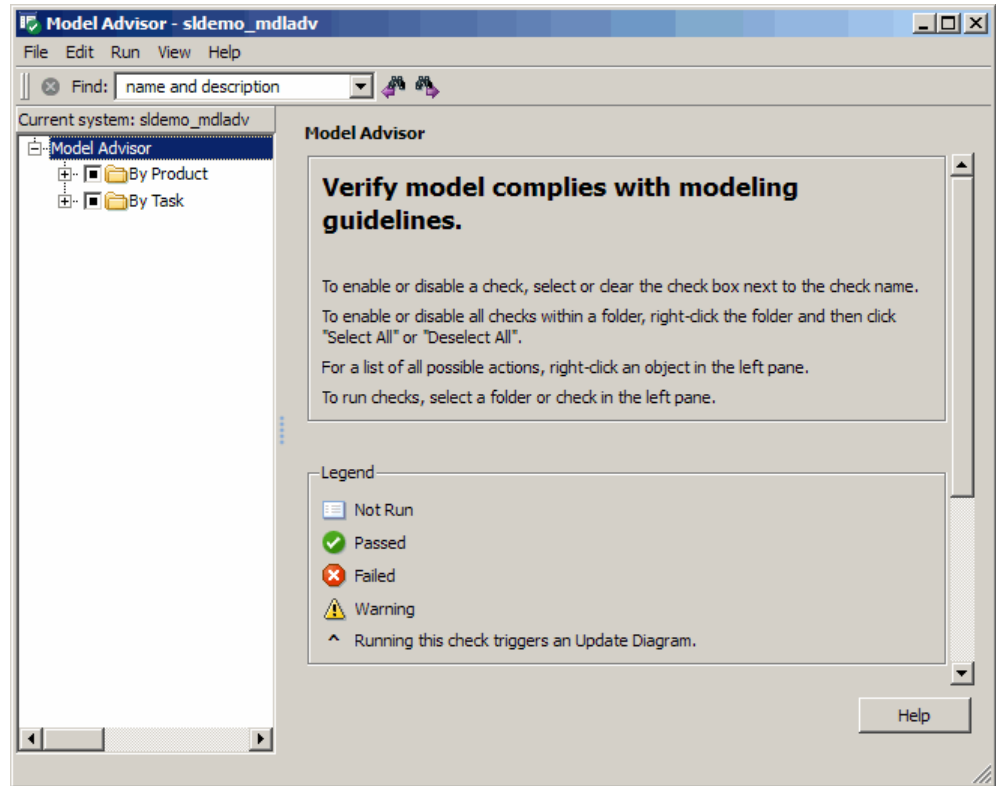
The Model Advisor uses the Simulink project (slprj) directory (for more information, see “Model Reference Simulation Targets” on page 6-17) in the current directory to store reports and other information. If this directory does not exist in the current directory, the Model Advisor creates it.

To start the Model Advisor...	For a...	Do this:
From the Model Editor	Model or subsystem	<p>1 From the Tools menu, select Model Advisor.</p> <p>The System Selector window opens.</p> <p>2 Select the model or subsystem of interest.</p> <p>3 Click OK.</p>
From the Model Explorer	Model	<p>In the Contents pane, select Advice for <i>model</i>, where <i>model</i> is the name of the model that you want to check. (For more information, see “The Model Explorer” on page 19-2.)</p>

To start the Model Advisor...	For a...	Do this:
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> , where <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, and provides a legend explaining the displayed symbols.




From the left pane, 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 associated with the checks, and then run an individual check or all selected checks.
- Reset the status of the checks to not run by right-clicking **Model Advisor** in the right pane and selecting **Reset** from the context menu.
- Specify input parameters for some checks to run (for an example, see “Check for proper Merge block usage” in the **Product > Simulink** folder).

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, which you can optionally view in a separate browser window by clicking the **Report** link at the folder level.

Note When you open the Model Advisor on a model that you have previously checked, the Model Advisor initially displays the check results generated the last time you checked the model. If you recheck the model, the new results replace the previous results in the Model Advisor window.

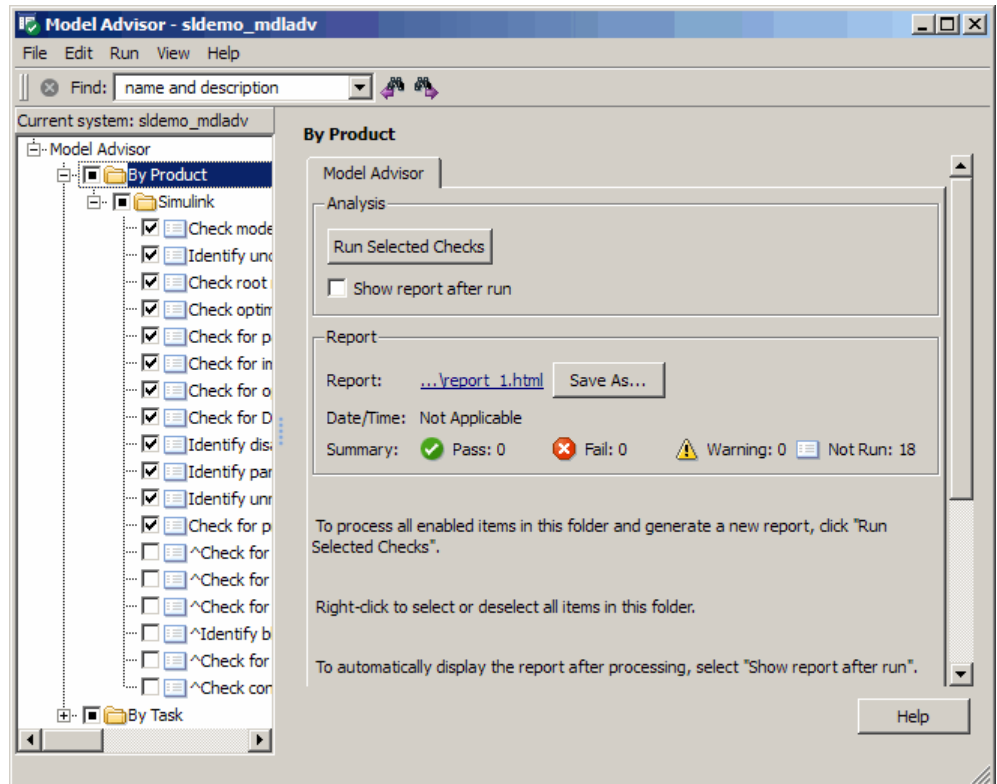
To find checks and folders, enter text in **Find** and click the **Find Next** button (). The Model Advisor searches in check names, folder names, and analysis descriptions for the text.

Running Model Advisor Checks

To use the Model Advisor to perform checks on your model and view the check results:

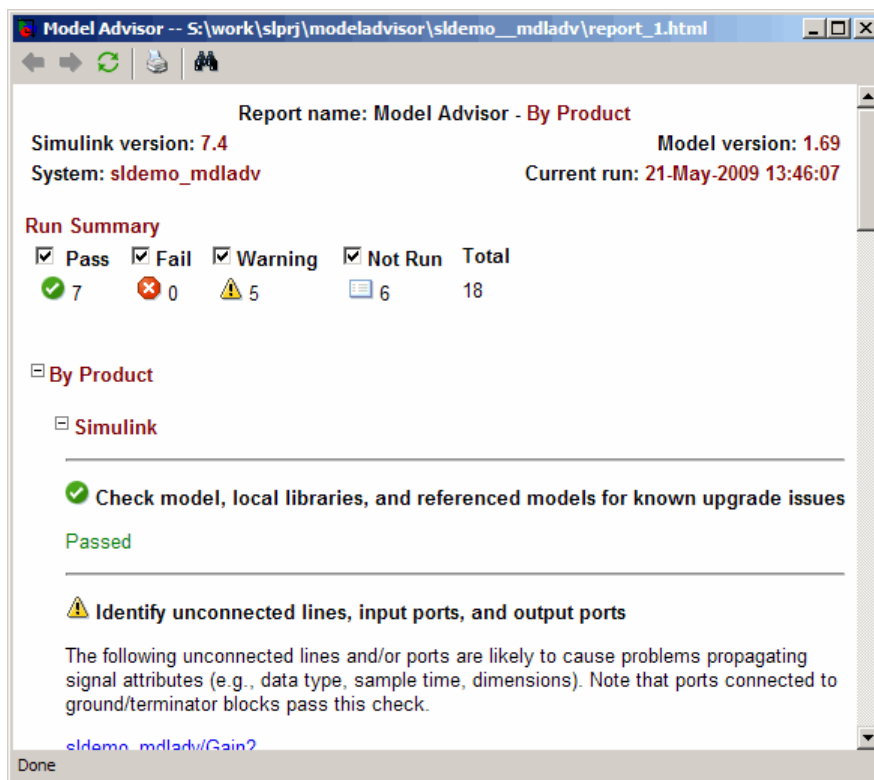
- 1** Open your model. For example, open the Model Advisor demo: `sldemo_mdladv`.
- 2** Start the Model Advisor.
 - a** From the Model Editor **Tools** menu, select **Model Advisor** .
The System Selector window opens.
 - b** In the System Selector window, select the model or system that you want to review. For example, `sldemo_mdladv` and click **OK**.
The Model Advisor window opens and displays checks for the `sldemo_mdladv` demo model.
- 3** In the left pane, expand the **By Product** folder to display the subfolders.
- 4** In the left pane, expand the **Simulink** folder to display the available checks.

- 5 Select the **By Product** folder in the left pane. The right pane changes to a **By Product** view.

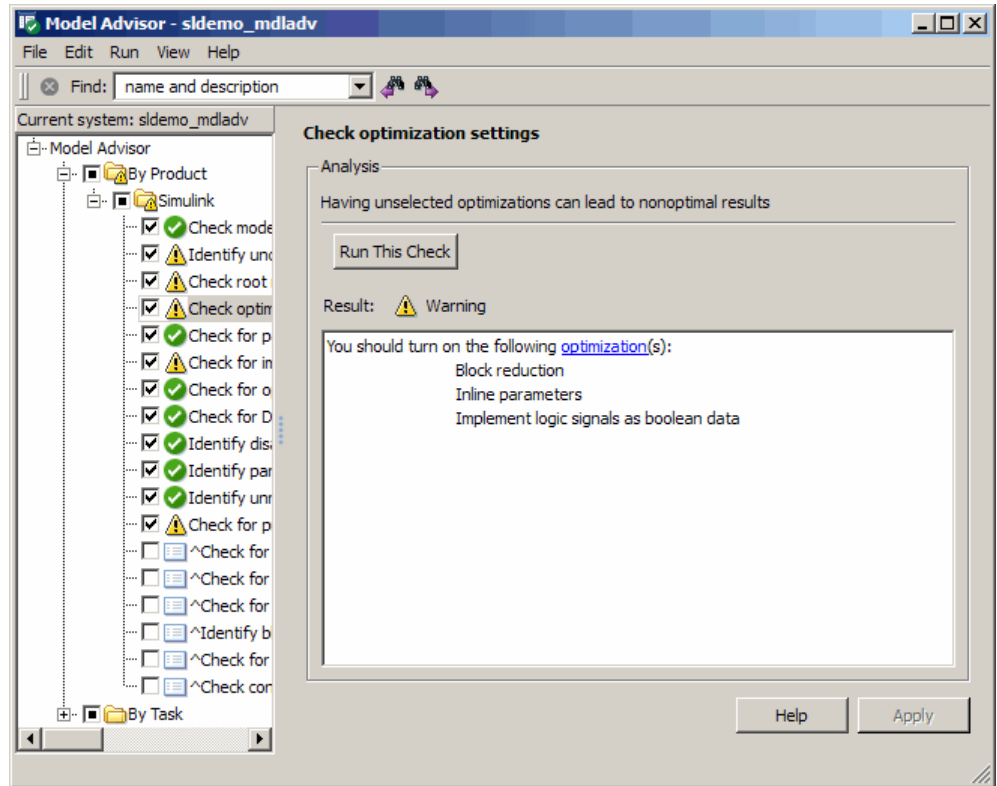


- 6 Select the **Show report after run** check box. This option causes an HTML report of check results to appear after you run the checks.
- 7 Run the selected checks by clicking the **Run Selected Checks** button. After the Model Advisor runs the checks, an HTML report displays the check results in a browser window.

Tip While you can fix warnings and failures through Model Advisor reports, use the Model Advisor window for interactive fixing. Model Advisor reports are best for viewing a summary of checks.



- 8 Return to the Model Advisor window, which shows the check results.
- 9 Select an individual check to open a detailed view of the results in the right pane. For example, selecting **Check optimization settings** changes the right pane to the following view. Use this view to examine and exercise a check individually.



- 10 After reviewing the check results, you can choose to fix warnings or failures as described in “Fixing a Warning or Failure” on page 4-87.

Fixing 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 “Simulink Checks” documentation.

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.

The Model Advisor provides the following ways to fix warnings and failures:

- Follow the instructions in the Analysis Result box to manually fix any warning or failure. See “Manually Fixing Warnings or Failures” on page 4-88.
- Use the Action box, when available, to automatically fix all failures. See “Automatically Fixing Warnings or Failures” on page 4-89.
- Use the Model Advisor Results Explorer, when available, to batch-fix failures. See “Batch-Fixing Warnings or Failures” on page 4-90.

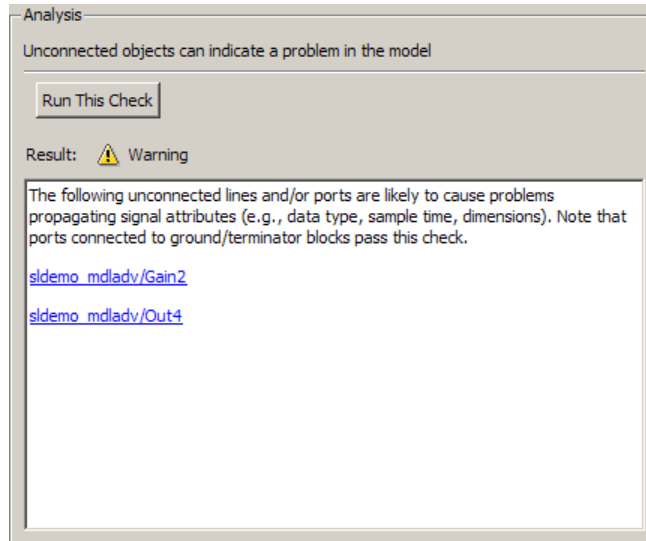
Manually Fixing Warnings or Failures

All 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 “Reverting Changes Using Restore Points” on page 4-92.

- 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 Fixing 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 Review the Analysis Result box before automatically fixing failures to ensure 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 “Reverting Changes Using Restore Points” on page 4-92.

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-Fixing 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, a version of the Model Explorer, helps you to modify only the items that the Model Advisor is checking. For more information about using this window, see “The Model Explorer” on page 19-2.

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 any changes that you make. For more information, see “Reverting Changes Using Restore Points” on page 4-92.

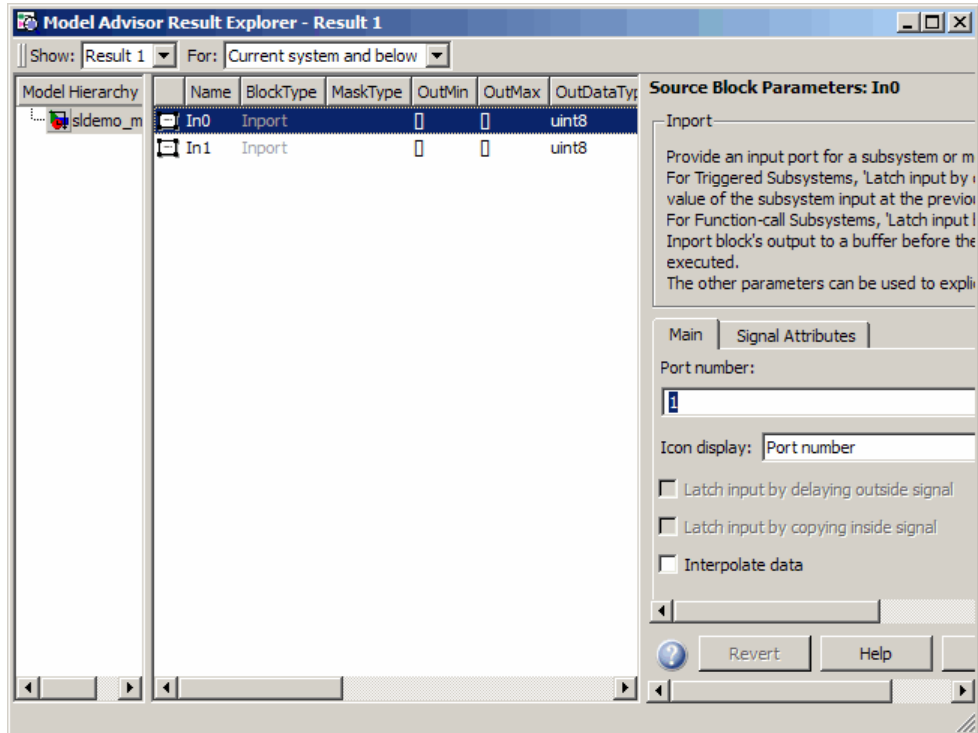
- 2** In the Analysis box, click **Explore Result**.

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_mdldv` model. The result is a warning. Clicking the **Explore Result** button opens the Model Advisor Result Explorer window.



Reverting 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.

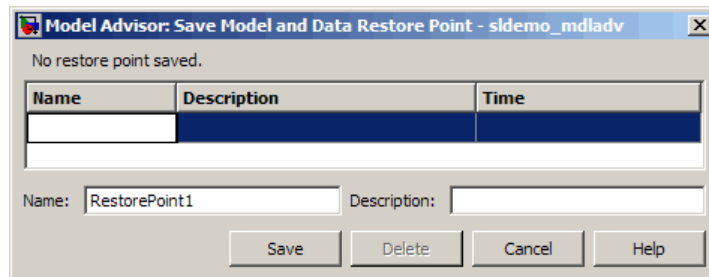
Saving a Restore Point

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 with a name and optional description:

- 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.

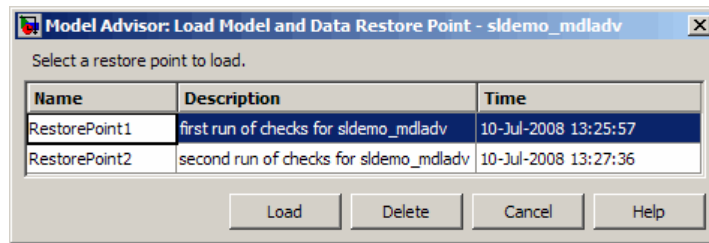
To quickly save a restore point, go to **File > Save Restore Point**. The Model Advisor saves a restore point with the name `autosaven`, where n is the sequential number of the restore point. If you use this method, you cannot change the name of, or add a description to, the restore point.

Loading a Restore Point

To load a restore point:

- 1 Optionally, save a model and data restore point so you can undo any 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 all changes that you made after saving the restore point.

- 5 Click **Load** to load the restore point you selected.

The Model Advisor reverts the model, base workspace, and Model Advisor status.

Viewing and Saving Model Advisor Reports

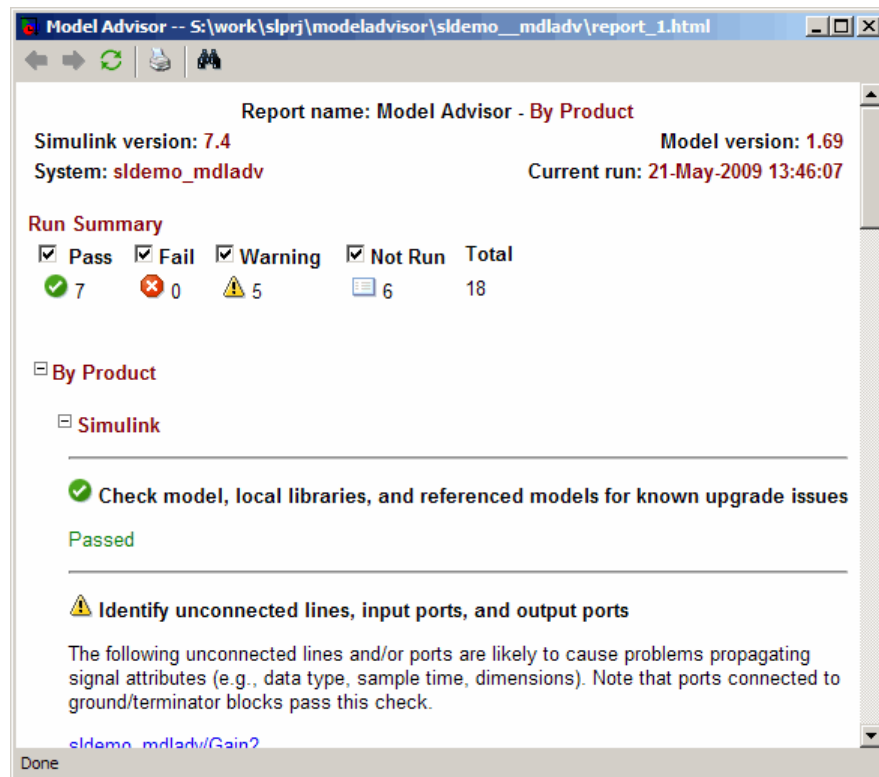
When the Model Advisor runs checks, it generates an HTML report of check results. Each folder in the Model Advisor contains a report for all of the checks in that folder and the subfolders within that folder.

Viewing Model Advisor Reports

You can access any report by selecting a folder and clicking the link in the **Report** box.

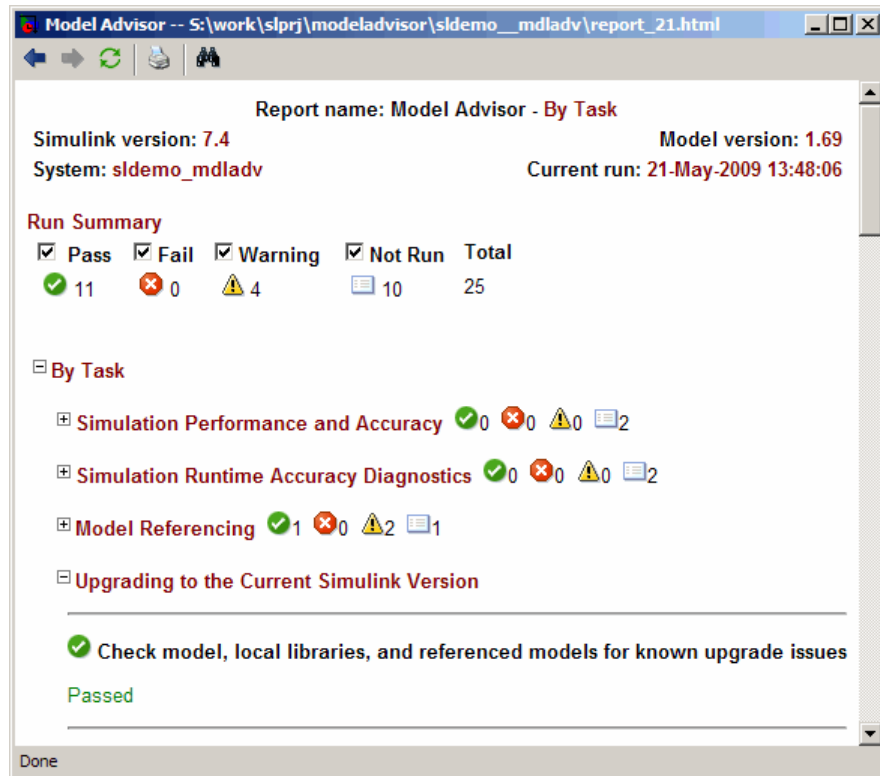
Tip While you can fix warnings and failures through Model Advisor reports, use the Model Advisor window for interactive fixing. Model Advisor reports are best for viewing a summary of checks.

As you run checks, the Model Advisor updates the reports with the latest information for each check in the folder. A message appears in the report when you run the checks at different times. 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.



You can manipulate the report to show only what you are interested in viewing as follows:

- The check boxes next to the Run Summary status allow you to view only the checks with the status that you are interested in viewing. 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 updates to display a run summary for that folder:



Saving Model Advisor Reports

You can archive a Model Advisor report by saving it to a new location. To save a report:

- 1 In the Model Advisor window, navigate to the folder that contains the report 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 directory, 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 directory: `s1prj/modeladvisor/model_name`.

Running the Model Advisor Programmatically

You can create M-file programs that run the Model Advisor programmatically. For example, you can create an M-file program to check that your model passes a specified set of the Model Advisor checks every time you open the model or start a simulation or generate code from the model. For more information, see the `Simulink.ModelAdvisor` class in the Simulink online reference.

The following M-file program is a simple example that selects and runs **Check solver for code generation** on the `sldemo_mdldv` model.

```
function result = demo_modelAdvisor_CommandLine
model = 'sldemo_mdldv';
load_system(model);
```

```
% Get model advisor handle
MdlAdvHandle = Simulink.ModelAdvisor.getModelAdvisor(model);

% BaselineMode false is for verification, true is for baseline
% generation
MdlAdvHandle.setBaselineMode(true);

% As an example, here we only select the check of choice 'Check solver for
% code generation'
MdlAdvHandle.deselectCheckAll;
MdlAdvHandle.selectCheck('Check solver for code generation');

% Run the selected check
MdlAdvHandle.runCheck;

% Get check result
result = MdlAdvHandle.getCheckResult('Check solver for code generation');
```

Managing Model Versions

In this section...

“How Simulink Helps You Manage Model Versions” on page 4-99

“Model File Change Notification” on page 4-100

“Specifying the Current User” on page 4-101

“Model Properties Dialog Box” on page 4-103

“Creating a Model Change History” on page 4-111

“Version Control Properties” on page 4-112

How Simulink Helps You Manage Model Versions

The Simulink software has these features to help you to manage multiple versions of a model:

- Model File Change Notification helps you manage work with source control operations and multiple users
- As you edit a model, the Simulink software generates version control information about the model, including a version number, who created and last updated the model, and an optional change history. The Simulink software automatically saves these **Version Control Properties** with the model
- The Model Properties dialog box lets you edit some of the version control information stored in the model and select various version control options
- The Model Info block lets you display version control information, including information maintained by an external version control system, as an annotation block in a model diagram
- The Simulink software version control parameters let you access version control information from the MATLAB command line or an M-file
- The **Source Control** submenu of the **File** menu allows you to check models into and out of your source control system. See “Source Control Interface” in the online MATLAB documentation for more information.

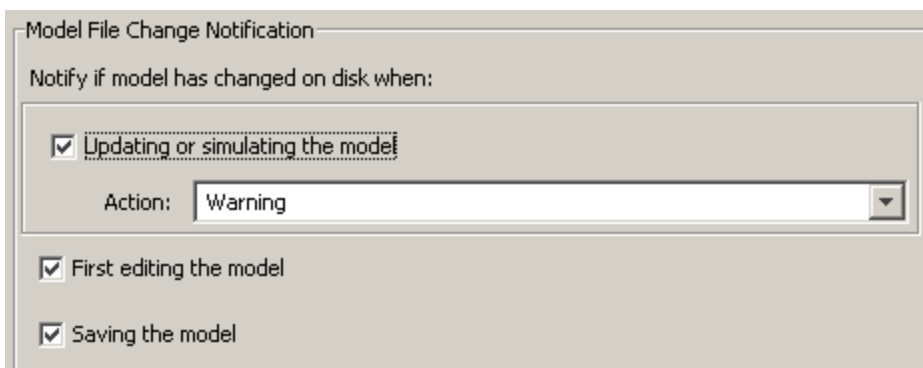
Model File Change Notification

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,

- Select **File > Preferences** in the Simulink product.
- Select **File > Preferences** in MATLAB to open the MATLAB Preferences, then select **Simulink** in the left pane, and click the button **Launch Simulink Preferences**.



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.

Specifying 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.

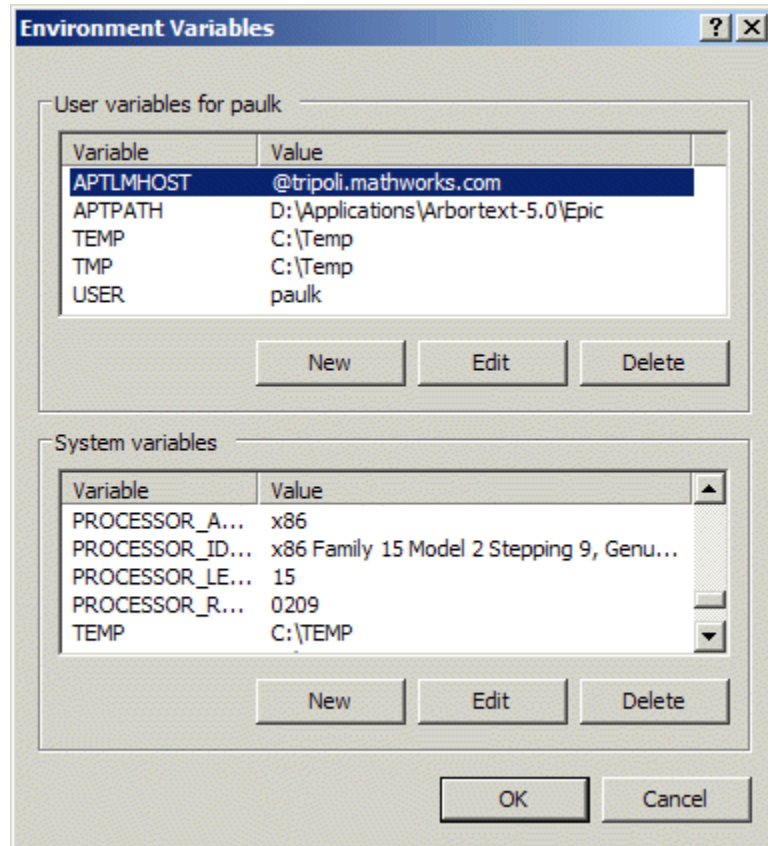
UNIX systems define the `USER` environment variable and set its value to the name you use to log on to your system. Thus, if you are using a UNIX system, you do not have to do anything to enable the Simulink software to identify you as the current user.

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.

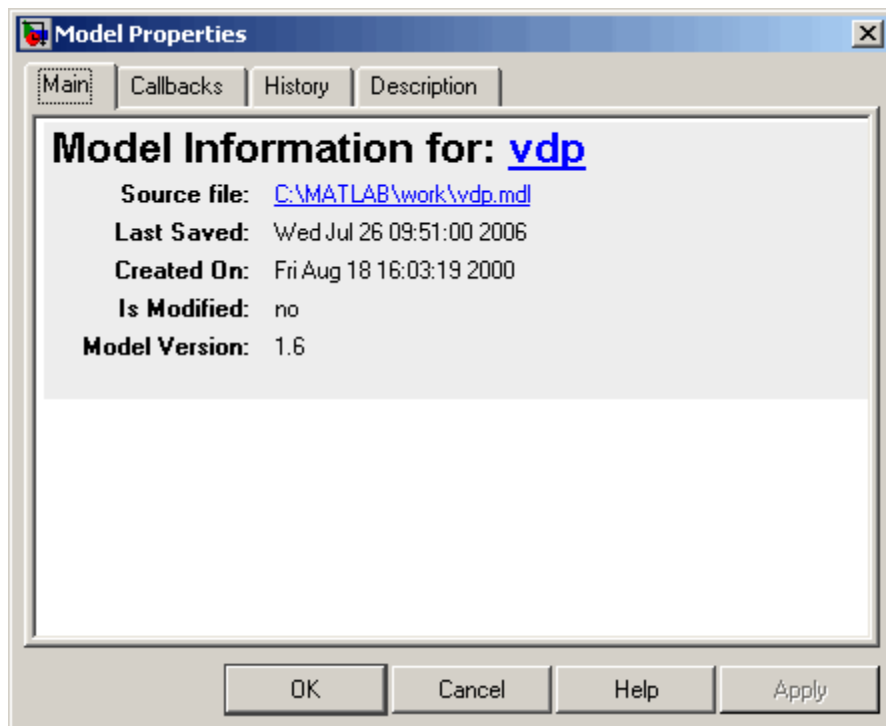
On Windows, use the **Environment** variables pane of the System Properties dialog box to set the `USER` environment variable (if it is not already defined). For Windows XP, access the **Environment** variables pane by clicking the **Environment Variables** button on the **Advanced** pane of the System Properties dialog box.



To display the System Properties dialog box, select **Start > Settings > Control Panel** to open the Control Panel. Double-click the **System** icon. To set the USER variable, enter USER in the **Variable** field and enter your login name in the **Value** field. Click **Set** to save the new environment variable. Then click **OK** to close the dialog box.

Model Properties Dialog Box

The Model Properties dialog box allows you to set various version control parameters and model callback functions. To display the dialog box, choose **Model Properties** from the **File** menu.



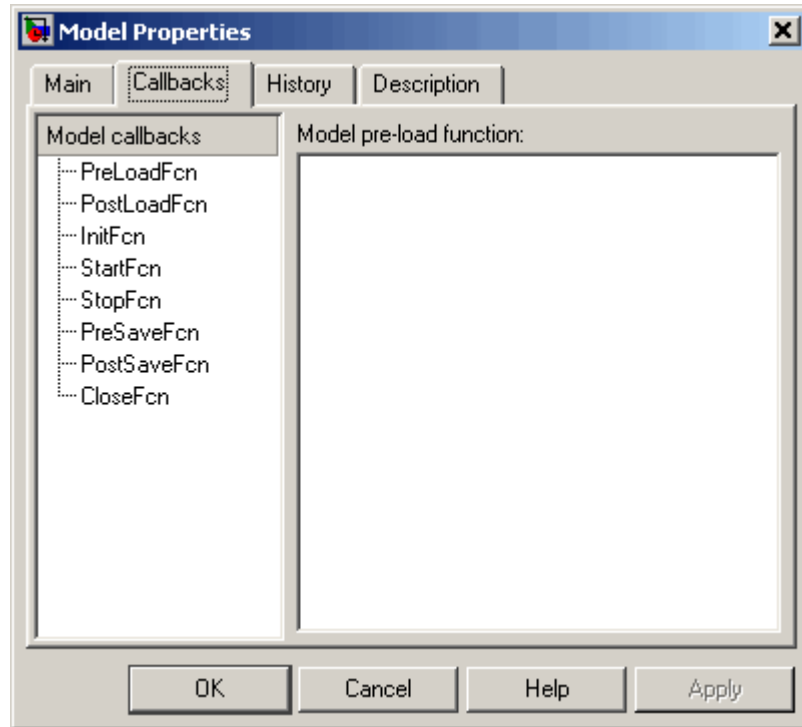
The dialog box includes the following panes.

Main Pane

The **Main** pane summarizes information about the current version of this model.

Callbacks Pane

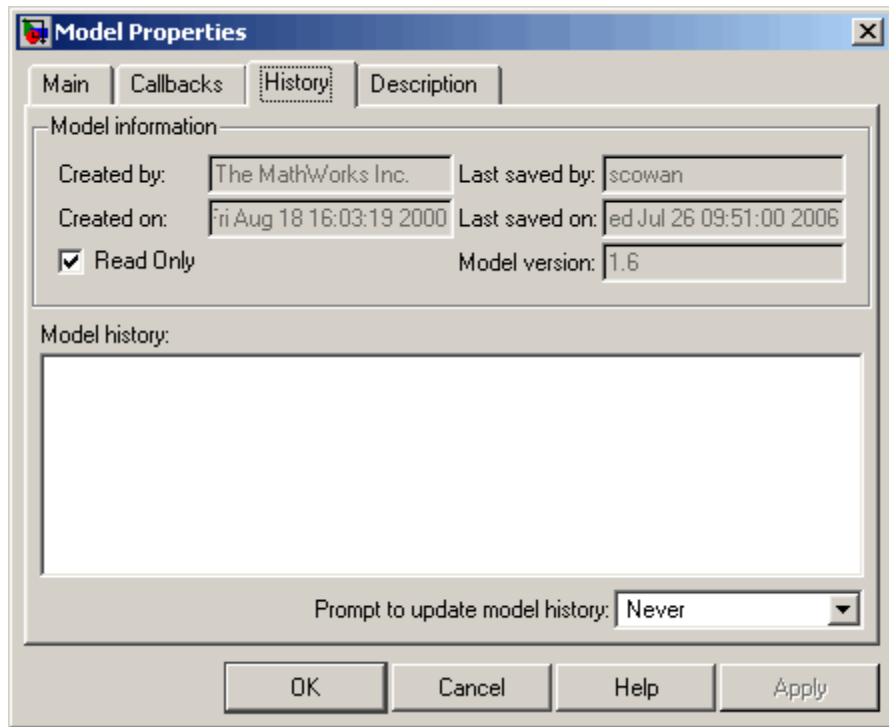
The **Callbacks** pane lets you specify functions to be invoked at specific points in the simulation of the model.



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 “Creating Model Callback Functions” on page 4-55 for information on the callback functions listed on this pane.

History Pane

The **History** pane allows you to enable, view, and edit this model’s change history.



The **History** pane has two control groups: the **Model information** group and the **Model History** group.

Model Information Controls

The contents of the **Model information** control group depend on the state of the **Read Only** check box.

Read Only Check Box Selected. When **Read Only** is selected, the dialog box shows the following fields grayed out.

- **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.

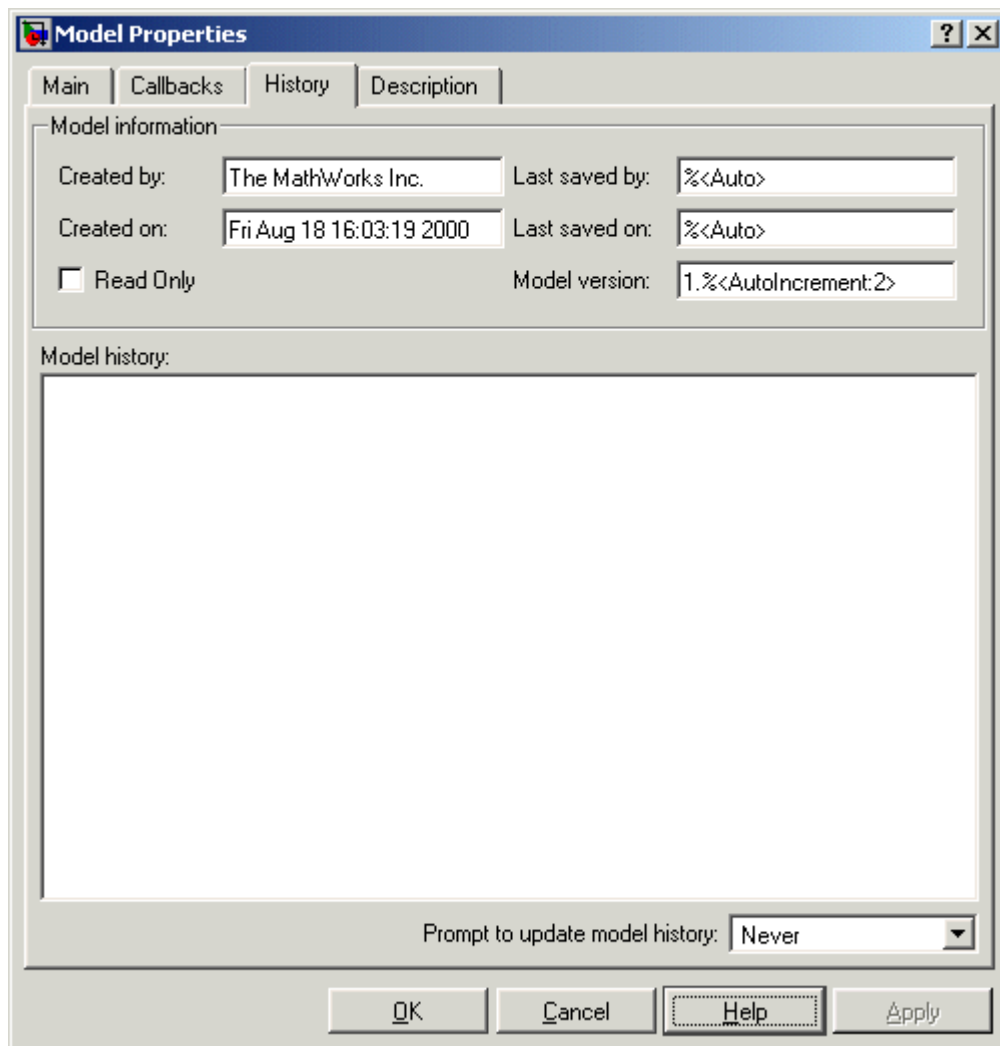
- **Last saved on**

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.

Read Only Check Box Deselected. When **Read Only** 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>`. The Simulink software 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. The Simulink software 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
```

The Simulink software increments `#` by 1 when saving the model. For example, when you save the model,

```
1.%<AutoIncrement:2>
```

becomes

```
1.%<AutoIncrement:3>
```

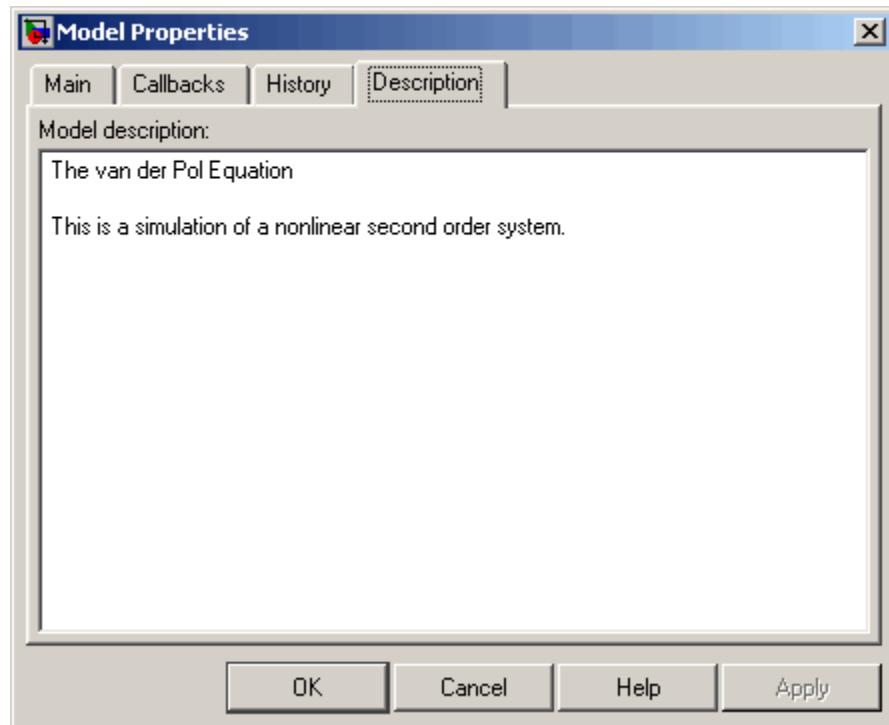
and the model version number is reported as 1.3.

Model History Controls

The model history controls group contains a scrollable text field and an option list. The text field displays the history for the model in a scrollable text field. To change the model history, edit the contents of this field. The option list allows you to enable or disable the Simulink software model history feature. To enable the history feature, select **When saving model** from the **Prompt to update model history** list. This causes the Simulink software to prompt you to enter a comment when saving the model. Typically you would enter any changes that you have made to the model since the last time you saved it. This information is stored in the model's change history log. See "Creating a Model Change History" on page 4-111 for more information. To disable the change history feature, select **Never** from the **Prompt to update model history** list.

Model Description Controls

This pane allows you to enter a description of the model. When typing `help` followed by the model name at the MATLAB prompt, the contents of the **Model description** field appear in the Command Window.

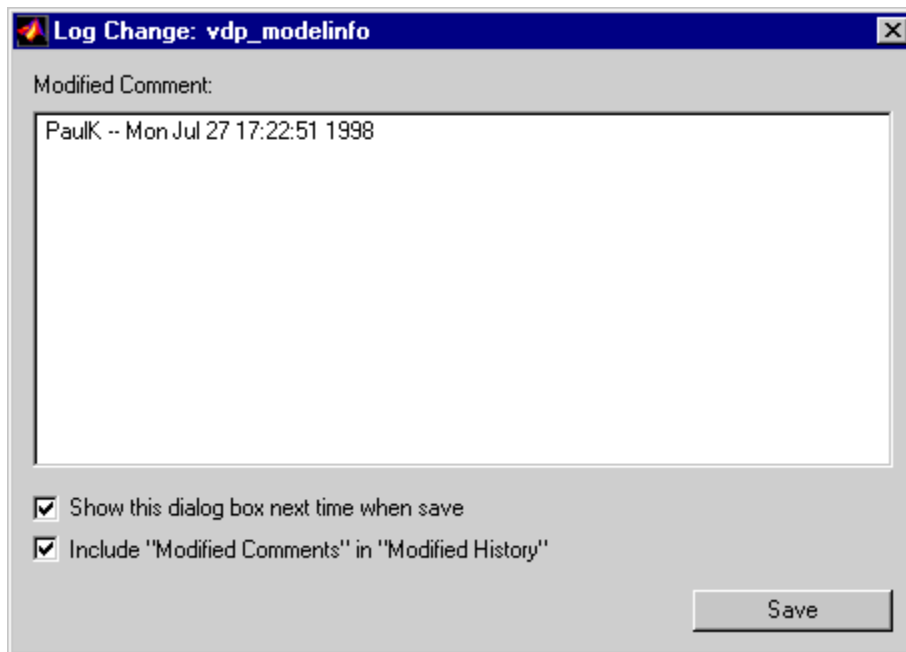


Creating a Model Change History

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.

Logging Changes

To start a change history, select **When saving model** from the **Prompt to update model history** list on the **History** pane on the Model Properties dialog box. The next time you save the model, a **Log Change** dialog box is displayed.



To add an item to the model's change history, enter the item in the **Modified Comments** edit field and click **Save**. If you do not want to enter an item for this session, clear the **Include "Modified Contents" in "Modified History"** option. To discontinue change logging, clear the **Show this dialog box next time when save** option.

Version Control Properties

Version control information is stored as model parameters in a model. You can access this information from the MATLAB command line or from an M-file, using the Simulink `get_param` command. The following table describes the model parameters used by Simulink to store version control information.

Property	Description
Created	Date created.
Creator	Name of the person who created this model.

Property	Description
LastModifiedBy	User name of the person who last modified this model.
ModifiedBy	Person who last modified this model.
ModifiedByFormat	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.
ModifiedDate	Date modified.
ModifiedDateFormat	Format of the ModifiedDate parameter. Value can be any string. The string can include the tag %<Auto>. The Simulink software replaces the tag with the current date and time when saving the model.
ModifiedComment	Comment entered by user who last updated this model.
ModifiedHistory	History of changes to this model.
ModelVersion	Version number.
ModelVersionFormat	Format of model version number. Can be any string. The string can contain the tag %<AutoIncrement:#> where # is an integer. The Simulink software replaces the tag with # when displaying the version number. It increments # when saving the model.
Description	Description of model.
LastModifiedDate	Date last modified.

Model Discretizer

In this section...

“What Is the Model Discretizer?” on page 4-114

“Requirements” on page 4-114

“How to Discretize a Model from the Model Discretizer GUI” on page 4-115

“Viewing the Discretized Model” on page 4-124

“How to Discretize Blocks from the Simulink Model” on page 4-127

“How to Discretize a Model from the MATLAB Command Window” on page 4-138

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

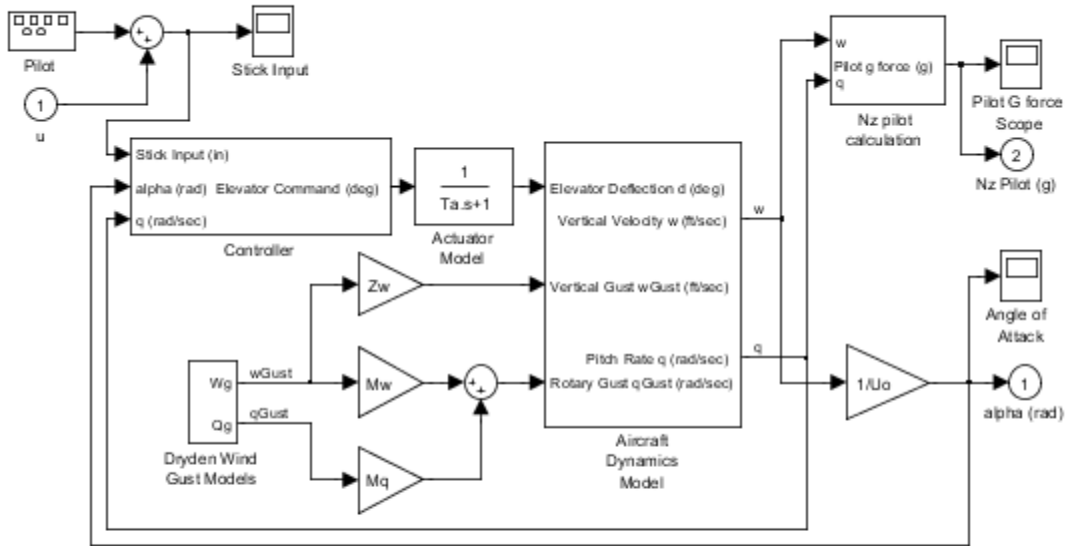
To use Model Discretizer, you must have a Control System Toolbox™ license, Version 5.2 or later.

How to Discretize a Model from the Model Discretizer GUI

To discretize a model:

- Start the Model Discretizer
- Specify the Transform Method
- Specify the Sample Time
- Specify the Discretization Method
- Discretize the Blocks

The f14 model, shown below, demonstrates the steps in discretizing a model.



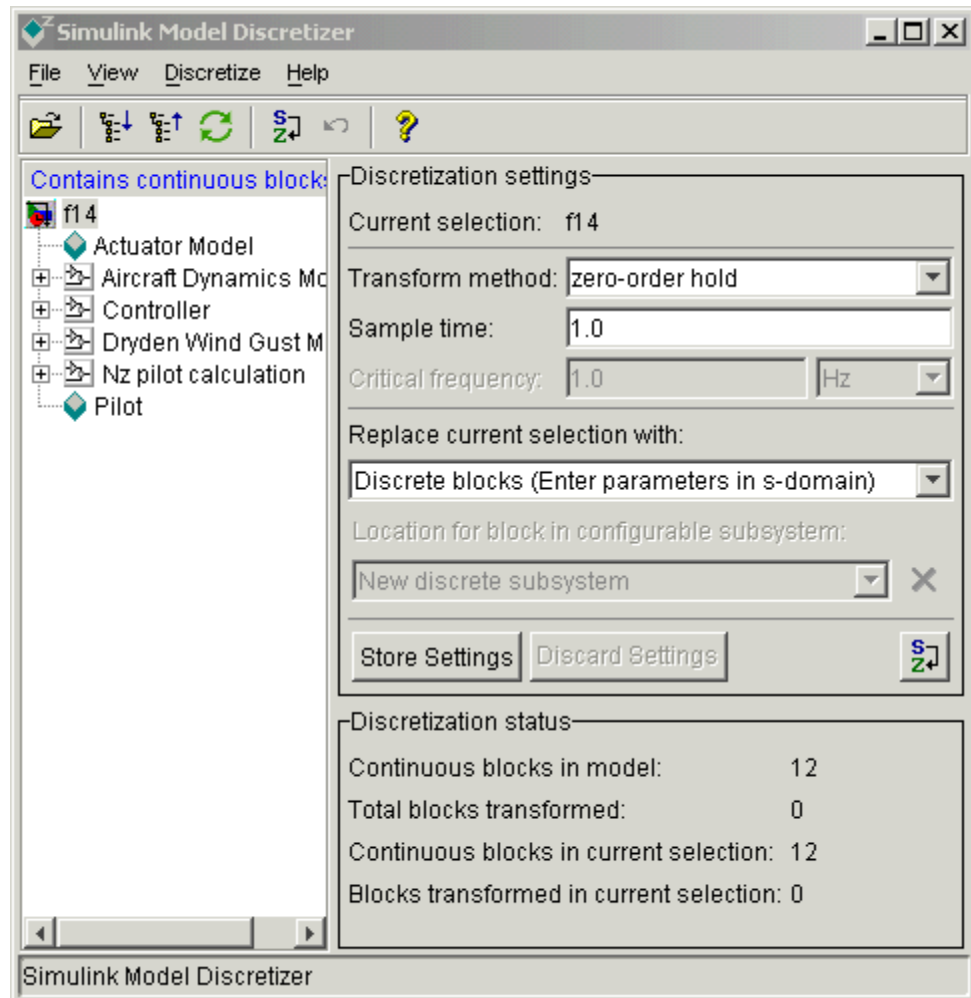
F-14 Flight Control
 (an updated version of this demo is available
 by running 'sidemo_f14')

Copyright 1990-2005 The MathWorks Inc.

Start Model Discretizer

To open the tool, select **Tools > Control Design > Model Discretizer** from the model editor's menu bar.

The **Simulink Model Discretizer** appears.



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 **Load model** from the **File** menu.

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 “Linear, Time-Invariant Models” in 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 4-119.

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 4-119
Creates a discrete block whose parameters are retained from the corresponding continuous block.
- “Discrete blocks (Enter parameters in z-domain)” on page 4-120
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 4-121
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 4-122
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.

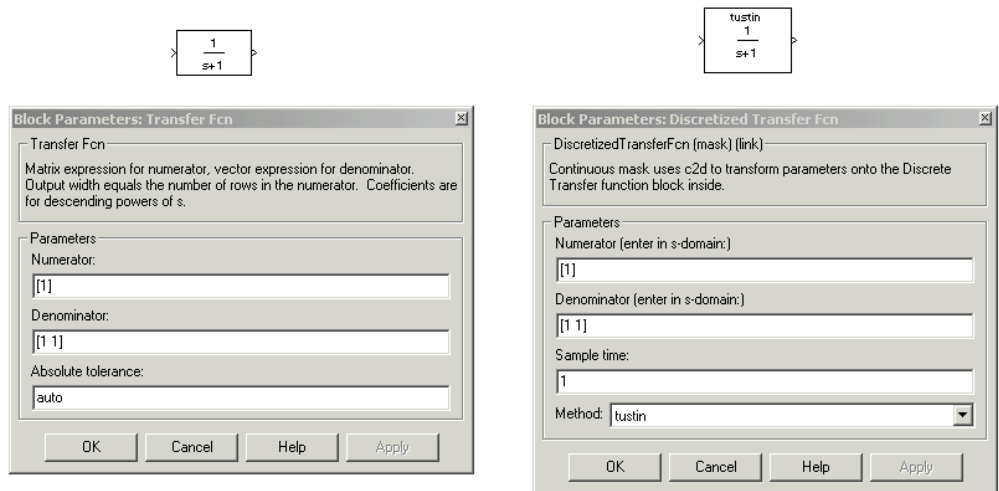
The block is implemented as a masked discrete block that uses `c2d` to transform the continuous parameters to discrete parameters in the mask initialization code.

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 4-118.

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.

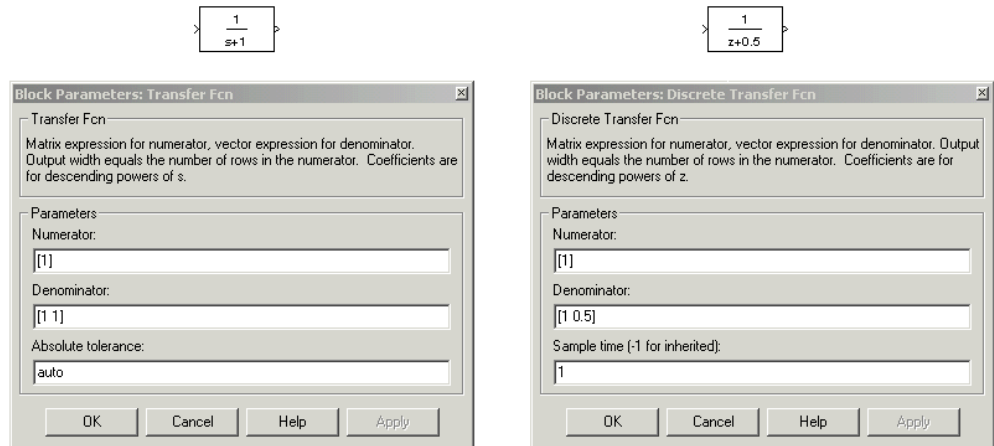


Discrete blocks (Enter parameters in z-domain). Creates a discrete block whose parameters are “hard-coded” values placed directly into the block’s dialog box. Model Discretizer uses the c2d function to obtain the discretized parameters, if needed.

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 directory 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 directory 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 directory. 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.

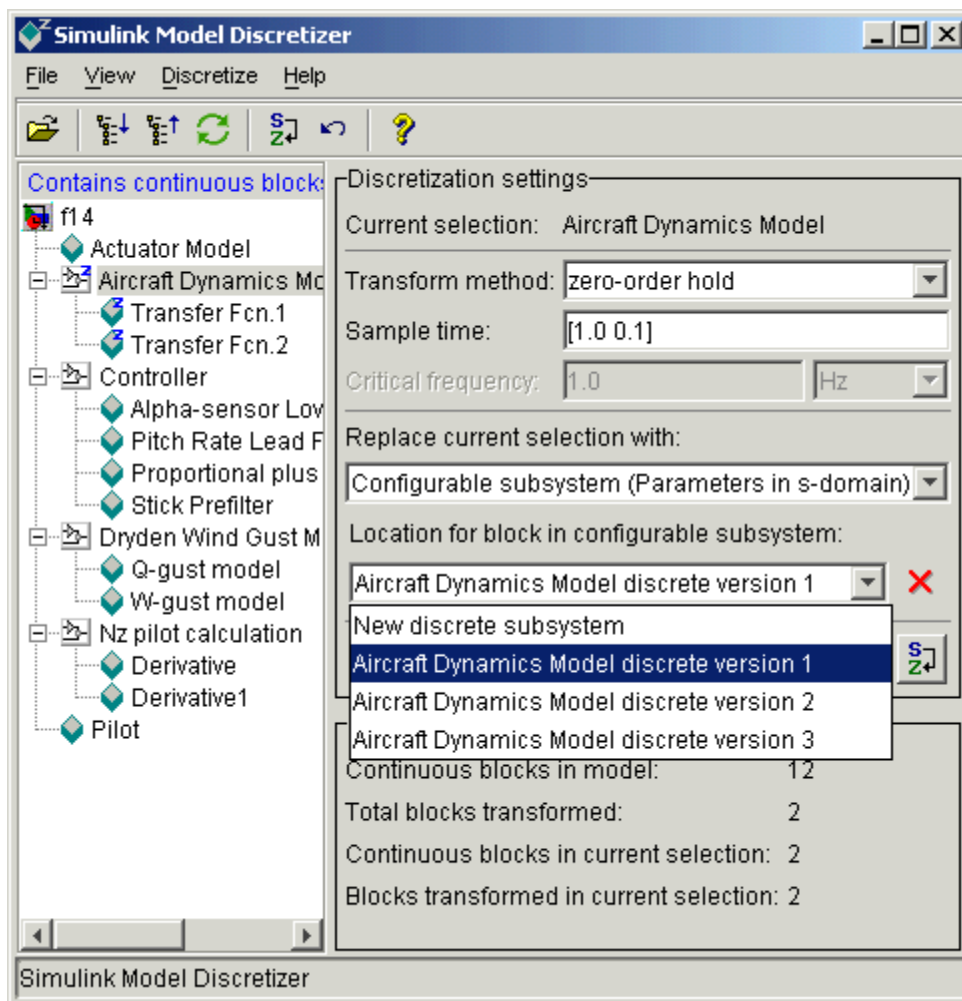
Viewing the Discretized Model

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.

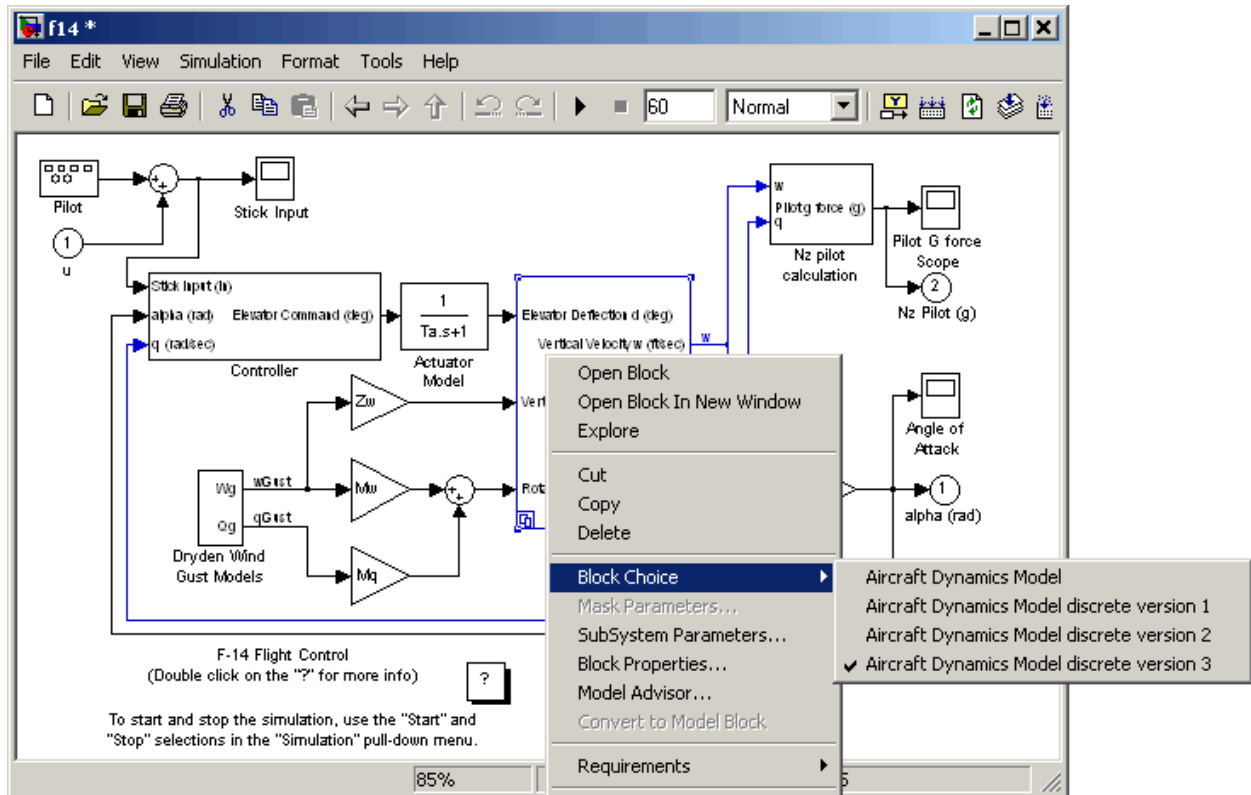
The following figure shows that the Aircraft Dynamics Model subsystem has been discretized into a configurable subsystem with three discretization candidates.



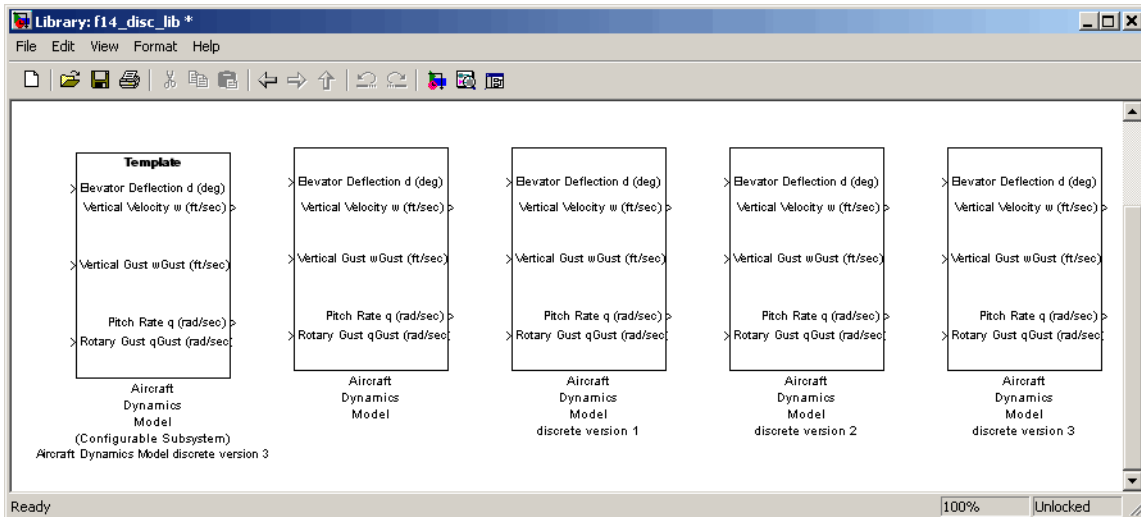
The other blocks in this f14 model have not been discretized.

4 Creating a Model

The following figure shows the Aircraft Dynamics Model subsystem of the f14 demo 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.

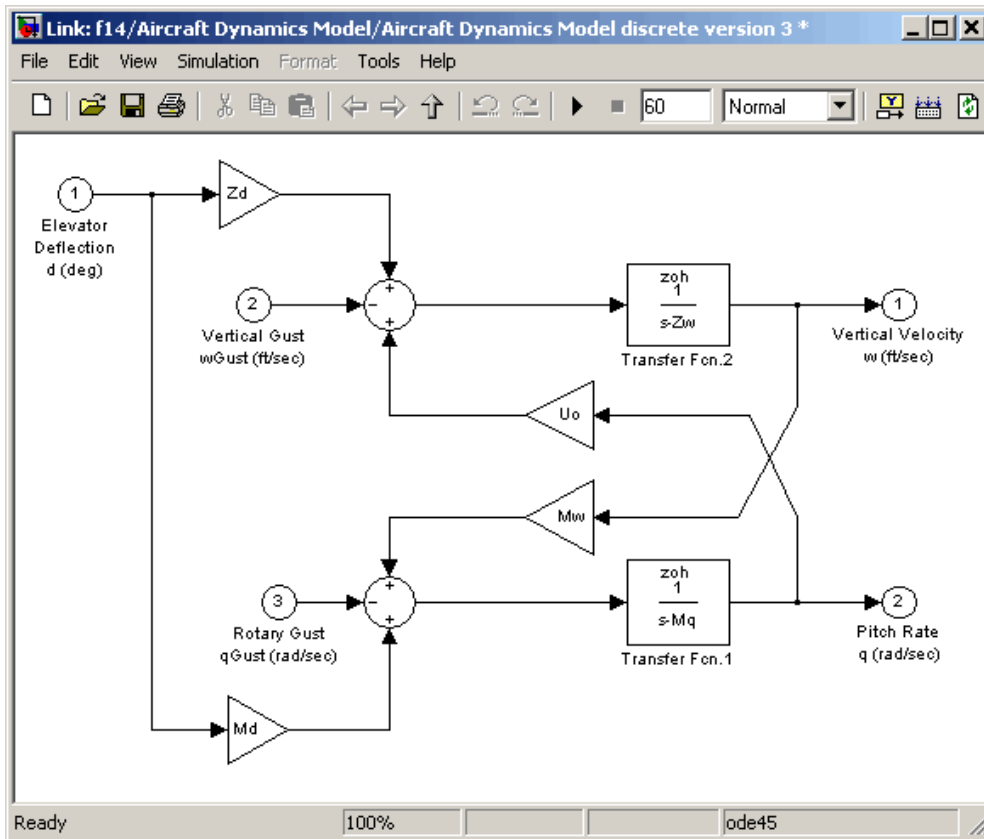
How to Discretize Blocks from the Simulink Model

You can replace continuous blocks in a Simulink software model with the equivalent blocks discretized in the s-domain using the Discretizing library.

The procedure below shows how to replace a continuous Transfer Fcn block in the Aircraft Dynamics Model subsystem of the f14 model with a discretized Transfer Fcn block from the Discretizing Library. The block is discretized

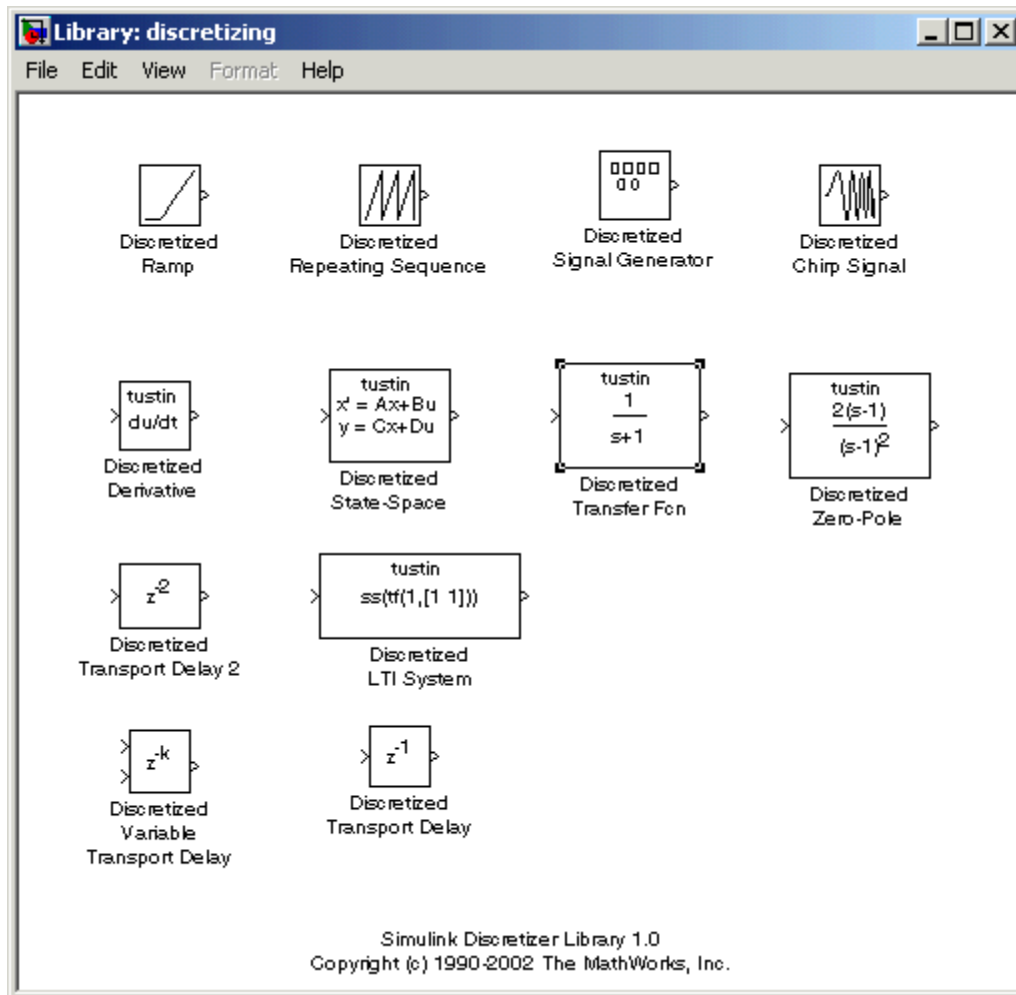
in the s-domain with a zero-order hold transform method and a two second sample time.

- 1 Open the f14 model.
- 2 Open the Aircraft Dynamics Model subsystem in the f14 model.



- 3 Open the Discretizing library window.
Enter discretizing at the MATLAB command prompt.

The **Library: discretizing** window opens.

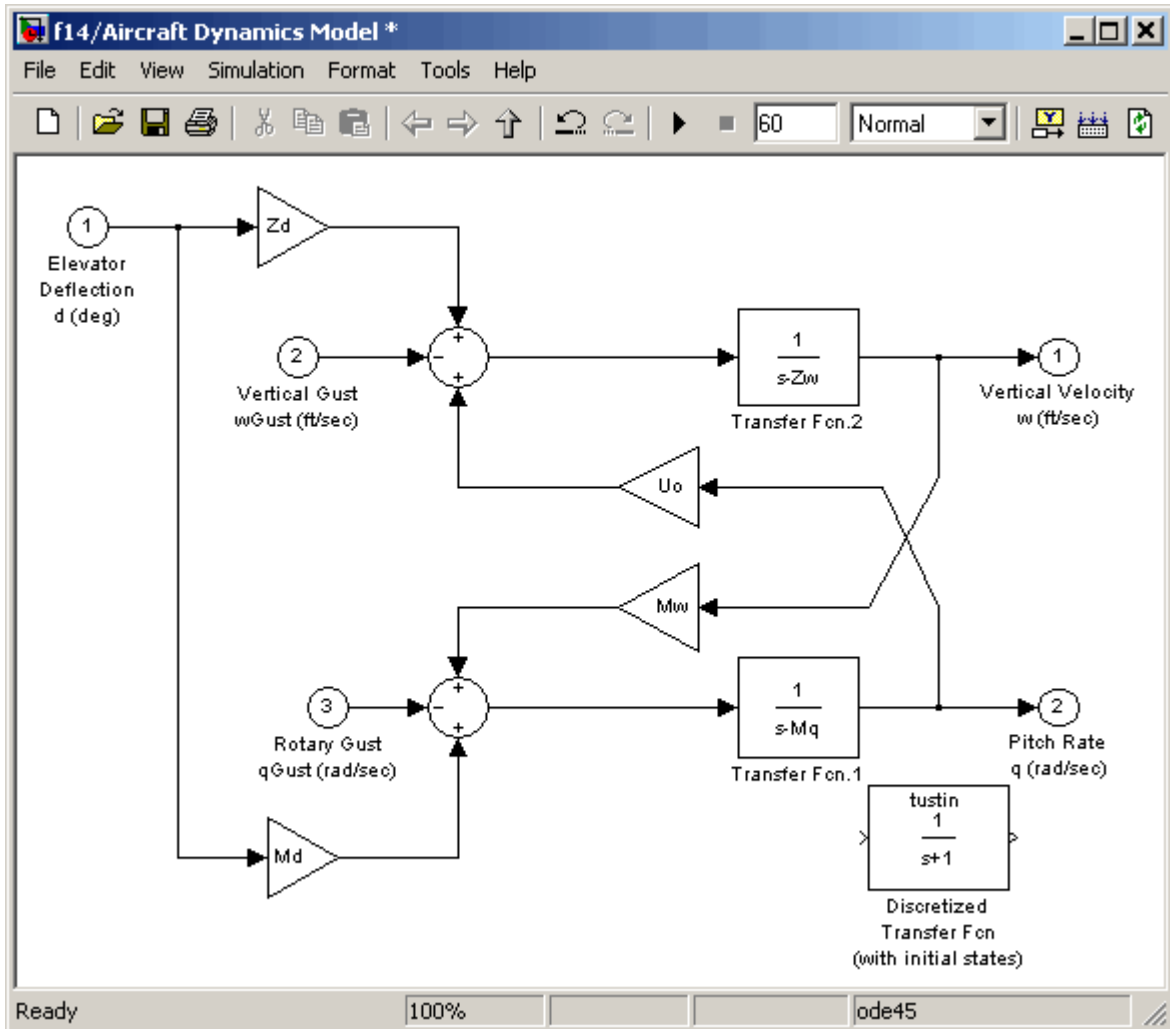


This library contains s-domain discretized blocks.

- 4** Add the Discretized Transfer Fcn block to the **f14/Aircraft Dynamics Model** window.

4 Creating a Model

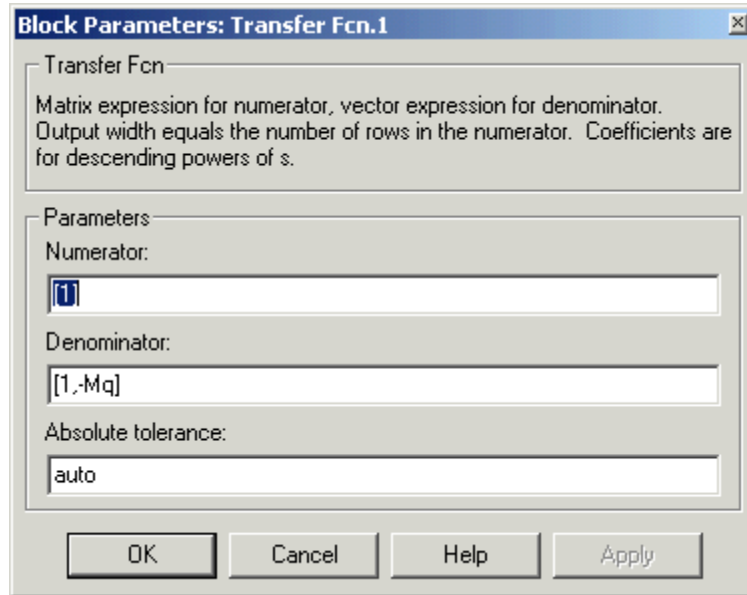
- a Click the Discretized Transfer Fcn block in the **Library: discretizing** window.
- b Drag it into the **f14/Aircraft Dynamics Model** window.



- 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.

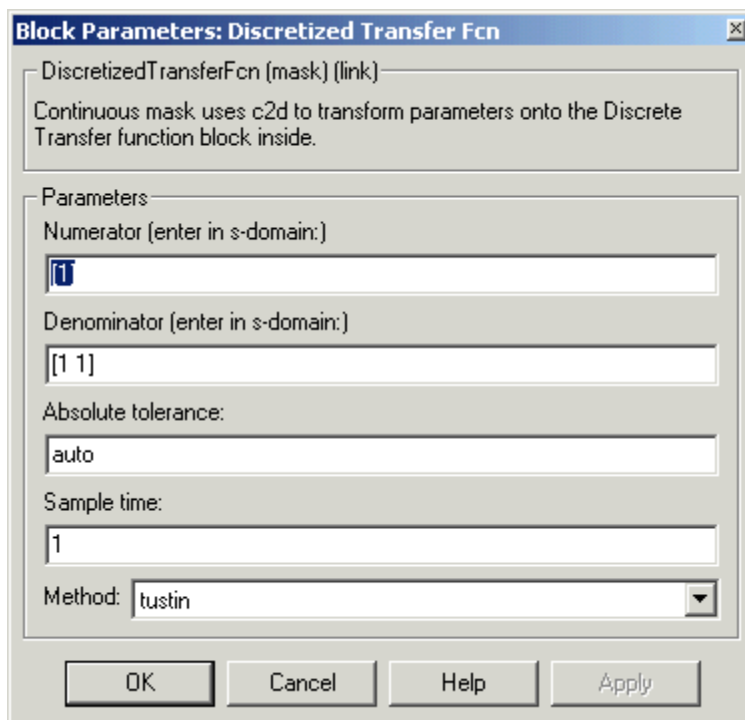
The Block Parameters: Transfer Fcn.1 dialog box opens.



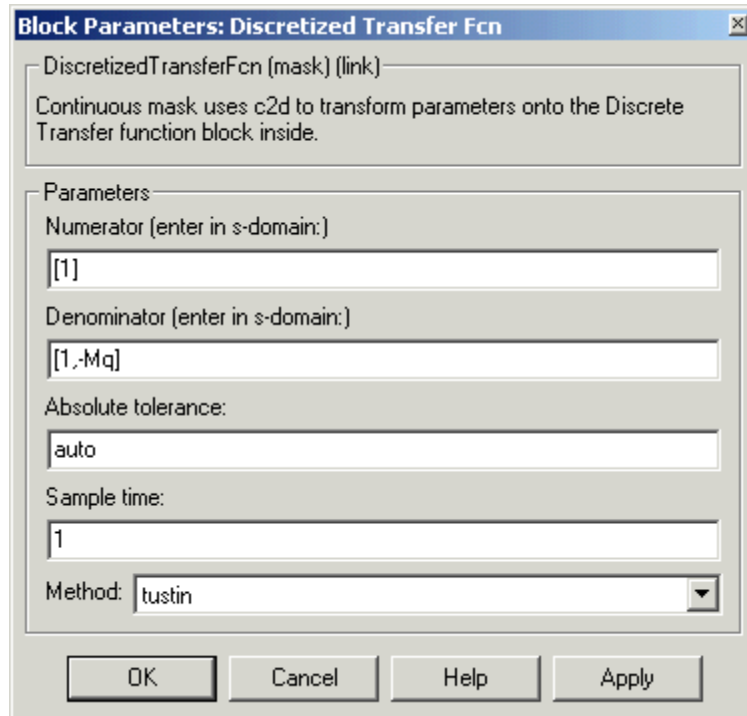
- 6 Open the parameter dialog box for the Discretized Transfer Fcn block.

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.

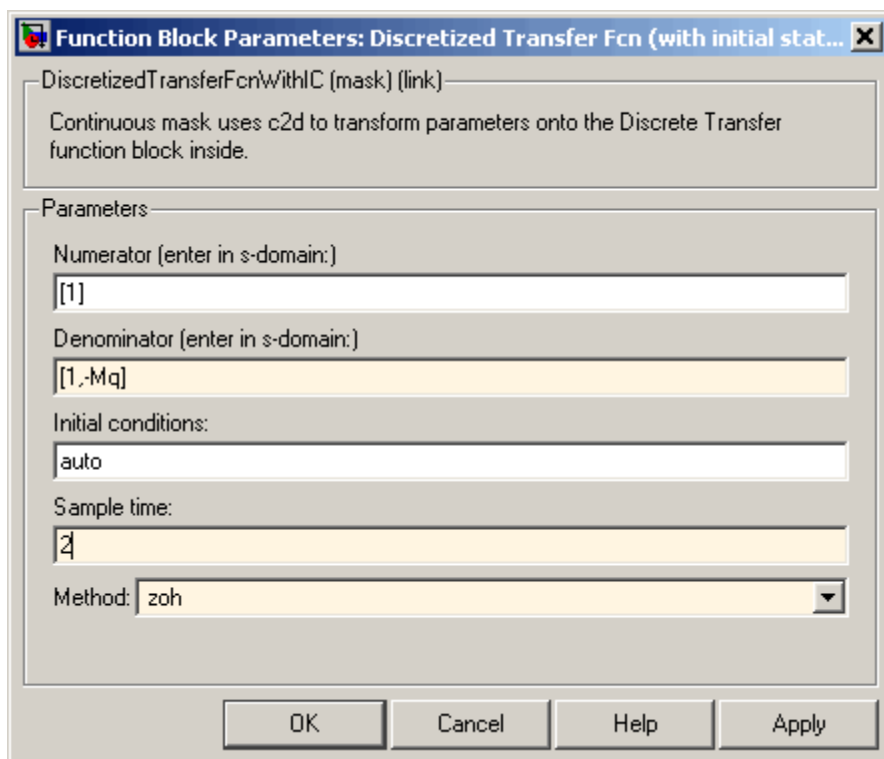


The image shows a dialog box titled "Block Parameters: Discretized Transfer Fcn". It contains the following fields and controls:

- A link field: "DiscretizedTransferFcn (mask) (link)" with a small 'x' icon in the top right corner.
- A text area: "Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside."
- A "Parameters" section with the following fields:
 - "Numerator (enter in s-domain:)" with a text box containing "[1]".
 - "Denominator (enter in s-domain:)" with a text box containing "[1,-Mq]".
 - "Absolute tolerance:" with a text box containing "auto".
 - "Sample time:" with a text box containing "1".
 - "Method:" with a drop-down menu currently showing "tustin".
- Buttons at the bottom: "OK", "Cancel", "Help", and "Apply".

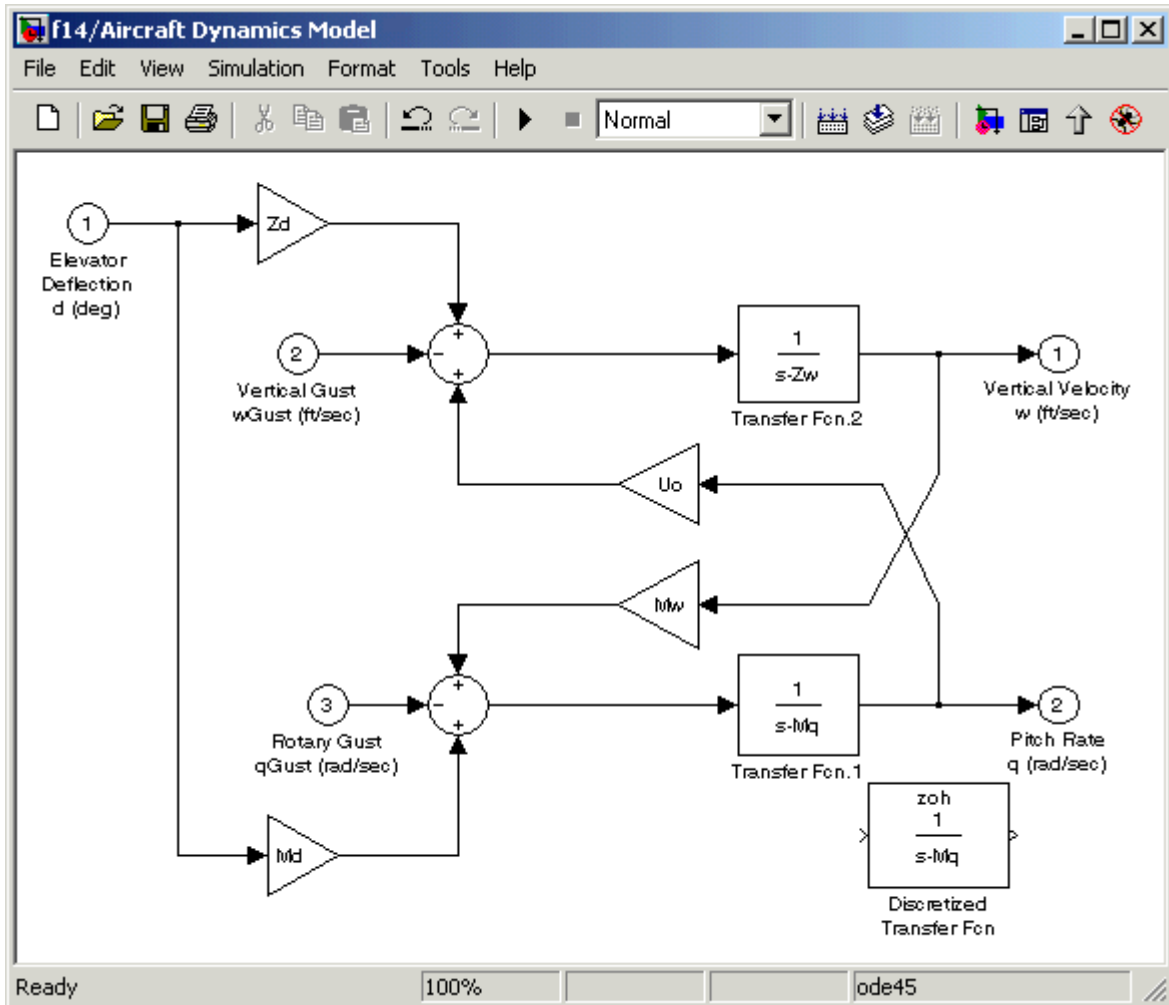
- 7** Enter 2 in the **Sample time** field.
- 8** Select **zoh** from the **Method** drop-down list.

The parameter dialog box for the Discretized Transfer Fcn now looks like this.



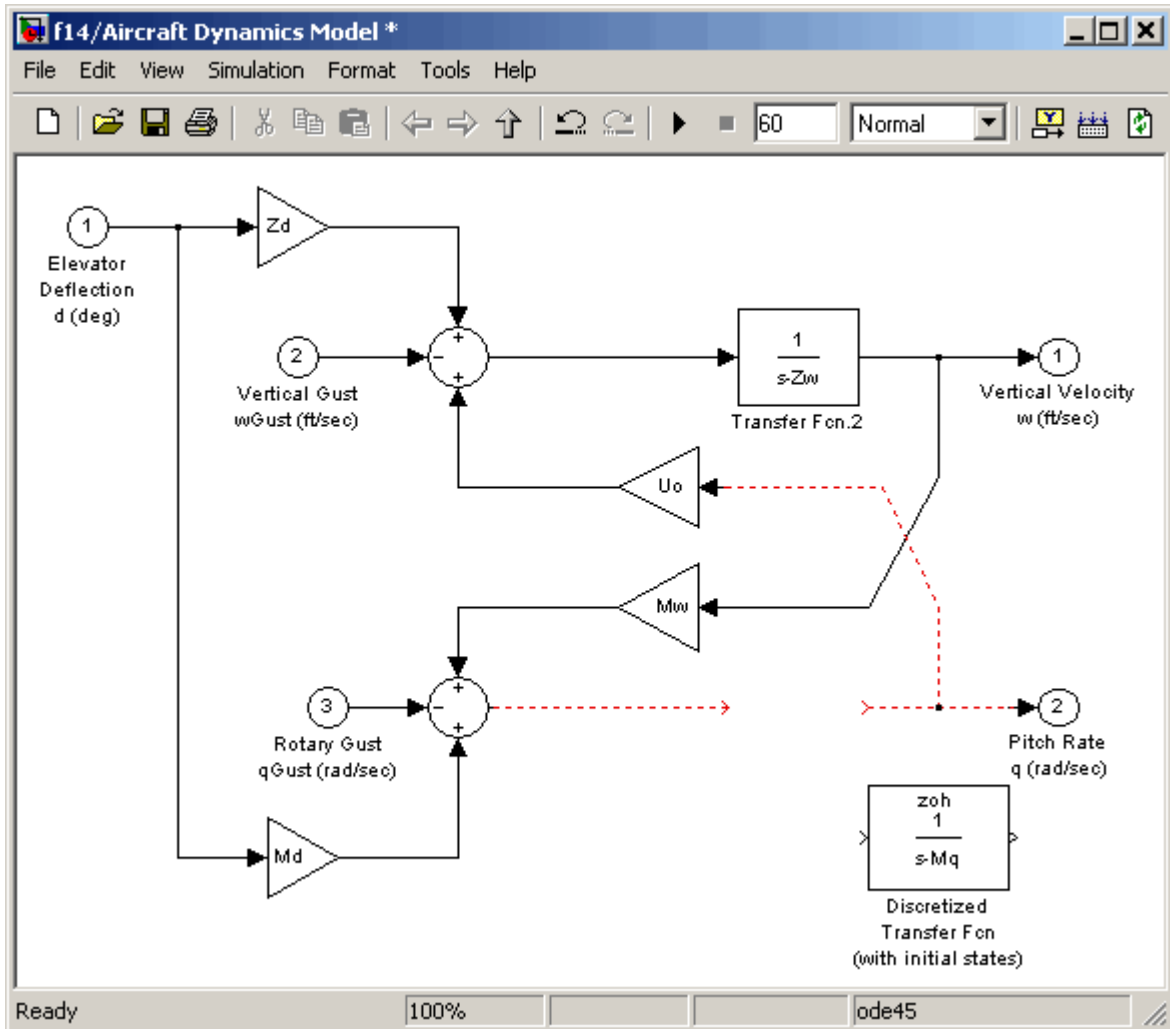
9 Click **OK**.

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.

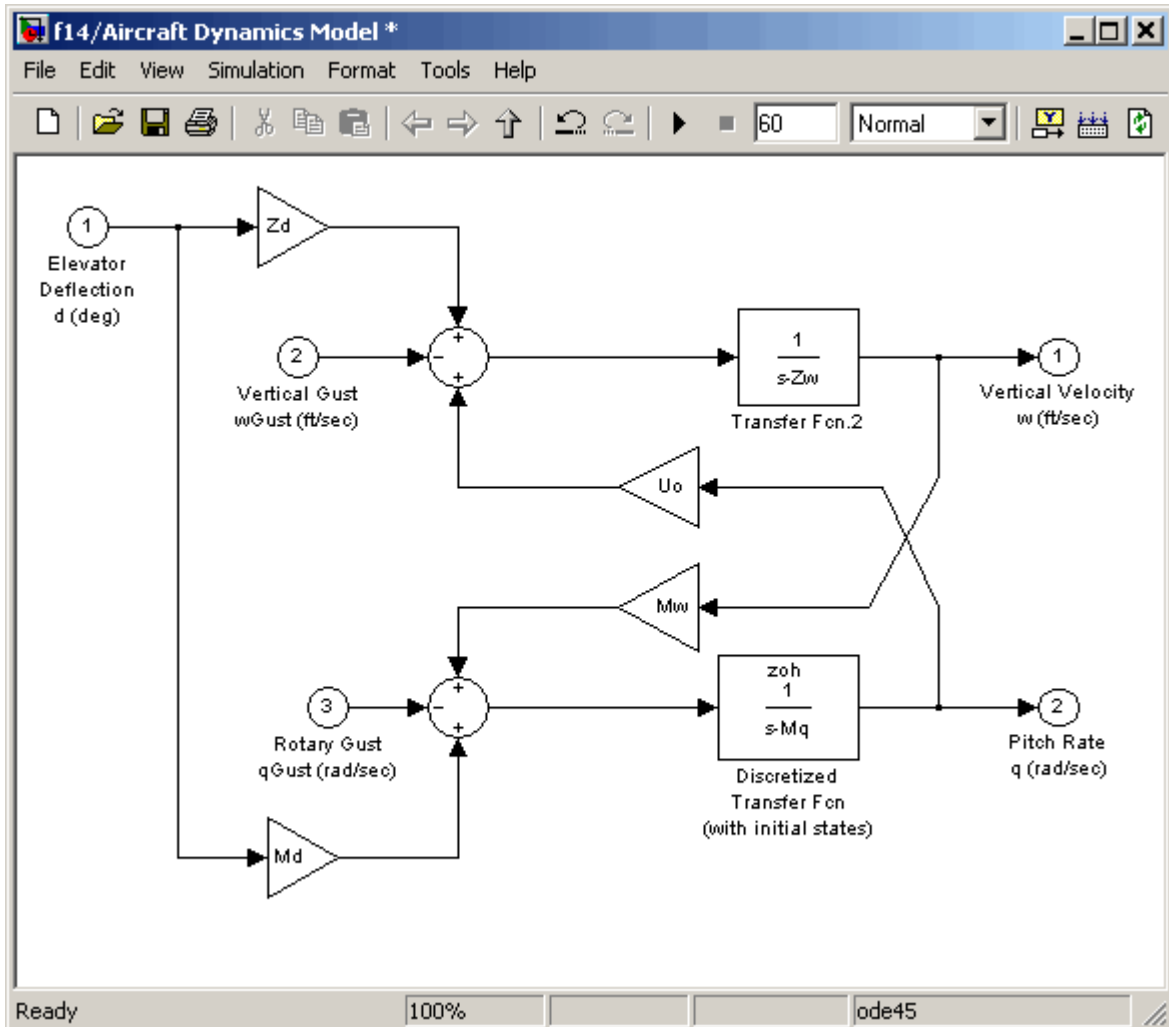
The f14/Aircraft Dynamics Model window now looks like this.



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.



How to Discretize a Model from the MATLAB Command Window

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')
```

For more information on the `sldiscmdl` function, see “Model Construction” in the *Simulink Reference*.

Creating Conditional Subsystems

- “About Conditional Subsystems” on page 5-2
- “Enabled Subsystems” on page 5-4
- “Triggered Subsystems” on page 5-14
- “Triggered and Enabled Subsystems” on page 5-19
- “Function-Call Subsystems” on page 5-24
- “Conditional Execution Behavior” on page 5-25

About 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 whose execution can be externally controlled. For information that applies to all subsystems, see “Creating Subsystems” 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 the 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. The following types of conditional subsystems are supported:

- 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 while the control signal remains positive. Enabled subsystems are described in more detail in “Enabled Subsystems” on page 5-4.
- 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. Triggered subsystems are described in more detail in “Triggered Subsystems” on page 5-14.
- A *triggered and enabled subsystem* executes once on the time step when a trigger event occurs if the enable control signal has a positive value at that step. See “Triggered and Enabled Subsystems” on page 5-19 for more information.
- A *control flow subsystem* executes one or more times at the current time step when enabled by a control flow block that implements control logic similar to that expressed by programming language control flow statements (e.g., *if-then*, *while*, *do*, and *for*). See “Modeling Control Flow Logic” on page 4-44 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 5-11 for more information.

For examples of conditional subsystems, see:

- Simulink Subsystem Semantics
- Triggered Subsystems
- Enabled Subsystems
- Advanced Enabled Subsystems

Enabled Subsystems

In this section...

“What Are Enabled Subsystems?” on page 5-4

“Creating an Enabled Subsystem” on page 5-5

“Blocks an Enabled Subsystem Can Contain” on page 5-9

“Using Blocks with Constant Sample Times in Enabled Subsystems” on page 5-11

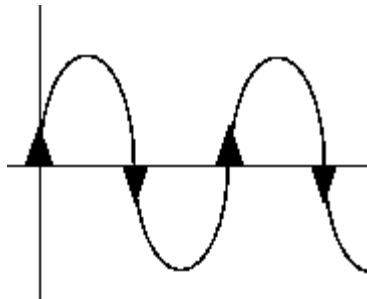
What Are Enabled Subsystems?

Enabled subsystems are subsystems that execute at each simulation step where the control signal has a positive value.

An enabled subsystem has a single control input, which can be scalar or vector valued.

- 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* of the vector elements is greater than zero.

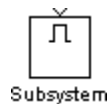
For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled, as shown in this figure. An up arrow signifies enable, a down arrow disable.



The Simulink software uses the zero-crossing slope method to determine whether an enable is to occur. If the signal crosses zero and the slope is positive, the subsystem is enabled. If the slope is negative at the zero crossing, the subsystem is disabled.

Creating an Enabled Subsystem

You create an enabled subsystem by copying an Enable block from the Ports & Subsystems library into a subsystem. 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 dialog box.
- 2 Select Dialog in the **Source of initial output value** drop-down list.
- 3 Specify the **Initial output** parameter.

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 5-7.

Inheriting Initial Conditions. The initial output value of the subsystem can be inherited from the following sources:

- Output port of another conditionally executed subsystem
- Merge block (with Initial output specified)

- Function-Call Model Reference block
- Constant block (simplified initialization mode only)
- IC block (simplified initialization mode only)

The procedure you use to inherit the initial conditions of the 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 dialog box.
- 2 Select Dialog in the **Source of initial output value** drop-down list.
- 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 dialog box.
- 2 Select Input signal in the **Source of initial output value** drop-down list.
- 3 Click **OK**.

The **Initial output** and **Output when disabled** parameters are disabled, and the values are both inherited from the input signal.

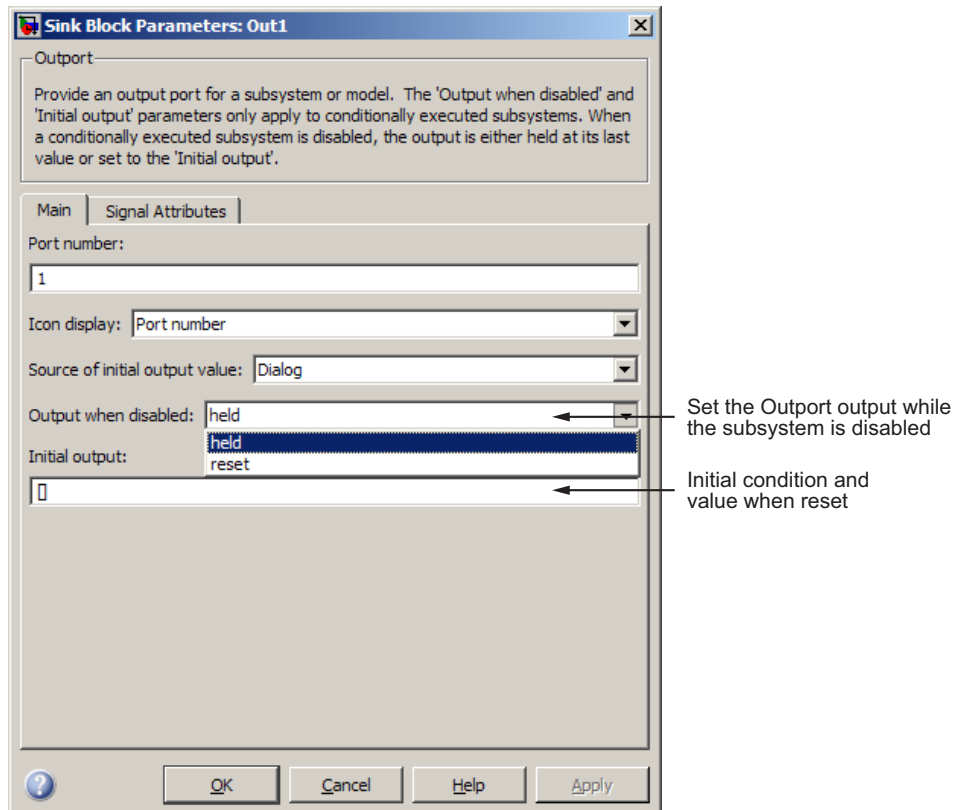
For more information on classic and simplified initialization mode, see “Underspecified initialization detection”.

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 each Outport block's dialog box and select one of the choices for the **Output when disabled** parameter, as shown in the following dialog box:

- 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.

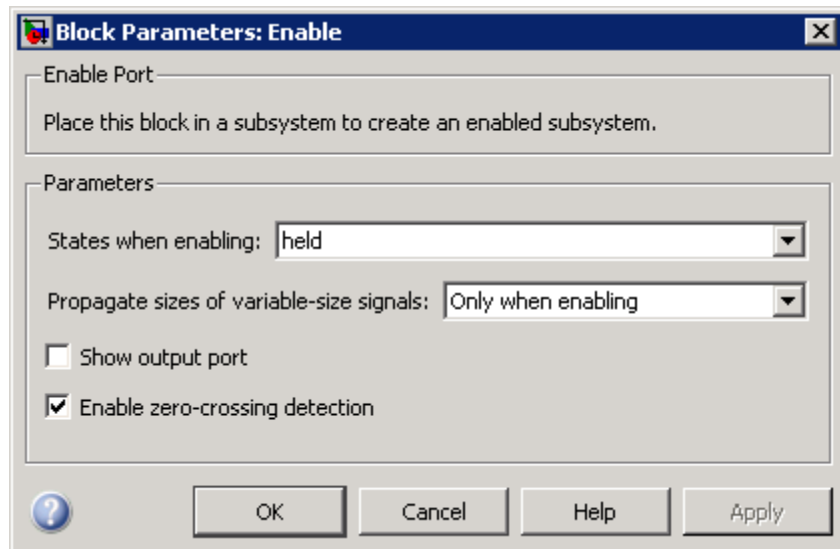
If you are using simplified initialization mode, you must select **held** when connecting a conditionally executed subsystem to a Merge block. For more information, see “Underspecified initialization detection”.

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 Enable block 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.

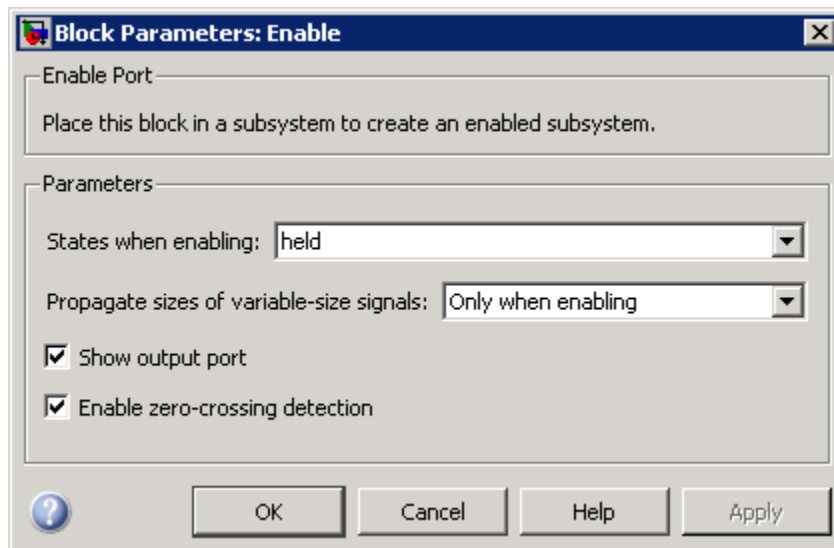


Note If you are using simplified initialization mode, subsystem elapsed time is always reset on first execution after becoming enabled, whether or not the subsystem is configured to reset on enable.

For more information on simplified initialization mode, see “Underspecified initialization detection”.

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 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

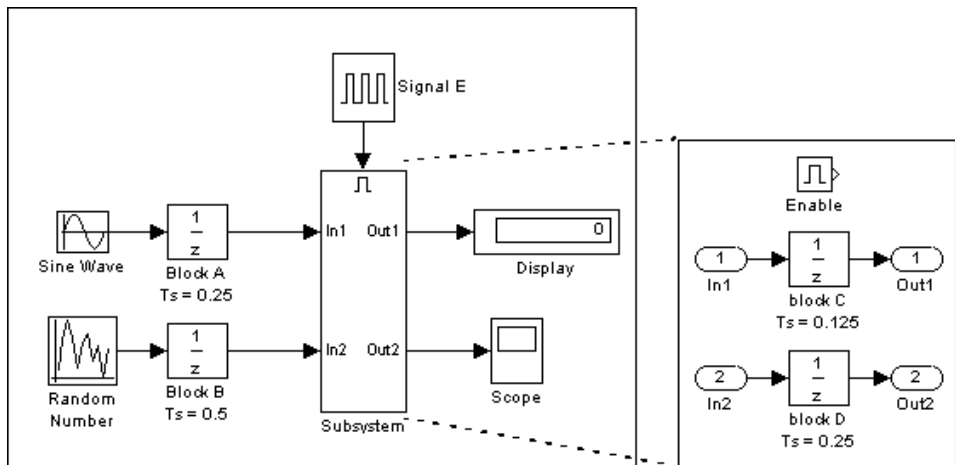
executes, and only when their sample times are synchronized with the simulation sample time. Enabled subsystems and the model use a common clock.

Note Enabled subsystems can contain Goto blocks. However, only state ports can connect to Goto blocks in an enabled subsystem. See the demo model, `clutch`, for an example of how to use Goto blocks in an enabled subsystem.

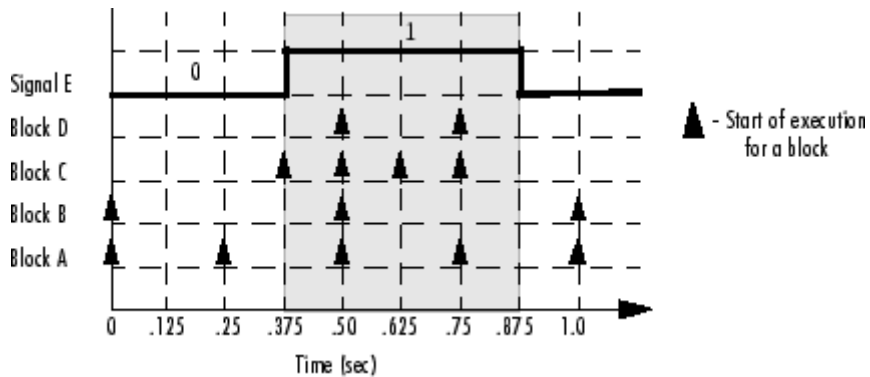
For example, this system contains four discrete blocks and a control signal. The discrete blocks are

- 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.



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.

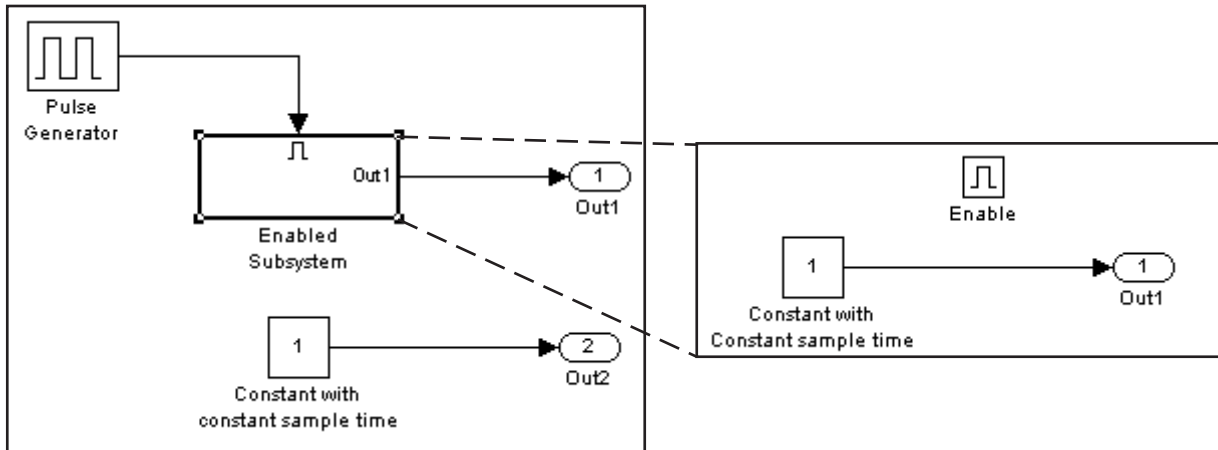
Using Blocks with Constant Sample Times in Enabled Subsystems

Certain restrictions apply when you connect blocks with constant sample times (see “Constant Sample Time” on page 3-17) 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.

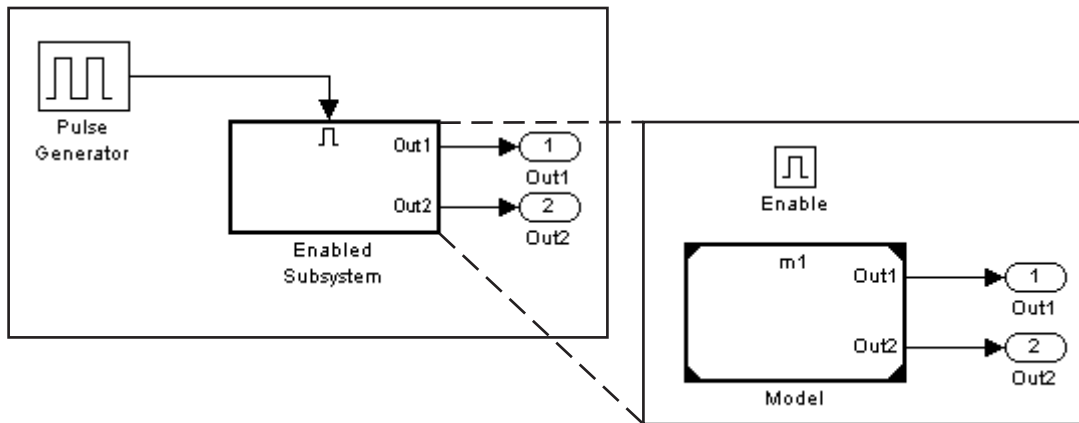
To avoid the error or conversion, either manually change the sample time of the block to a non-constant sample time or use a Signal Conversion block. The example below shows how to use the Signal Conversion block to avoid these errors.

Consider the following model `m1.mdl`.



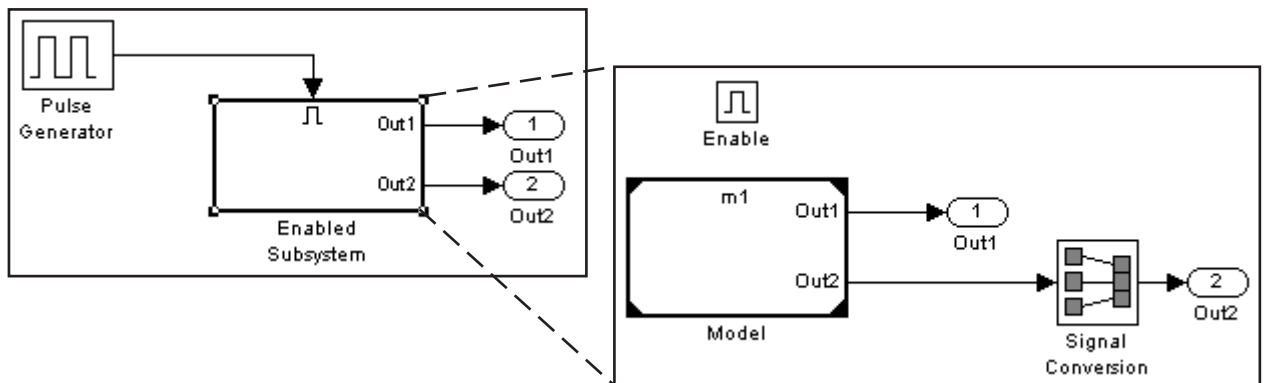
The two Constant blocks in this model have constant sample times. When you simulate the model, the Simulink software converts the sample time of the Constant block inside the enabled subsystem to the rate of the Pulse Generator. If you simulate the model with sample time colors displayed (see “Displaying Sample Time Colors” on page 4-12), the Pulse Generator and Enabled Subsystem blocks are colored red. However, the Constant and Outport 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 Chapter 6, “Referencing a Model”.)



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 Chapter 6, “Referencing a Model”.)

To avoid this error, insert a Signal Conversion block between the second output of the Model block and the enabled subsystem’s Outputport block.



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, that is, systems that contain multiple sample times.

Triggered Subsystems

In this section...

“What Are Triggered Subsystems?” on page 5-14

“Creating a Triggered Subsystem” on page 5-15

“Blocks That a Triggered Subsystem Can Contain” on page 5-18

What Are Triggered Subsystems?

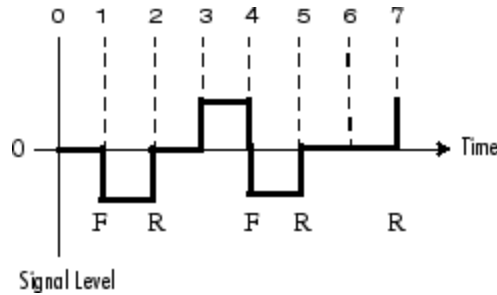
Triggered subsystems are subsystems that execute each time a trigger event occurs.

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:

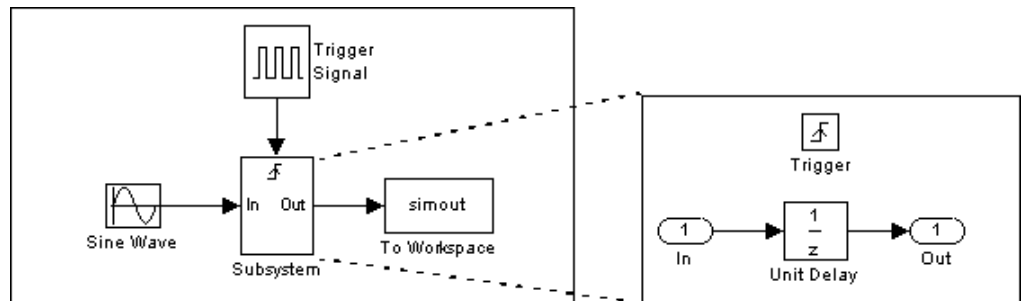
- **rising** triggers execution of the subsystem when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative).
- **falling** triggers execution of the subsystem when the control signal falls from a positive or a zero value to a negative value (or zero if the initial value is positive).
- **either** triggers execution of the subsystem when the signal is either rising or falling.

Note In the case of discrete systems, a signal’s rising or falling from zero is considered a trigger event only if the signal has remained at zero for more than one time step preceding the rise or fall. This 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 has remained at zero for only one time step when the rise occurs.



A simple example of a triggered subsystem is illustrated.



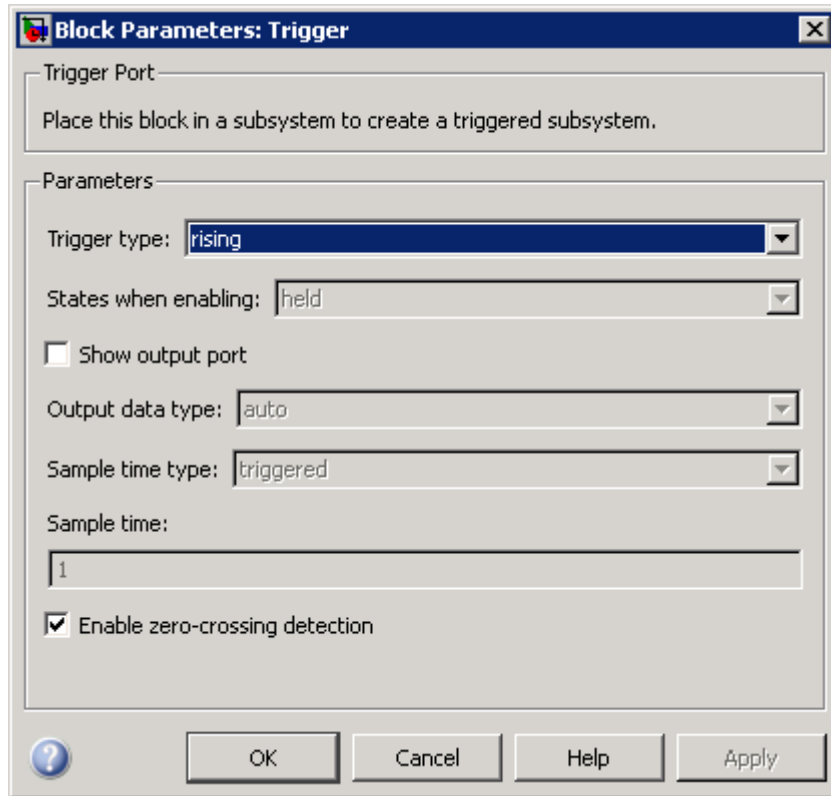
In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Creating a Triggered Subsystem

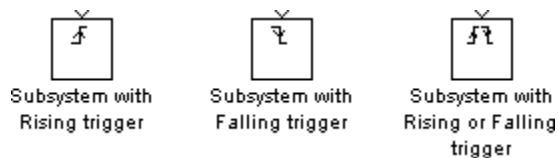
You create a triggered subsystem by copying the Trigger block from the Ports & Subsystems library into a subsystem. The Simulink software adds a trigger symbol and a trigger control input port to the Subsystem block.



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.



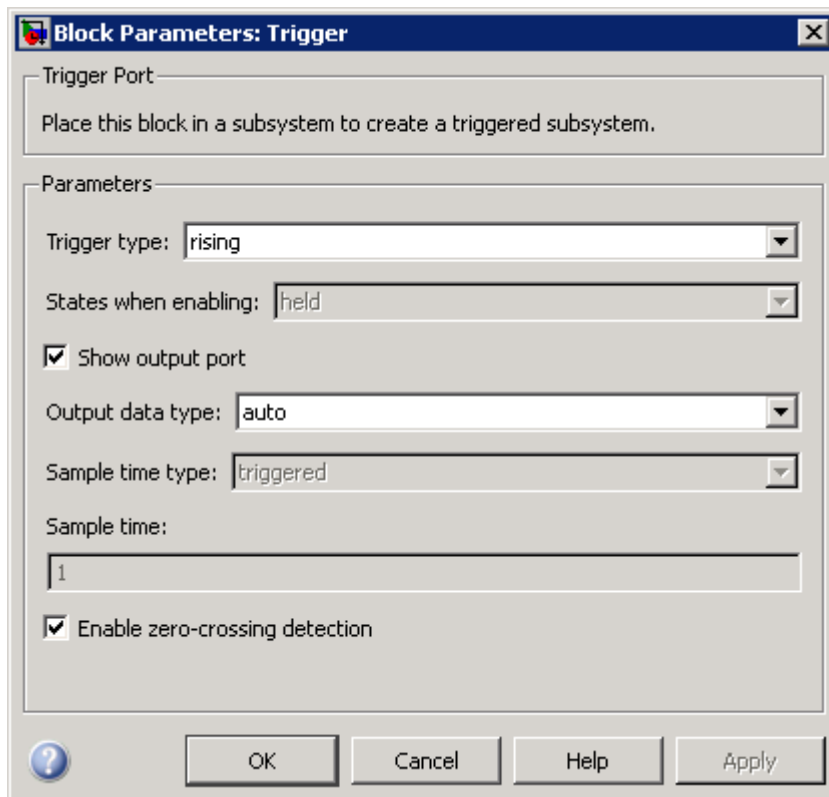
Outputs and States Between Trigger Events

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; states of any discrete blocks are 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, you 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.

Triggered and Enabled Subsystems

In this section...

“What Are Triggered and Enabled Subsystems?” on page 5-19

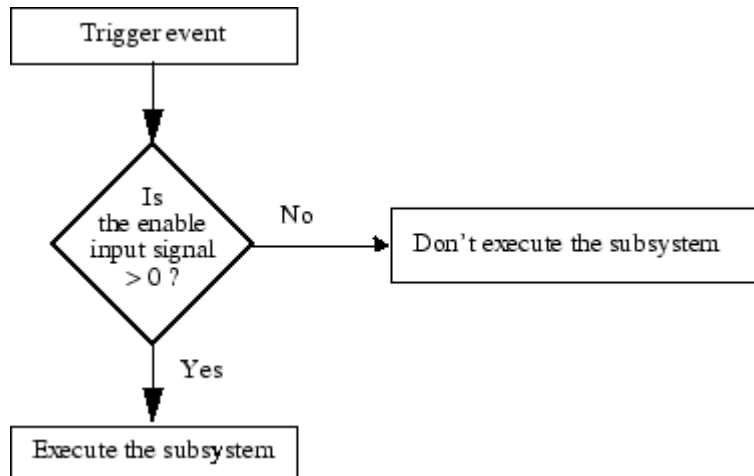
“Creating a Triggered and Enabled Subsystem” on page 5-20

“A Sample Triggered and Enabled Subsystem” on page 5-21

“Creating Alternately Executing Subsystems” on page 5-21

What Are Triggered and Enabled Subsystems?

A third kind of conditional subsystem combines both 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 and enable control inputs to the Subsystem block.



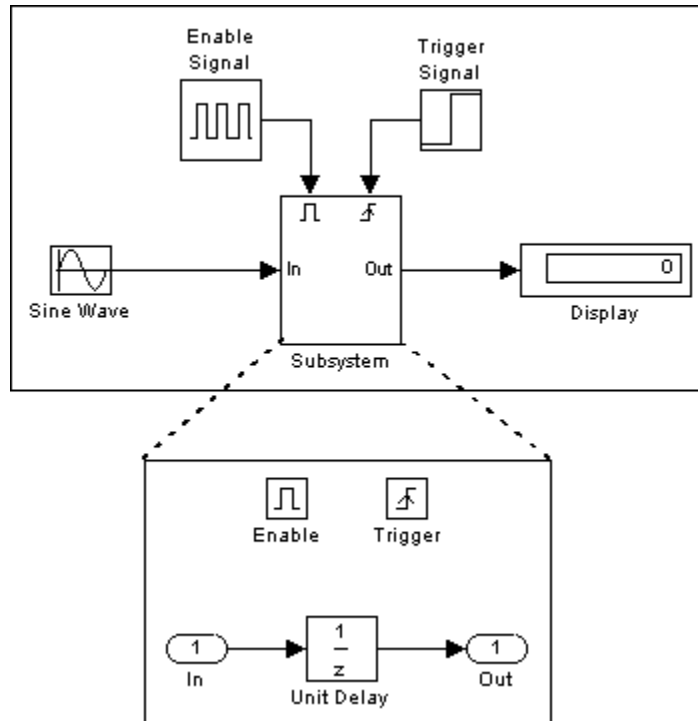
Subsystem

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 5-7. 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 5-8.

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

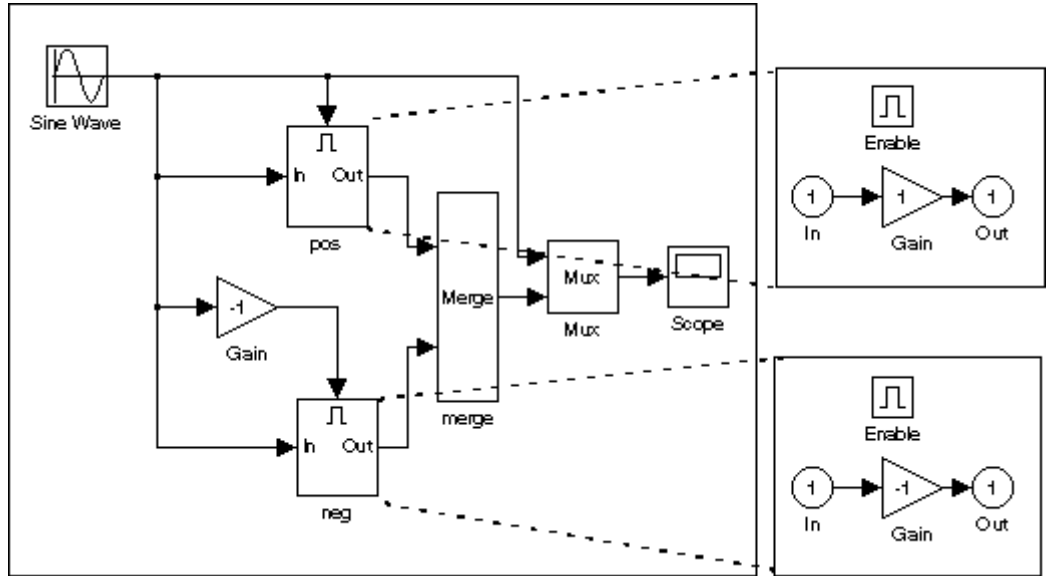
A simple example of a triggered and enabled subsystem is illustrated in the following model.



Creating Alternately Executing Subsystems

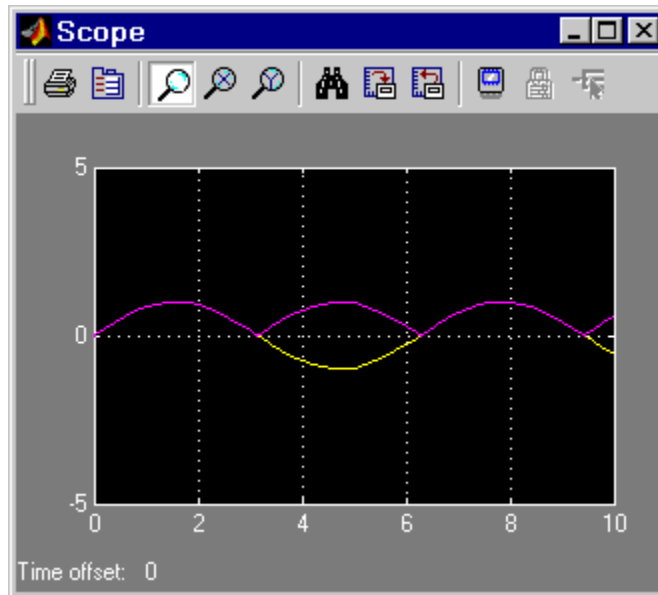
You can use conditional subsystems in combination with Merge blocks to create sets of subsystems that execute alternately, depending on the current state of the model.

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.



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 methods (see “Block Methods” on page 2-17) of the blocks that the subsystem contains in sorted order (see “How Simulink Determines the Sorted Order” on page 7-51). 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’s **Trigger type** to function-call.

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, 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, the simulation is halted and an error message 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 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.

For more information about function-call subsystems, see “Function-Call Subsystems” in “Writing S-Functions” in the online documentation.

Conditional Execution Behavior

In this section...

“What Is Conditional Execution Behavior?” on page 5-25

“Propagating Execution Contexts” on page 5-27

“Behavior for Switch Blocks” on page 5-28

“Displaying Execution Contexts” on page 5-28

“Disabling Conditional Execution Behavior” on page 5-29

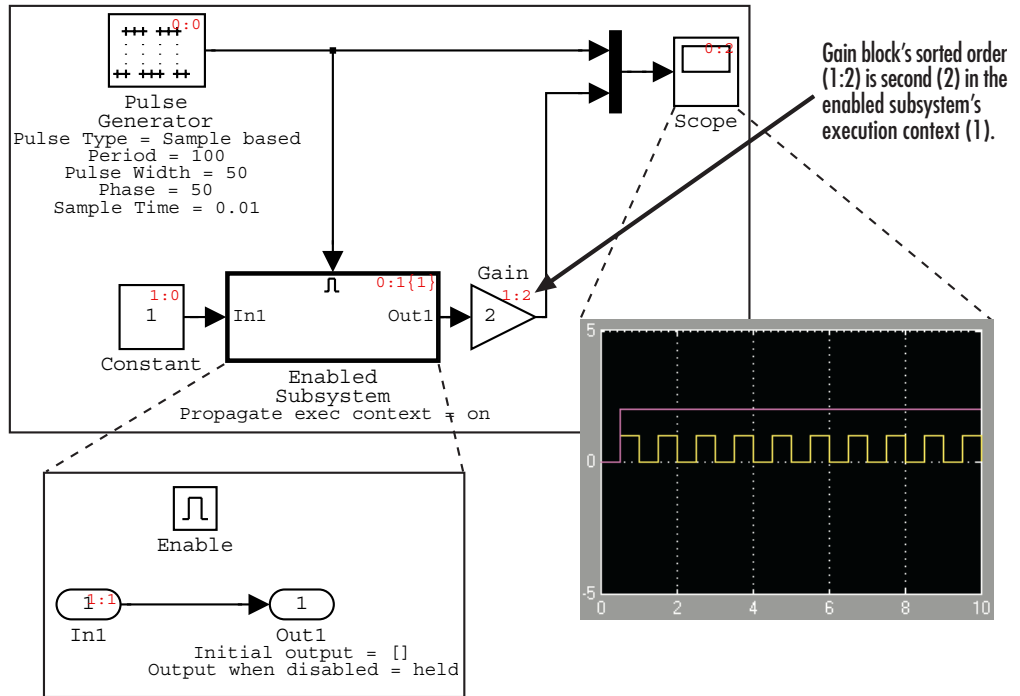
“Displaying Execution Context Bars” on page 5-29

What Is Conditional Execution Behavior?

To speed simulation of a model, by default the Simulink software avoids unnecessary execution of blocks connected to Switch, Multiport Switch, and of conditionally executed blocks, a behavior called *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 5-29.

The following model illustrates conditional execution behavior.

5 Creating Conditional Subsystems



The outputs of the Constant block and Gain blocks are computed only while the enabled subsystem is enabled (that is, at time steps 0.5 to 1.0, 1.5 to 2.0, and so on). This is 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 model's root system, the Simulink software invokes the blocks' 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 model's root level (id = 0).

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:

- Its output is required only by a conditional subsystem or its input changes only as a result of the execution of a conditionally executed.
- The subsystem's execution context can propagate across its boundaries.
- The output of the block is not a testpoint (see "Working with Test Points" on page 10-61).
- The block is allowed to inherit its conditional execution context.

The Simulink software does not allow some built-in blocks, e.g., 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.
- Its sample time is 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 5-29), the Simulink software moves the block into the execution context of the subsystem. This ensures that the block's methods are executed during the simulation loop only when the corresponding conditional subsystem executes.

Note Execution contexts are not propagated to constant sample time blocks.

Behavior for Switch Blocks

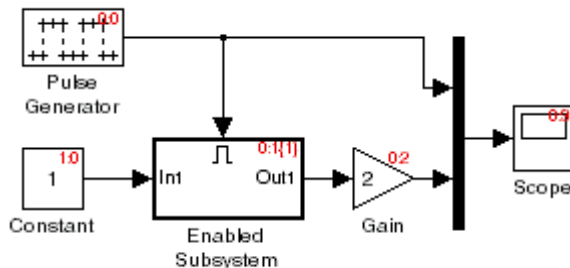
This behavior treats the input branches of a Switch or Multiport Switch block as invisible, conditional subsystems, each of which has its own execution context that is enabled only when the switch's control input selects the corresponding data input. As a result, switch branches execute only when selected by switch control inputs.

Displaying Execution Contexts

To determine the execution context to which a block belongs, select **Sorted order** from the model window's **Format** menu. 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's sorted order in the subsystem's execution context, e.g., 0:0 indicates that the block is the first block in the root subsystem's execution context.

If a bus is connected to the block's input, the block's sorted order is displayed as $s:B$, e.g., 0:B indicates that the block belongs to the root system's execution context 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 model's root system. The first 1 indicates that the enabled subsystem is the second block on

the root system's sorted list (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 Constant block's index, 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

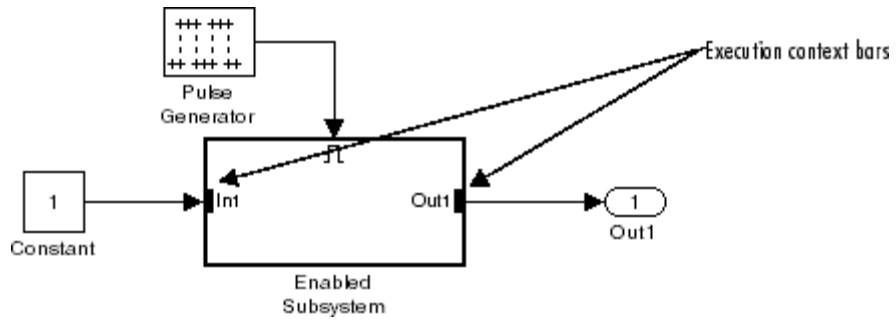
To disable conditional execution behavior for all Switch and Multiport Switch blocks in a model, turn off the **Conditional input branch execution** optimization on the **Optimization** pane of the Configuration Parameters dialog box (see "Optimization Pane"). To disable conditional execution behavior 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, a subsystem's execution context 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, for example, whose initial condition is other than []. In this case, a block connected to the subsystem's output cannot inherit the subsystem's execution context.

Displaying Execution Context Bars

Simulink software can optionally display bars next to the ports of subsystems across which execution contexts cannot propagate, for example, on subsystems from which no block can inherit its execution context.



To display the bars, select **Execution Context Indicator** from model editor's **Format > Block Displays** menu.

Referencing a Model

- “Overview of Model Referencing” on page 6-2
- “Creating a Model Reference” on page 6-7
- “Converting a Subsystem to a Referenced Model” on page 6-10
- “Referenced Model Simulation Modes” on page 6-12
- “Viewing a Model Reference Hierarchy” on page 6-16
- “Model Reference Simulation Targets” on page 6-17
- “Simulink Model Referencing Requirements” on page 6-20
- “Parameterizing Model References” on page 6-27
- “Defining Function-Call Models” on page 6-34
- “Using Model Reference Variants” on page 6-38
- “Protecting Referenced Models” on page 6-63
- “Inheriting Sample Times” on page 6-73
- “Refreshing Model Blocks” on page 6-77
- “Simulink Model Referencing Limitations” on page 6-78

Overview of Model Referencing

In this section...

“About Model Referencing” on page 6-2

“Referenced Model Advantages” on page 6-4

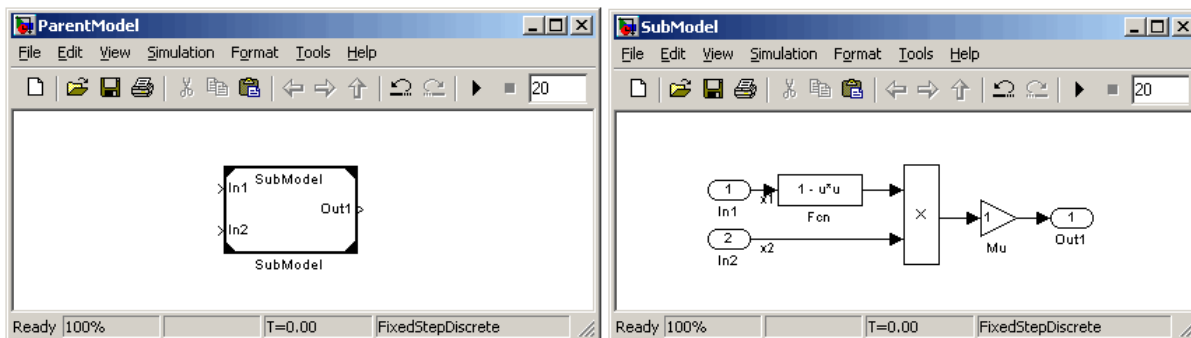
“Model Referencing Demos” on page 6-5

“Model Referencing Resources” on page 6-6

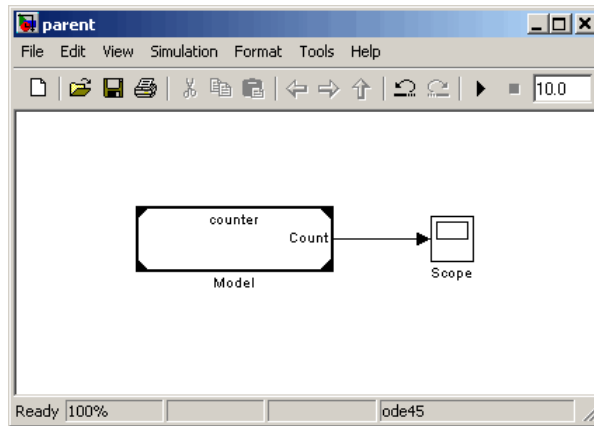
About Model Referencing

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.

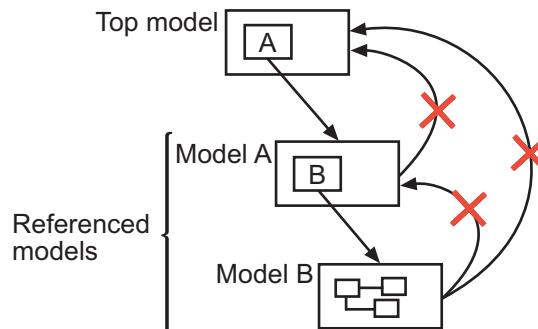
A referenced model’s interface consists of its input and output ports (and trigger port in the case of a function-call model) and its parameter arguments. A Model block displays inputs and outputs corresponding to the root-level inputs and outputs of the model it references. These ports enable you to incorporate the referenced model into the block diagram of the parent model. For example, in the next figure the Model block in the parent model on the left could represent the submodel on the right:



You can 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.



A referenced model can itself contain Model blocks and thus reference lower-level models, and so on to any depth. The topmost model in a hierarchy of referenced models is called the *top model*. 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, as shown in this figure:



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, or let different Model blocks specify different values for variables that define the submodel's behavior. See “Parameterizing Model References” on page 6-27 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 6-12 for details.

You can use any referenced model as a standalone model, provided that it does not depend on any data that is available only from a higher-level model. An appropriately configured model can function as both a standalone model and as a referenced model without requiring any change to the model or to 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 does not load a referenced model until it is needed, 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 code was previously generated. Otherwise, the existing code can be reused.

- **Independent configuration sets**

The configuration set used by a referenced model can differ from that of its parent or other referenced models.

Model Referencing Demos

Simulink includes several demos that illustrate model referencing. To access these demos from the MATLAB command line:

- 1 In the MATLAB Command Window, type

```
demos
```

A list of MATLAB products appears on the left side of the Help window.

- 2 In the left side of the Help window, select **Simulink**.

A list of Simulink demos appears on the right side of the Help window.

- 3 Under Simulink, select **Modeling Features**.

This category contains model referencing demos, including:

- Component-Based Modeling with Model Reference — `sldemo_md1ref_basic`
- Visualizing Model Reference Architectures — `sldemo_md1ref_depgraph`
- Interface Specification Using Bus Objects — `sldemo_md1ref_bus`
- Parameterizing Model Reference — `sldemo_md1ref_paramargs`

- Converting Subsystems to Model Reference — `sldemo_mdhref_conversion`
- Model Reference Function-Call — `sldemo_mdhref_fcncall`

In addition, the demo `sldemo_absbrake` (**Simulink > Automotive Applications > Modeling an Anti-Lock Brake System**) represents a wheel speed calculation as a Model block within the context of an anti-lock braking system (ABS).

Model Referencing Resources

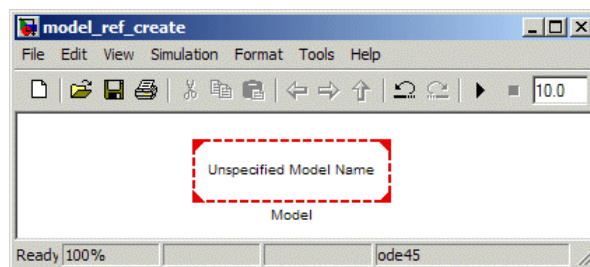
The following are the most commonly needed resources for working with model referencing:

- The Model block, which represents a model that is included as a referenced model in another model. See the Model block parameters in “Ports & Subsystems Library Block Parameters” 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.

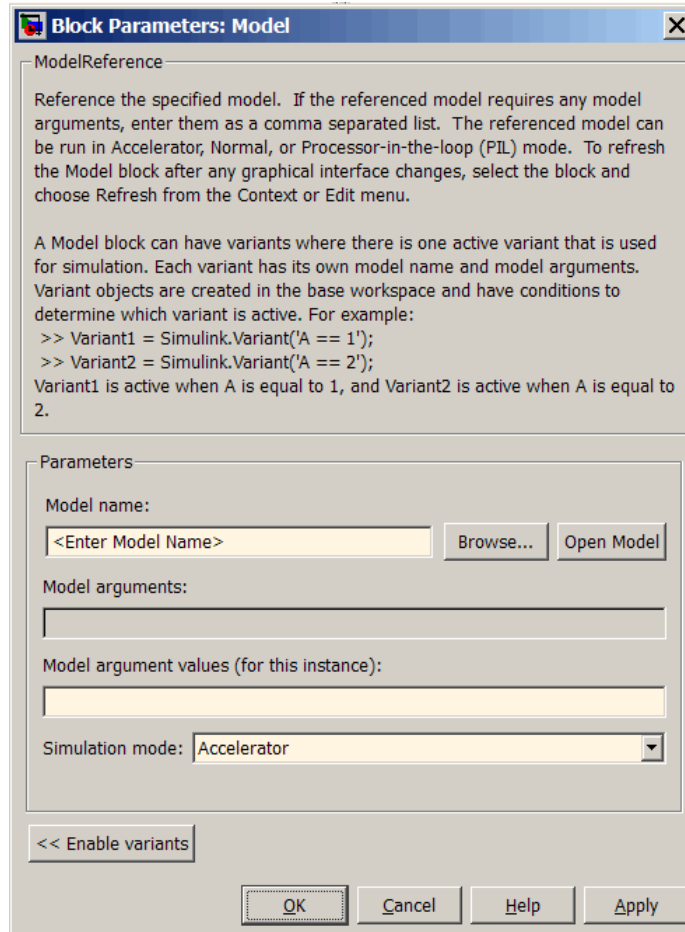
Creating a Model Reference

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. To create a reference to a model (submodel) in another model (parent model):

- 1 If the directory containing the submodel to be referenced is not on the MATLAB path, add the directory 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 hierarchy's top model. See "Inline Parameter Requirements" on page 6-24 for details.
 - Set **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** to **One** if the model will be used at most once in any hierarchy, or to **Multiple** if it will be used more than once. To reduce overhead, specify **Multiple** only when necessary. You can also set the option to **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 6-20.)
 - For information about **Model Arguments** and **Model argument values**, see “Using Model Arguments” on page 6-28.
 - For information about the **Simulation mode**, see “Referenced Model Simulation Modes” on page 6-12.
- 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, and is therefore subject to the restriction that a given signal can have at most one associated signal object. See `Simulink.Signal` and “Multiple Signal Objects” for more information. For information about connecting a bus signal to a referenced model, see “Bus Usage Requirements” on page 6-25.

Converting a Subsystem to a Referenced Model

You can convert any atomic subsystem to a referenced model. The conversion requires the model containing the subsystem to have the following configuration parameter settings:

- **Configuration Parameters > Optimization > Inline parameters** must be On.
- **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** must be Explicit only.
- **Configuration Parameters > Diagnostics > Connectivity > Mux blocks used to create bus signals** must be Error.

After specifying the indicated parameter values, select **Convert to Model Block** from the subsystem's context menu. 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.

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. If an error occurs during the conversion, the outcome depends on the error:

- For some errors, a message box appears that gives you the choice of cancelling or continuing.
- If continuing is impossible, Simulink cancels the conversion without offering a choice to continue.

Even if conversion is successful, you may still need to reconfigure the resulting model to meet your requirements. For example, the **Interpolate data** parameter of each root input port in the new model is selected by default. You can clear the parameter wherever this default is not appropriate. See the Inport block documentation for information about **Interpolate data**.

Once you have successfully created a Model block and referenced model from a subsystem, you can delete the subsystem block from the source model and copy the Model block to the deleted subsystem block's location. All signals

will automatically reconnect. The source model is now a parent model that contains the referenced model.

You can use `Simulink.SubSystem.convertToModelReference` to convert subsystems to model references programmatically. The function provides more capabilities than **Convert to Model Block**, such as the ability to replace a subsystem with an equivalent Model block in a single operation.

Referenced Model Simulation Modes

In this section...
“About Referenced Model Simulation Modes” on page 6-12
“Specifying the Simulation Mode” on page 6-13
“Mixing Simulation Modes” on page 6-14
“Accelerating a Freestanding or Top Model” on page 6-14

About Referenced Model Simulation Modes

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 three modes: Normal, Accelerator, or Processor-in-the-loop (PIL).

Normal Mode

Simulink executes a Normal mode submodel interpretively, as if the submodel were an atomic subsystem implemented directly within the parent model.

Normal mode:

- Requires no delay for code generation or compilation
- Works with most Simulink tools
- Executes slower than Accelerator mode
- Works with only one instance of each model used in a reference hierarchy

Simulation results for a given model are essentially identical in either Normal or Accelerator mode. Trivial differences may occur due to differences in the optimizations and libraries used.

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 6-17 for more information. Accelerator mode:

- Takes time for code generation and compilation
- Does not work with most Simulink tools.
- Executes faster than Normal mode
- Works with multiple instances of each model used in a reference hierarchy

Simulation results for a given model are essentially identical in either Normal or Accelerator mode. Trivial differences may occur due to differences in the optimizations and libraries used.

Processor-in-the-loop (PIL) mode

Simulink executes a PIL-mode referenced model by:

- Creating both a MEX-file and a model reference target for the submodel.
- Executing cross-compiled object code on a target processor or an equivalent instruction set simulator.

Simulink PIL mode:

- Is designed for verifying deployment code on target processors
- Requires Real-Time Workshop® Embedded Coder™ software
- Works with only one branch (top model and subordinates) in a hierarchy

For more information, see “Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation.

Specifying the Simulation Mode

The Model block for each instance of a referenced model controls its simulation mode. The default referenced model simulation mode is Accelerator mode. To set or change a submodel’s simulation mode:

- 1 Access the Model block’s parameter dialog box. (See “Navigating a Model Block”.)

2 Set the **Simulation mode** field to Normal, Accelerator, or Processor-in-the-loop (PIL).

3 Click **OK** or **Apply**.

Mixing Simulation Modes

When a parent model executes in Normal mode, it can contain Normal mode, Accelerator mode, and PIL mode submodels. At most one instance of a given model can execute in Normal mode. A submodel can execute in Normal mode *only* 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.

When a parent model executes in Accelerator mode, 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 PIL-mode model is subordinate to an Accelerator mode model, an error occurs.

When a parent model executes in PIL mode, all subordinate models also execute in PIL mode regardless of the simulation mode specified by their Model blocks. The PIL mode Model block uses the model reference targets of the blocks beneath. Only one branch (top model and all subordinates) in a model reference hierarchy can execute in PIL mode. For more information, see “Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation, and “Creating Model Components” in the Real-Time Workshop documentation.

Accelerating a Freestanding or Top Model

You can use Simulink Accelerator mode (see Chapter 27, “Accelerating Models”) or Rapid Accelerator mode (see “Testing and Refining Concept Models With Standalone Rapid Simulations”) to achieve faster execution of any Simulink model, including a top model in a model reference hierarchy.

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 an error 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 Chapter 27, “Accelerating Models”
- Rapid Accelerator mode execution of a freestanding or top model, as described in “Testing and Refining Concept Models With Standalone Rapid Simulations”.

While the different types of acceleration share many capabilities and techniques, they are implemented differently, and have different requirements and limitations.

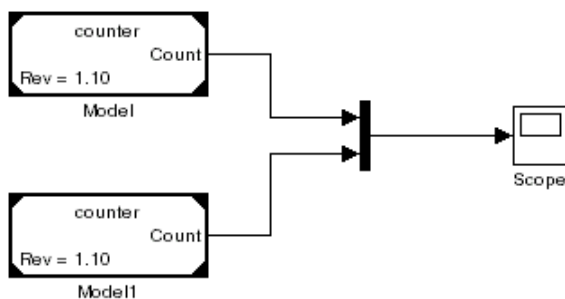
Viewing a Model Reference Hierarchy

Simulink provides tools and functions that you can use to examine a model reference hierarchy:

- **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 “Using the Model Dependency Viewer” on page 19-52 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.

Displaying Version Numbers

To display the version numbers of the models referenced by a model, choose **Model block version** from the **Block displays** submenu of the parent model’s **Format** menu. 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 create the block, or used most recently to refresh the block. See “Managing Model Versions” on page 4-99 and “Refreshing Model Blocks” on page 6-77 for more information.

Model Reference Simulation Targets

In this section...
“About Simulation Targets” on page 6-17
“Building Simulation Targets” on page 6-18

About 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.

Be careful not to confuse a submodel’s simulation target with any of these other types of target:

- Hardware target — A platform for which Real-Time Workshop generates code
- System target — A file that tells Real-Time Workshop how to generate code for particular purpose
- Rapid Simulation target (RSim) — A system target file supplied with Real-Time Workshop
- Model reference target — A library module that contains Real-Time Workshop 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 instance of a submodel executes in Normal mode, and one or more instances execute in Accelerator mode, Simulink creates a simulation target for the Accelerator mode instance(s), but the Normal mode instance does not use it.

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 6-81 for details.

Building Simulation Targets

If a simulation target does not exist at the beginning of a simulation, or when you update a parent model’s block diagram, Simulink by default generates the needed target from the referenced model. If the simulation target already exists, Simulink by default checks whether the submodel has changed significantly since the target was last generated. If so Simulink by default regenerates the target to reflect changes in the model.

You can change this default behavior to modify the rebuild criteria or specify that Simulink always or never rebuilds targets. See “Rebuild options” for details. To generate simulation targets interactively for Accelerator mode referenced models, update the model’s diagram or 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.

Simulink creates simulation targets in a subdirectory of the working directory. This subdirectory is named `slprj`. If `slprj` does not exist, Simulink creates it. Subdirectories in `slprj` provide separate places for simulation code, Real-Time Workshop code, and other files.

Reducing Change Checking Time

You can reduce the time that Simulink spends checking whether any or all simulation targets need to be rebuilt 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”.)

- For referenced models you want to minimize change checking time, set **Configuration Parameters > Model Referencing > Rebuild options** to **If any changes in known dependencies detected on the top model**. See “Rebuild options”.

These parameter values exist in a referenced model’s configuration set, 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.

Simulink Model Referencing Requirements

In this section...
“About Model Referencing Requirements” on page 6-20
“Name Length Requirement” on page 6-20
“Configuration Parameter Requirements” on page 6-20
“Model Structure Requirements” on page 6-25

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 “Simulink Model Referencing Limitations” on page 6-78.

Name Length Requirement

The name of a referenced model must contain fewer than 60 characters, exclusive of the .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 “Setting Up Configuration Sets” on page 20-2. By default, every model in a hierarchy has its own configuration set, which it uses in 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. Simulink’s 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. You must then change some or all parameter values to eliminate the problem.

When a model reference hierarchy contains many submodels that have incompatible parameter values, or a changed parameter value must propagate to many submodels, manually eliminating all configuration parameter incompatibilities can be tedious. You can control or eliminate such overhead by using configuration references to assign an externally stored configuration set to multiple models. See “Referencing Configuration Sets” on page 20-12 for details.

Note Configuration parameters on the Real-Time Workshop pane of the Configuration Parameters dialog do not affect simulation in either Normal or Accelerated mode. Real-Time Workshop parameters affect only code generation by Real-Time Workshop itself. Although Accelerated mode simulation requires code generation to create a simulation target, Simulink uses default values for all Real-Time Workshop 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, or if set 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 the top model's Stop time for simulation, overriding any differing Stop time in a submodel.
	Type Solver	The top model's Type and Solver apply throughout the hierarchy. See "Solver Requirements" on page 6-23.
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 6-24.
	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 6-24.
Hardware Implementation	Embedded 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, as shown in the following table:

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 2-22 and “Choosing a Solver” on page 21-9.

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 inlined 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. Consequently, you cannot use this dialog box to override the inline parameters optimization for selected parameters and thereby permit them to be tuned. “Parameterizing Model References” on page 6-27 describes 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. **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** provides this specification. See “Total number of instances allowed per top model” for more information. The possible values for this parameter are:

- **Zero** — The model cannot be referenced. An error occurs if a reference to the model occurs in another model.
- **One** — The model can be referenced at most once in a model reference hierarchy. An error occurs if more than one instance exists. This value may be preferable or required.
- **Multiple** — The model can be referenced more than once in a hierarchy, provided that 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, but does not affect data values that result from simulation or from executing code generated by Real-Time Workshop. Specifying **Multiple** when only one model instance exists facilitates later reusing the model in the same hierarchy, or multiple times in a different hierarchy, without having to change or rebuild the model.

Some model properties and constructs require setting **Total number of instances allowed per top model** to **One**, limiting the model to being used only once in a hierarchy. For details, see “General Reusability Limitations” on page 6-79 and “Accelerator Mode Reusability Limitations” on page 6-82.

Model Structure Requirements

The following requirements relate to the structure of a model reference hierarchy independently of configuration parameter requirements.

Signal Propagation Requirements

The signal name must explicitly appear on any signal line connected to an Output of a referenced model. A signal connected an unlabeled line to an Output of a referenced model cannot propagate out of the Model block to the parent model.

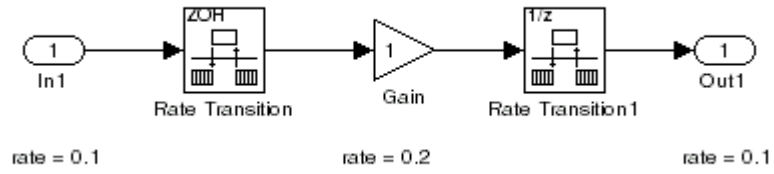
Bus Usage Requirements

A bus that propagates between a parent model and a referenced model must specify the properties of the bus in both the parent and the referenced model using a bus object. This object must be defined in the MATLAB workspace. See for more information.

Sample Time Requirements

The first nonvirtual block connected to a root-level Inport or Output of a referenced model must have the same sample time as the port to which it connects. You can use Rate Transition blocks to match input and output sample times as illustrated in the following diagram.

6 Referencing a Model



Parameterizing Model References

In this section...
“Introduction” on page 6-27
“Global Nontunable Parameters” on page 6-27
“Global Tunable Parameters” on page 6-28
“Using Model Arguments” on page 6-28

Introduction

A parameterized referenced model obtains values that affect the referenced model’s behavior from some source outside the model. Changing the values changes the behavior of the model without requiring the model to be recompiled.

Due to the constraints described in “Inline Parameter Requirements” on page 6-24, you cannot use the Model Parameter Configuration dialog box to parameterize referenced models. The Simulink software 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.

Global Nontunable Parameters

A *global nontunable parameter* is a MATLAB variable or a Simulink.Parameter object whose storage class is auto. The parameter

can exist in the MATLAB workspace or any model workspace visible to all referenced models that use the parameter.

Using a global nontunable parameter in a referenced model allows you to control the behavior of the referenced model by setting the parameter value before simulation begins. 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 “Specifying Numeric Parameter Values” on page 7-17 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 “Changing the Values of Block Parameters During Simulation” on page 7-23 for details.

If you want to reference an existing model that uses tunable parameters defined with the “Model Parameter Configuration Dialog Box”, you must change the model to implement tunability in some other 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, you create MATLAB variables in the model workspace, then add the variables to a list of model arguments associated with the model. You can then 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 then exists to provide model argument values, and the model uses the values of the MATLAB variables themselves, as defined in the model workspace. Thus the same model can be used without change as a top model, a standalone model, and a parameterized referenced model.

The demo model `sldemo_mdhref_paramargs` demonstrates techniques for using model arguments. The demo 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 for information about masking.

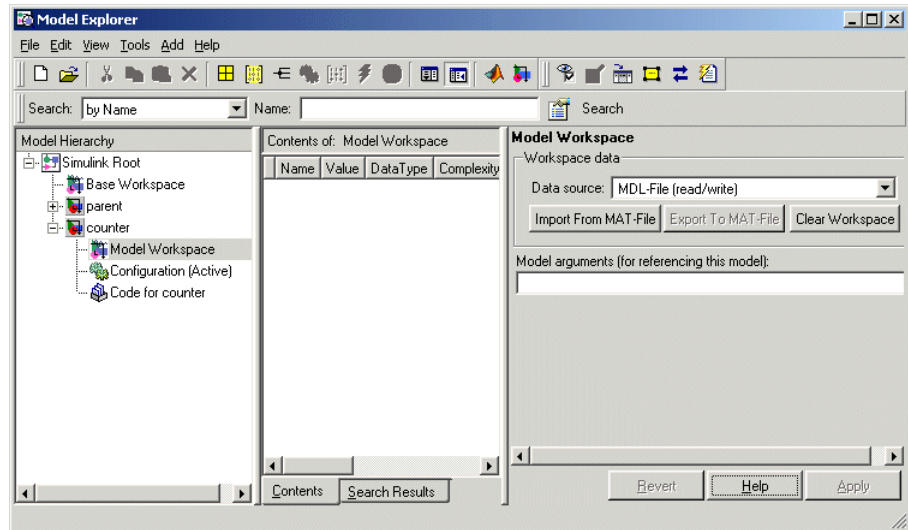
The rest of this section describes techniques for declaring and using model arguments to parameterize a referenced model independently of any Model block masking. The steps are:

- 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 that will be used as model arguments:

- 1** Open the model for which you want to define model arguments.
- 2** Open the Model Explorer.
- 3** Select the model's workspace in Model Explorer's **Model Hierarchy** pane:



- 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 that the variable should have 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.

Registering the Model Arguments

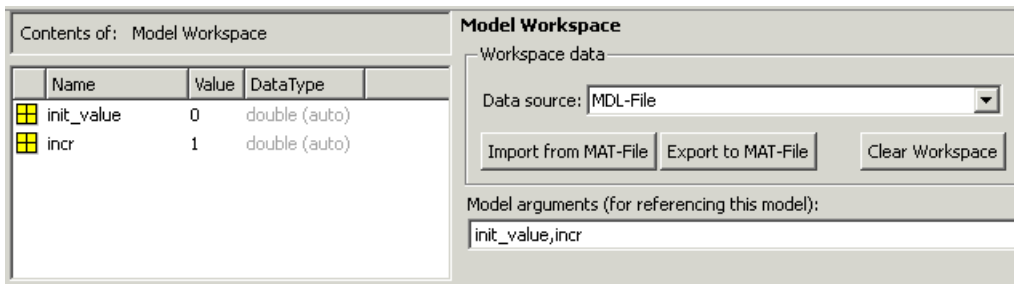
To register MATLAB variables as model arguments:

- 1 Again select the model's workspace in Model Explorer's **Model Hierarchy** pane.

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 as a comma-separated list in the **Model arguments** field.

For example, if you added two MATLAB variables named `init_value` and `incr`, and declared them to be model arguments, the Contents and Dialog panes of the Model Explorer could look like this:

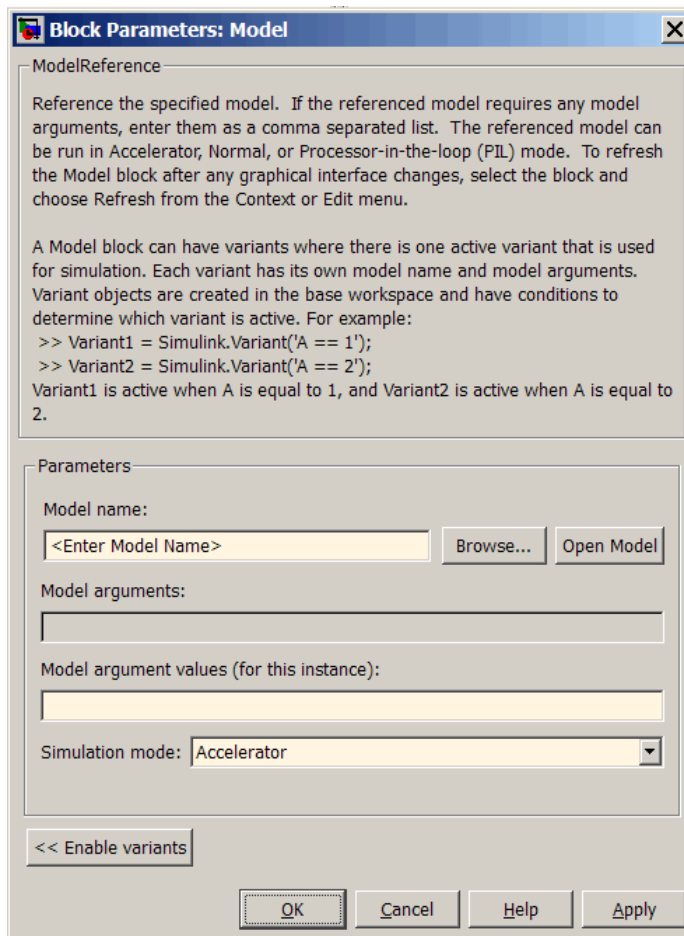


- 3 Click **Apply** to confirm the entered names.

Assigning Model Argument Values

If a model declares model arguments, you must 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 a referenced model's arguments:

- 1 Open the Model block's parameter dialog box by right-clicking the block and choosing **Model Reference Parameters** 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. The values are assigned 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 “Hierarchical Symbol Resolution” on page 4-76. All values must be numeric (including objects with numeric values).

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** will have the values specified in the Model block’s **Model argument values** field.

Defining Function-Call Models

In this section...

“About Function-Call Models” on page 6-34

“Function-Call Model Demo” on page 6-34

“Creating a Function-Call Model” on page 6-34

“Referencing a Function-Call Model” on page 6-35

“Function-Call Model Requirements” on page 6-36

About Function-Call Models

Simulink allows certain blocks, such as a Function-Call Generator or an appropriately configured custom S-function, to control execution of a referenced model during a time step, using a function-call signal. See “Function-Call Subsystems” on page 5-24 for more information. A referenced model capable of being invoked in this way is called a *function-call model*.

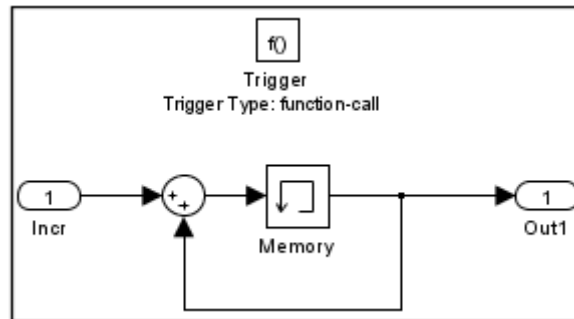
Function-Call Model Demo

To view a function-call model demo, select **Simulink > Modeling Features > Model Reference > Model Reference Function-Call** from the **Demos** pane of the MATLAB Help Browser, or execute `sldemo_mdref_fncall` at the MATLAB command line.

Creating a Function-Call Model

To create a function-call model:

- 1 Insert a Trigger block at the root level of the model.
- 2 Set the Trigger block’s **Trigger type** parameter to `function-call`.
- 3 Create and connect any other blocks required to implement the model.



- 4 Ensure that the model satisfies the conditions imposed on function-call models. See “Function-Call Model Requirements” on page 6-36 for details.

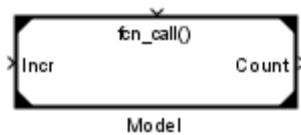
You can now simulate the function-call model either by itself or by running a model that references the function-call model directly.

Referencing a Function-Call Model

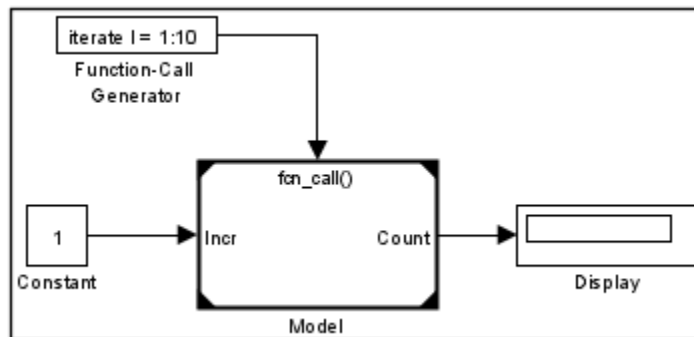
To create a reference to a function-call model:

- 1 Create a Model block in the referencing model that references the function-call model. See “Creating a Model Reference” on page 6-7 for details.

The top of the Model block displays a function-call port corresponding to the function-call trigger port in the function-call model.



- 2 Connect a Stateflow chart, Function-Call Generator block, or other function-call-generating block to the Model block’s function-call port. The signal connected to the port must be scalar.
- 3 Connect the Model blocks inputs and outputs if any to the appropriate blocks in the parent model.



- 4 Create and connect any other blocks required to implement the parent model.
- 5 Ensure that the referencing model satisfies the conditions for a model to reference other models. See “Simulink Model Referencing Requirements” on page 6-20 and “Simulink Model Referencing Limitations” on page 6-78 for details.

You can now simulate the model that references the function-call model.

Function-Call Model Requirements

To be a function-call model, a referenced model must meet the following requirements in addition to the requirements that every referenced model must meet.

- A function-call model cannot have an output that is 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 inserted block’s **Override optimizations and always copy signal** option.
- If the function-call model specifies a fixed-step solver and contains one or more blocks that use absolute or elapsed time, the referencing model must trigger the function-call model at the rate specified by the 'Fixed-step size' option on the **Solver** page of the **Configuration Parameters**

dialog. 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 Model block's function-call port must be scalar.

Using Model Reference Variants

In this section...

“Adapting Models to Different Contexts” on page 6-38

“About Model Reference Variants” on page 6-39

“Model Reference Variant Components” on page 6-41

“Model Reference Variant Example” on page 6-42

“Model Reference Variant Requirements” on page 6-46

“Implementing Model Reference Variants” on page 6-47

“Saving Variant Referenced Models” on page 6-57

“Changing Variant Model Definitions” on page 6-57

“Parameterizing Variant Models” on page 6-58

“Reusing Variant Objects and Models” on page 6-59

“Overriding Variant Conditions” on page 6-59

“Variant Models and Enumerated Types” on page 6-60

“Generating Code for Variant Models” on page 6-61

“Model Reference Variant Limitations” on page 6-61

Adapting Models to Different Contexts

Many situations require the ability to customize a model to fit different contexts without having to replace the model entirely. For example, suppose a design engineer has a model that simulates a car on the road. Different versions of the car may have much in common yet differ in specific ways, such as using different fuels, having different size engines, or meeting different emission standards.

If each permutation required a separate model, the resulting combinatorial explosion could be unmanageable. Simulink therefore provides several techniques that you can use to customize a model to fit different contexts without needing to replace the model entirely. For example, you can:

- Change base workspace data values, as described in “Tunable Parameters” on page 2-9
- Select a subsystem from a library by using a Configurable Subsystem block.
- Use a mask to change a subsystem, as described in “Creating Dynamic Masked Subsystems” on page 9-55
- Change the arguments to a parameterized referenced model, as described in “Parameterizing Model References” on page 6-27

Where these techniques do not provide sufficient adaptability, you can configure any Model block to select the model that it references from a predefined set of models. The Model block selects the model to reference based on current values in the base workspace. This section describes the necessary Simulink techniques.

About Model Reference Variants

Model reference variants allow you to configure any Model block to select its referenced model from a set of candidate models. A model used as a model reference variant needs no special properties: functionally it is just another referenced model.

- Simulink selects the variant to use when you compile the model that contains the Model block.
- The selection depends on the values of one or more MATLAB variables and/or Simulink parameters in the base workspace.

To configure a Model block to select the model that it references, you:

- Provide a set of Boolean expressions that reference base workspace values.
- Associate each expression with one of the models that the block could reference.

The Boolean expressions must be organized so that one and only one expression is true for any combination of base workspace values. When you compile the model, Simulink evaluates all the expressions. Each Model block that uses model reference variants then selects the candidate model whose associated expression is true, and ignores all the other models. Compilation

then proceeds just as if you had entered the name of the selected model literally in the Model block's **Model name** field.

When a candidate model is parameterized, as described in “Parameterizing Model References” on page 6-27, you can associate model argument values with the object whose Boolean expression selects the model. When that expression is `true`, the Model block provides the selected model with those argument values, just as it would if the values appeared literally in the Model blocks's **Model argument values** field. You can also specify whether the selected model executes in Accelerator, Normal, or PIL mode.

The Boolean expressions that control model reference variants reside in Simulink objects in the base workspace. Once you define such an object, you can reuse it with as many Model blocks as needed, so that all the Model blocks select the model that each references based on the same criteria. You can also use the same model as a variant more than once in a Model block, predicating each use on a different Boolean expression, and configuring the model differently for each use.

Reusing expressions and models allows you to globally reconfigure a model hierarchy to suit different environments by doing nothing more than changing base workspace values. No matter how many simulation environments you define, selecting an environment requires only setting variable or parameter values appropriately in the base workspace.

Most Simulink capabilities and tools work with variant models in the same way that they do with ordinary referenced models. The few exceptions are listed in “Model Reference Variant Limitations” on page 6-61. You can nest Model blocks that use variants to any level, provided that the resulting hierarchy satisfies all applicable requirements and limitations when you compile the model or generate code for it.

To see demos that show how to define model reference variants in Simulink and use them with enumerated types, access MATLAB Online Help and select:

- Simulink > Demos > Modeling Features > Model Reference > Model Reference Variants
- Simulink > Demos > Modeling Features > Model Reference > Model Reference Variants Enumerations and Reuse

To see a demo that shows how to define model reference variants in Simulink and use them for simulation and code generation, access MATLAB Online Help and select either of the following:

- Real-Time Workshop > Demos > Variants > Code Variants Using Model Blocks
- Real-Time Workshop Embedded Coder > Demos > Variants > Code Variants Using Model Blocks

For complete information about generating code for variant referenced models, see “Generating Code Variants for Variant Models”. Reference information about the model reference variants API appears in the documentation of the Model block and the `Simulink.Variant` class.

Model Reference Variant Components

To configure a model to use model reference variants, you must provide the following:

- **Variant control variables** — Base workspace variables or parameters that you set to specify the environment in which you want to simulate the model.
- **Variant objects** — Base workspace objects that contain Boolean expressions called **variant conditions**. Each variant object corresponds to a different simulation environment. See `Simulink.Variant` for reference information.
- **Variant models** — A set of candidate models that a particular Model block can reference. Each variant model has at least one associated variant object.

Before simulating, you set the variant control variables to values that describe the environment in which you want to simulate. When compiling the model, Simulink tests the variant condition specified by each variant object. The variant model whose variant object’s condition is `true` becomes the **active variant**, which the Model block references during simulation. A Model block that uses model reference variants can be called a **variant Model block**.

Model Reference Variant Example

Suppose that an automobile can use either a diesel or a gasoline engine, and that the engine must meet either the European or American (USA) emission standard. Other parts of the automobile might also change to reflect the applicable fuel and emission standards, but they do not appear in this example.

Suppose that the Simulink model for the described automobile includes a Model block named Engine that uses model reference variants to select one of four referenced models, each representing one permutation of engine fuel and emission standards.

This section shows one way to specify the necessary variant control variables, variant objects, and variant models, and describes how those components work together. The next section, “Implementing Model Reference Variants” on page 6-47, shows you how to construct the example used in this section.

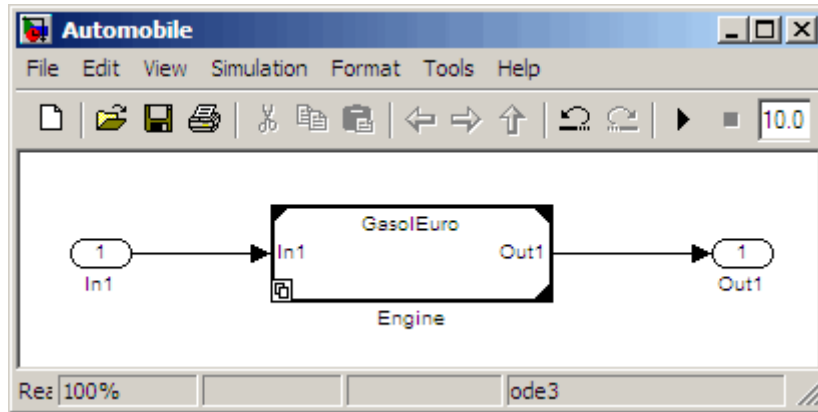
Opening the Example


To see the example, open the model `Automobile.mdl` by doing one of the following

- Click `Automobile.mdl`
- In the MATLAB command window, enter:

```
cd([docroot ' /toolbox/simulink/ug/examples/variants/mdlref']);  
Automobile
```

The example initially looks like this.



The icon  on the lower left corner of the Model block indicates that the block uses model reference variants. The name at the top of the block is the name of the active variant. When you open the model, a callback function loads an M-file that populates the base workspace with the variables and objects that the model uses. The base workspace then contains these definitions:

Name	Value
EMIS	1
FUEL	2

Example Variant Control Variables

The example uses two variant control variables: FUEL and EMIS. The values used for the variables in the example, and the (arbitrarily assigned) meanings of those values, are:

Variant Control Variable	Value	Meaning
FUEL	1	gasoline
	2	diesel
EMIS	1	American
	2	European

The current variable values, as shown in the base workspace, specify that the fuel is diesel (FUEL==2), and the emission standard is American (EMIS==1). Techniques for defining variant control variables appear in “Implementing Variant Control Variables” on page 6-48.

Example Variant Objects

The example uses four variant objects, one for each of the four environments defined by the permutations of fuel and emission standards. The variant conditions and the objects that contain them are:

Variant Condition	Variant Object
GU (FUEL==1 && EMIS==1)	GU
GE (FUEL==1 && EMIS==2)	GE
DU (FUEL==2 && EMIS==1)	DU
DE (FUEL==2 && EMIS==2)	DE

Because the fuel is diesel (2) and the emission standard is American (1), the third variant condition, in the variant object DU, is true, and the rest are false. Techniques for defining variant control variables appear in “Implementing Variant Objects” on page 6-49.

Example Variant Models

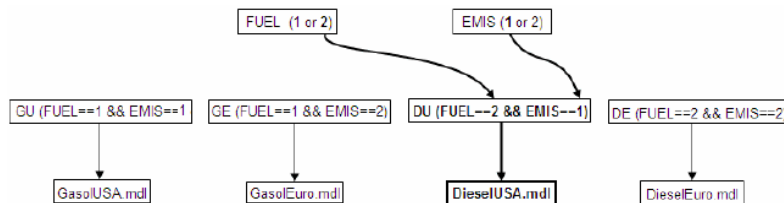
The example contains a Model block named Engine that can reference one of four variant models, named GasolUSA, GasolEuro, DieselUSA, and DieselEuro. Each of the models is associated with a variant object as follows:

Variant Object	Variant Model
GU	GasolUSA.mdl
GE	GasolEuro.mdl
DU	DieselUSA.mdl
DE	DieselEuro.mdl

Because the variant condition associated with the variant object DU is true, the selected variant model is DieselUSA.mdl, which is correct for the specified fuel (diesel) and emission standard (American). The variant models themselves do not matter to this example, so each is a shell containing a callback that displays the model's name when it executes. The variant models are siblings of Automobile.mdl, so you can use the **File > Open** command to access and inspect them. Techniques for associating variant conditions with variant models appear in “Enabling Model Variants” on page 6-51 and “Specifying Variant Models” on page 6-54.

Variant Components Relationship

This figure shows how the components of a variant model implementation work together to specify which variant a Model block will use for simulation. The figure uses the objects and values that appear in the example, which select DieselUSA.mdl as the active variant. The principles are the same for all model reference variants implementations.

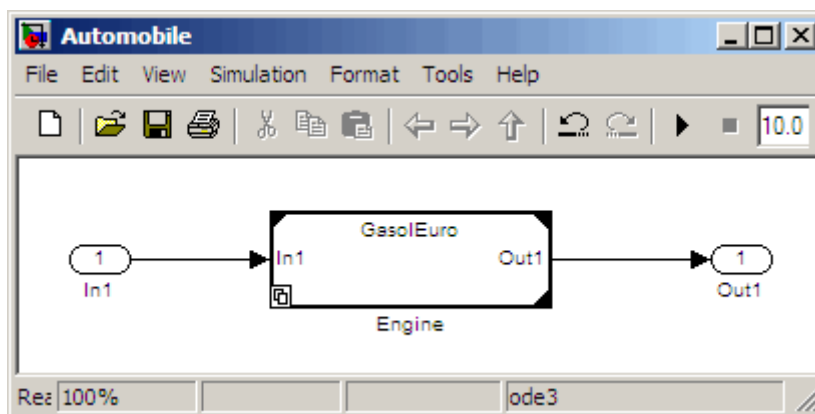


Running the Example

Expose the MATLAB command window so you can see the messages it displays, then simulate the model `Automobile`. Because `FUEL` is 2 and `EMIS` is 1, the variant condition `FUEL==2 && EMIS==1` is true, and the other three variant conditions are false. The true condition resides in the variant object `DU`, which is associated in the Model block with the model `DieselUSA`. The Model block therefore uses `DieselUSA` as its referenced model, and simulating `Automobile` runs a callback that displays:

```
Using variant model: DieselUSA.mdl
```

The top line in the Model block changes to display the name of the variant model that it used:



Now change `EMIS` to 2 in the base workspace and simulate the model again. The values `FUEL==2 && EMIS==2` make the variant condition in `DE` evaluate to true, selecting the variant model `DieselEuro.mdl`. The simulation therefore displays `Using variant model: DieselEuro.mdl`. The top line in the Model block reads `DieselEuro`.

Model Reference Variant Requirements

A Model block that uses variant models, and every model used as a variant, must meet the following requirements:

- All variant models must exist on the MATLAB path (though not necessarily in the same directory).
- Variant control variables must be of a scalar noncomplex Boolean, integer, or enumerated data type.

You can enable or suppress warning messages about mismatches between a Model block and its referenced model by setting diagnostics on the “Diagnostics Pane: Model Referencing”.

Model reference variants must also satisfy the requirements listed in “Simulink Model Referencing Requirements” on page 6-20 and the limitations listed in “Simulink Model Referencing Limitations” on page 6-78 and “Model Reference Variant Limitations” on page 6-61. Additional requirements and limitations that apply to code generation only appear in “Generating Code Variants for Variant Models”.

Implementing Model Reference Variants

This section shows you how to use Simulink capabilities to implement model reference variants. See “Model Reference Variant Requirements” on page 6-46 for a list of requirements that a Model block and all of its variant models must satisfy. The order of operations shown in this section is not required: any other order that provides the same specifications before model compilation occurs would do as well. The necessary steps, listed in the order described in this section, are:

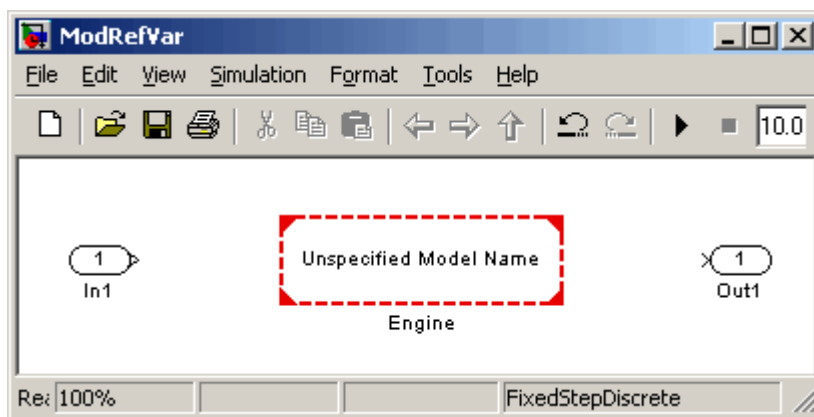
- Implement variant control variables
- Implement variant objects
- Enable model variants
- Specify variant models

This section illustrates the described operations using the model `Automobile.mdl` shown in “Model Reference Variant Example” on page 6-42. You can read the section and apply the operations to your own model, or you can use the section as a tutorial, as follows:

- 1 Close `Automobile.mdl` if it is open, which automatically clears its variables and objects from the base workspace. Otherwise the leftover definitions would interfere with the operations described in this section.
- 2 Open the model `ModRefVar.mdl` by clicking `ModRefVar.mdl` or executing:

```
run([docroot ' /toolbox/simulink/ug/examples/variants/mdlref/ModRefVar.mdl'])
```

The model initially looks like this:



- 3 Save the model in a writable working directory.
- 4 Follow the instructions in the rest of this section to implement the capabilities shown in `Automobile.mdl`.

You do not need to copy the four variant model files used in the example, because they are already on the MATLAB path and need no changes. The variant models are siblings of `ModRefVar.mdl`, so you can use the **File > Open** command to access and inspect them.

Implementing Variant Control Variables

A variant control variable used in simulation can be a MATLAB variable, or a `Simulink.Parameter` object that resides in the base workspace. A variant control variable used for code generation must be a Simulink parameter that meets the requirements described in “Generating Code Variants for Variant Models”. You can define control variables in the MATLAB Command

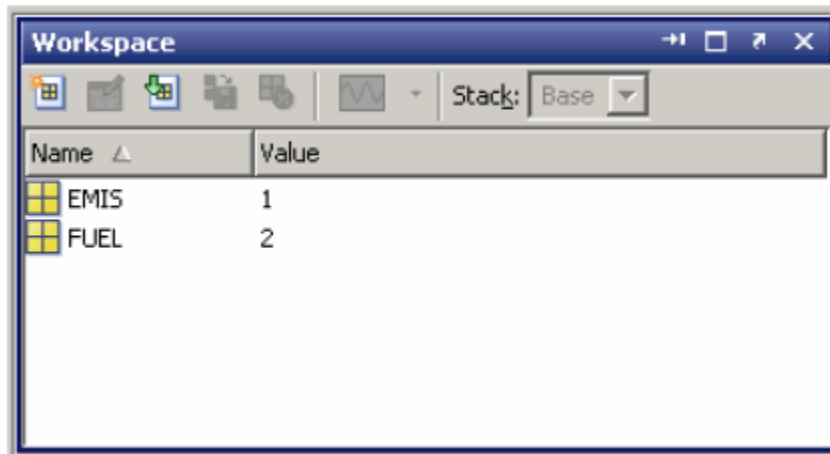
Window or the Model Explorer. For example, to define the variables used in `Automobile.mdl`, do one of the following:

- **Technique 1:** In the MATLAB Command Window, enter (or paste from this page):

```
EMIS = 1
FUEL = 2
```

- **Technique 2:** In the Model Explorer, select the **Base Workspace**, then use **Add MATLAB Variable** to define each variant control variable.

Whichever technique you use, the base workspace should look like this when you are done:



Implementing Variant Objects

A variant control object is an instance of class `Simulink.Variant` that exists in the base workspace and specifies a variant condition. The object can have any unique legal name. A variant condition can include scalar variables, enumerated values, the operators `==`, `!=`, `&&`, `||`, and `~`, and parentheses if needed for grouping.

You can define variant objects in the MATLAB Command Window or the Model Explorer. For example, to define the objects used in `Automobile.mdl`, do one of the following:

- **Technique 1:** In the MATLAB Command Window, enter (or paste from this page):

```
GU=Simulink.Variant('FUEL==1 && EMIS==1')
GE=Simulink.Variant('FUEL==1 && EMIS==2')
DU=Simulink.Variant('FUEL==2 && EMIS==1')
DE=Simulink.Variant('FUEL==2 && EMIS==2')
```

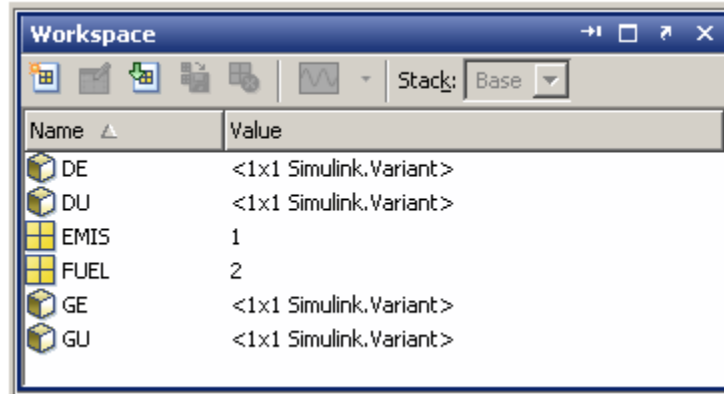
- **Technique 2:** In the Model Explorer:

- 1** Select the Base Workspace.
- 2** Choose **Add > Simulink.Variant**

A new variant object named `Variant` appears in the base workspace.

- 3** Use Contents (center) pane **Name** field to change the name of a variant object to `GU`.
- 4** Use the Dialog (left) pane to enter the associated variant condition: `FUEL==1 && EMIS==1`. *Do not* surround the condition with parentheses or single quotes, which are required only by the MATLAB API.
- 5** Repeat the instructions to define all needed variant objects and their variant conditions.

Whichever technique you use, the base workspace should look like this when you are done:

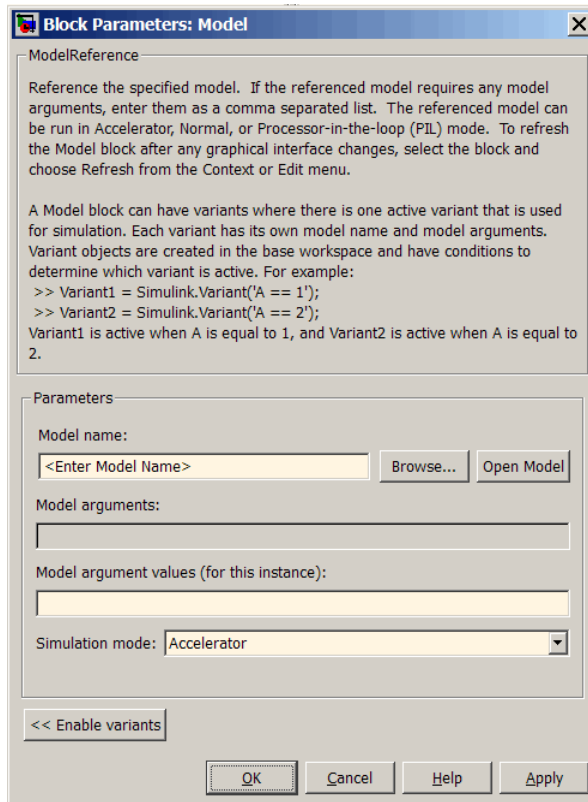


Enabling Model Variants

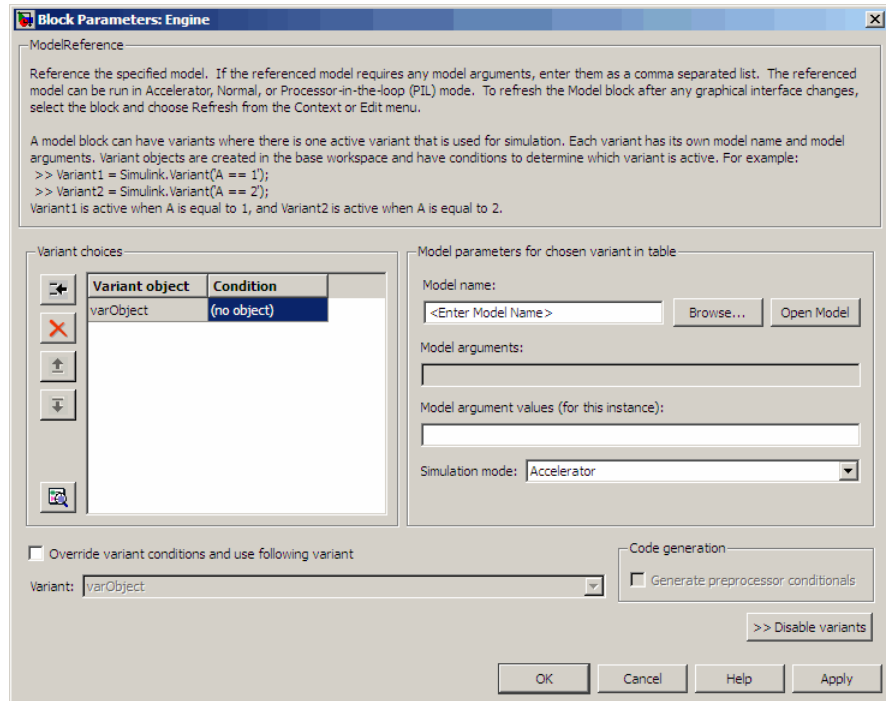
Defining variant control variables and objects does not enable model variants for any Model block. It only makes the resources available to every Model block that needs to use them. The Model block parameters dialog provides an optional section for enabling and specifying model reference variants. By default, model reference variants are disabled and the section for defining them is hidden. To enable variants and display the Model Reference Variants section:

- 1 Select the relevant Model block (Engine in the tutorial example).
- 2 Choose **Model Reference Parameters** from the **Edit** menu or the context menu.


The Model block parameters dialog box opens. For a new block, model reference variants are initially disabled, and the dialog box looks like this:



- 3 Click **Enable Variants** at the bottom left of the dialog box to enable variants and display the Model Reference Variants section. For a new Model block, the dialog box now looks like this:



Once you enable variants, they remain enabled until you explicitly disable them. You can close and reopen the Model block as needed without affecting model variant enablement or specification. Whenever the Model Reference

Variants section is open, you can use the Model Explorer button  on the left side of the dialog to open the Model Explorer on the base workspace. See “Changing Variant Model Definitions” on page 6-57 for information about the other buttons on the left side of the dialog.

Disabling Model Variants. Disabling model reference variants hides the Model Reference Variants section, and leaves in effect the model name and execution environment of the variant that was active when variants were disabled. Subsequent changes to variant control variables, variant conditions, and other models other than the current model have no effect on the behavior of the Model block. To disable model variants:

- Click **Disable variants** at the bottom right of the Model block dialog box.

Disabling variants for one Model block has no effect on variants specified by any other Model block. Disabling variants does not discard any variant-related specifications; it just disables and hides them. If you later re-enable variants, the variant model specifications in the Model block will be the same as they were before, and the Model block will select a variant according to the current base workspace variables and conditions.

Specifying Variant Models

With variants enabled, the Model block displays the parameters specific to variant models, as well as the parameters that apply to any referenced model, as shown in the previous figure. A variant model specification in a Model block parameters dialog box must include at least:

- The name of a variant object, in the **Variant object** column of the **Variant choices** table.
- The name of a variant model, in the **Model Name** field of the **Model parameters** section.
- A simulation mode, in the **Simulation mode** field of the **Model parameters** section.

The variant objects and models do not need to exist at the time that you specify them in the Model block dialog box. Existing variant objects do not initially appear in the **Variant object** column, because Simulink cannot know in advance which existing objects are relevant to the particular Model block. You cannot use the **Variant choices** table to define variant objects: the objects must be defined separately, as described in “Implementing Variant Objects” on page 6-49. To specify the four variant models in the example:

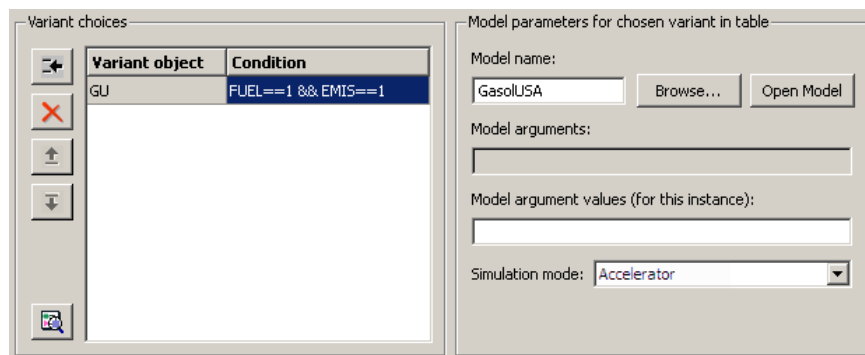
- 1** Select the Model block named Engine, open its block parameters dialog, and enable variants as previously described.
- 2** In the **Variant object** column of the **Variant choices** table, double-click varObject. Edit its value to be the name of the variant object GU.

For an existing variant object, Simulink retrieves the object’s variant condition as soon as you click away from the **Variant object** field. Simulink displays the variant condition in the **Condition** field, but you

cannot edit it there. To change a variant condition, you must redefine the variant object in the base workspace.

- 3** In the **Model name** field of the **Model parameters** section, enter the name of the associated variant model, `GasolUSA.mdl`, or browse to the model using the Browse button to the right of the **Model name** field. (You could also specify a protected model with a `.mdlp` extension. See “Protecting Referenced Models” on page 6-63.)
- 4** In the **Simulation mode** field of the **Model parameters** section, select Normal mode if you are building the Automobile model. (All simulation mode work with variant models. See “Referenced Model Simulation Modes” on page 6-12)

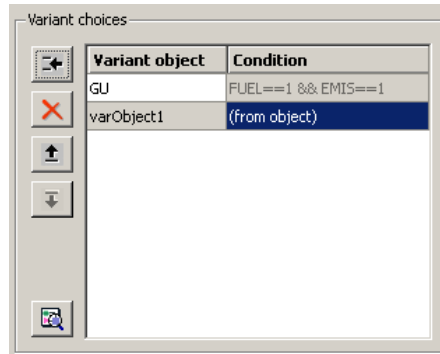
For example, in `ModRefVar.mdl` the changed parts of the dialog box now look like this:



- 5** Click the **Add a new variant** button:



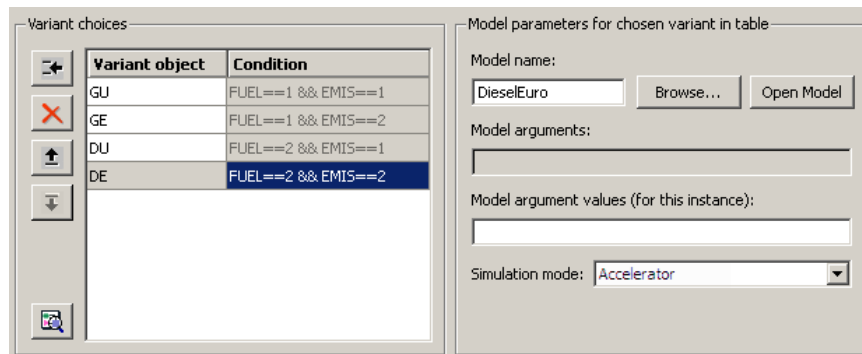
A new **Variant choices** row appears below the first, for example:



- 6 Repeat the preceding instructions until you have specified all variant objects and their variant models. The complete specifications for `ModRefVar.mdl` (which you can paste from this page) are :

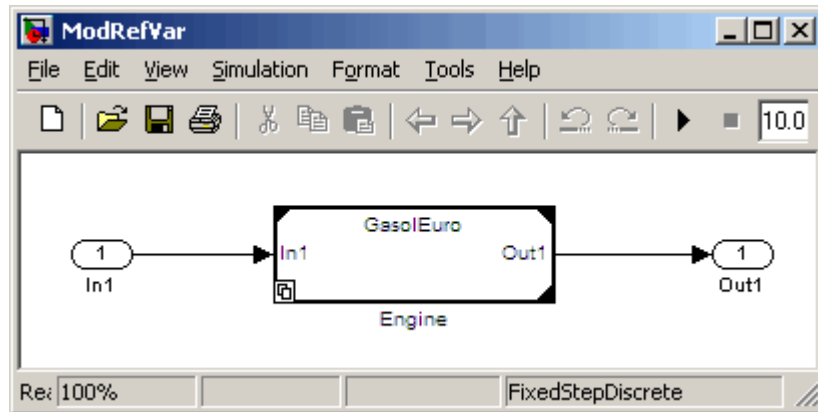
Variant Object	Variant Model
GU	GasolUSA.mdl
GE	GasolEuro.mdl
DU	DieselUSA.mdl
DE	DieselEuro.mdl

In `ModRefVar.mdl` the changed parts of the dialog box now look like this:



- 7 Click **Apply** or **OK**.

- 8 Connect the Model block to any input and output signals. (You could have done this at any time after the first variant model specification existed.) For example, the completed model `ModRefVar.mdl` looks like this:



Executing the Example

If you have used this section as a tutorial, you can now do anything with `ModRefVar.mdl` that you could do with `Tutorial.mdl`. See the following sections for more about what you can do with model reference variants. See the Model block documentation for information about all **Model Block Parameters** dialog box capabilities, including the capabilities of the **Variant choices** pane.





Saving Variant Referenced Models

Saving the model does *not* save the variant control variables or variant objects. As with all base workspace definitions, you must save the variables and objects in an M-file or MAT-file if you want to be able to reload them later. See the `save` command for more information.

Changing Variant Model Definitions

You can change variant control variables and variant objects between simulations as needed, using either the MATLAB Command Window or the Model Explorer. The changes take effect immediately. You can change variant model specifications between simulations by reopening the Model

block parameters dialog, selecting any line in the **Variant choices** table, and modifying the specification. Use the four buttons at the left of the table to manipulate the table rows:

Button	Function
	Add a new, empty row at the bottom of the list.
	Delete the currently selected row. Models and objects are unaffected.
	Move the currently selected row up one slot in the table.
	Move the currently selected row down one slot in the table.

The order of rows in the table has no functional significance, but systematic ordering can help you to track the specifications. Changes to variant model specifications take effect when you click **Apply** or **OK**. You can change multiple specifications before saving the changes. Be sure to save the model to make the changes permanent, and to save any changed variables or objects using the save command.

Parameterizing Variant Models

You can parameterize any or all variant models, as described in “Parameterizing Model References” on page 6-27. You can parameterize some variants but not others, and you can parameterize different variants differently from one another. To parameterize a variant model, configure the model appropriately, then specify the necessary values in the **Model parameters** section’s **Model argument values** field. The values needed by each variant model are the same as if it were an ordinary referenced model and no variants existed.

Reusing Variant Objects and Models

For simplicity the preceding example and instructions show only one Model block (Engine) and use each variant object and model only once. In a real model, more than one Model block might need to use model reference variants, and the various Model blocks might need to reuse some of the same variant objects and models.

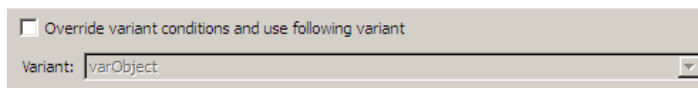
For example, a real model of an automobile might include many other capabilities, like the fuel and exhaust systems, that need to change depending on the applicable fuel and emission standards. To model different contexts efficiently with variant referenced models, you can:

- Use the same variant object in more than one Model block, associating it with the same or a different model in each case, so that the blocks change their selected variants in synchrony as variant control values change. The separate uses of the variant object do not affect one another.
- Use the same variant model with more than one variant object in the same model block, assigning a different parameterization and/or execution mode in each case. The separate uses of the model do not affect one another.

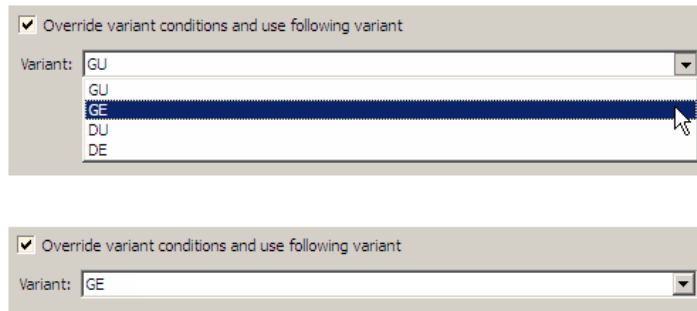
You cannot use the same variant object more than once in the same Model block, because a conflict would occur if that variant condition were true. The same type of conflict occurs if two different variant objects used by a Model block have variant conditions that are true.

Overriding Variant Conditions

The bottom left of the Model block parameters dialog with variants enabled provides the **Override variant conditions and use following variant** checkbox. You can use this checkbox to override the boolean values of all variant conditions, and specify that a particular variant is the active variant. Initially the checkbox looks like this:



If you select the checkbox, the **Variant** field becomes active. You can then use its pulldown menu to select any available variant object. For example:



Click **Apply** or **OK** after making the change. The variant object that you specified becomes the active variant, overriding all other specifications that determine which variant is active. The variant remains active until you either use the **Variant** pulldown to select another variant, or clear **Override variant conditions and use following variant**. The default variant model selection criteria then take effect. You can change the effect of the variant specified by the override by editing it in the same way you ordinarily would. See “Changing Variant Model Definitions” on page 6-57 for details.

Variant Models and Enumerated Types

You can use enumerated types to give meaningful names to the integers that you use as values of variant control variables. See Chapter 14, “Using Enumerated Data” for information about defining and using enumerated types. For example, suppose you define the following enumerated class, whose elements represent vehicle properties:

```
classdef(Enumeration) VP < Simulink.IntEnumType
    enumeration
        gasoline(1)
        diesel(2)
        American(1)
        European(2)
    end
end
```

This class could just as well have been two classes, one for fuels and one for emissions. If many more variant control variable values existed, the result might be more readable, but no computational difference would result.

With the class VP defined, you could use meaningful names to specify variant control variables and variant conditions. For example, you could use the techniques shown in “Implementing Variant Control Variables” on page 6-48 and “Implementing Variant Objects” on page 6-49 , substituting names for integers:

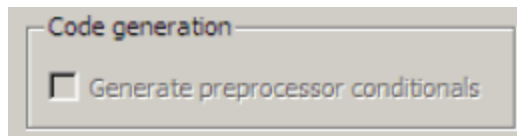
```
EMIS = American
FUEL = diesel

GU=Simulink.Variant('FUEL==VP.gasoline && EMIS==VP.American')
GE=Simulink.Variant('FUEL==VP.gasoline && EMIS==VP.European')
DU=Simulink.Variant('FUEL==VP.diesel && EMIS==VP.American')
DE=Simulink.Variant('FUEL==VP.diesel && EMIS==VP.European')
```

The ability to specify meaningful names rather than integers as the values of variant control variables facilitates creating complex variant model specifications. It also clarifies code generated from the model, because the code shows the names rather than arbitrary integers.

Generating Code for Variant Models

The bottom right of the Model block parameters dialog with variants enabled provides a Code Generation section that contains the **Generate preprocessor conditionals** checkbox:



This checkbox controls whether generated code contains preprocessor conditionals. The checkbox is relevant only to code generation, and has no effect on the behavior of a model in Simulink. The checkbox is available only for ERT targets when **Inline parameters** is 'on'. See “Generating Code Variants for Variant Models” for complete information.

Model Reference Variant Limitations

The following limitations apply to model reference variants.

- A model can perform only default data logging when used as a variant model.
- Model reference variants cannot be used with External mode.

In addition, a Model block and its variant models must satisfy all the requirements listed in “Model Reference Variant Requirements” on page 6-46.

Protecting Referenced Models

In this section...

- “About Protected Models” on page 6-63
- “The Model Protection Facility” on page 6-64
- “Model Protection Requirements” on page 6-65
- “Protecting a Referenced Model” on page 6-65
- “Packaging a Protected Model” on page 6-69
- “Testing a Protected Model” on page 6-70
- “Using a Protected Model” on page 6-70
- “Model Protection Limitations” on page 6-72

About Protected Models

A *protected model* is a referenced model from which all block and line information have been eliminated. Protecting a model does not use encryption technology. A protected model can be distributed without revealing the intellectual property that it embodies. When simulated, a protected model gives the same results that its source model does when referenced in Accelerator mode.

You can use a protected model much as you could any other referenced model. Simulink tools work with protected models to the extent possible given that the model’s contents are obscured. For example, the Model Explorer and the Model Dependency Viewer show the hierarchy under an ordinary referenced model, but not under a protected model. Signals in a protected model cannot be logged, because the log could reveal information about the protected model’s contents.

When a referenced model requires object definitions or tunable parameters that are defined in the MATLAB base workspace, the protected version of the model may need some or all of those same definitions when it executes as part of a third-party model. Simulink provides techniques for identifying and obtaining the needed data, and then packaging it with the protected model for delivery.

Protecting a model requires a Real-Time Workshop license, which makes code generation capabilities available for use internally when creating the protected version of the model. A third party that receives the model does not need a Real-Time Workshop license to use the model, and cannot use Real-Time Workshop to generate code for the model or any model that references it.

The Model Protection Facility

The *Model Protection facility* is a set of Simulink commands, tools, and techniques that you can use to protect a referenced model, identify definitions that it needs, and package the model and any definitions for delivery to a third party. The receiver can then unpackage and use the protected model and any definitions. To see a demo of the Model Protection facility, access MATLAB Online Help and choose:

```
Demos > Simulink > Modeling Features > Model Reference > Model Reference Protected Models
```

The Model Protection facility is not an integrated unit like a viewer or editor, and has only a command-line interface. To begin using the facility, you execute a `protect` command in the MATLAB base workspace. See the `Simulink.ReferencedModel.protect` reference page for complete information about the `protect` command. The syntax of the command is:

```
[harnessHandle, neededVars] = Simulink.ModelReference.protect  
(model, 'Param', Val, ...)
```

You can use the `protect` command and its arguments to:

- Specify the model to be protected. See “Protecting a Referenced Model” on page 6-65.
- Specify the directory in which the `protect` command will place the protected model. See “Specifying an Output Directory” on page 6-66.
- Create a *harness model*, which contains a model block that is preconfigured to work with the protected model. See “Creating a Harness Model” on page 6-66.
- Obtain a list that includes the names of all base workspace entities that the model needs in order to run. See “Obtaining Object and Variable Names” on page 6-67.

For simplicity, “Protecting a Referenced Model” on page 6-65 shows you how to use each of these capabilities separately when protecting a model. In practice, you can use any or all of the capabilities at the same time, and specify optional argument pairs in any order.

After you have created or obtained all the components that you intend to provide to the receiver, you can package them for delivery as described in “Packaging a Protected Model” on page 6-69. The receiver of the model will need instructions for unpacking the component(s) and using them in a model. The necessary instructions appear in “Using a Protected Model” on page 6-70.

Model Protection Requirements

Protecting a model requires meeting all requirements listed in “Simulink Model Referencing Requirements” on page 6-20, as well as the following:

- You must have a Real-Time Workshop license, which you can obtain via <http://www.mathworks.com>.
- Your platform and your version of Simulink must match those on which the protected model will be used.
- The model to be protected must be available on the MATLAB path and have no unsaved changes.
- The model to be protected cannot reference any subordinate protected model directly or indirectly.

Model protection is also subject to certain limitations, as described in “Model Protection Limitations” on page 6-72.

Protecting a Referenced Model

If you need nothing but the protected model itself, with no harness model or list of names, all you need to do is specify the source model to the `protect` command.

- Be sure that the model to be protected and the model hierarchy that contains it:
 - 1 Meet the “Model Protection Requirements” on page 6-65

2 Respect the “Model Protection Limitations” on page 6-72 .

- In the MATLAB Command Window, execute:

```
Simulink.ModelReference.protect(model)
```

The *model* is the name or handle of the model to be protected, or the full path to the model to be protected. If the model block uses variants, only the active variant is protected. See “Using Model Reference Variants” on page 6-38.

The `protect` command creates a protected version of the model and stores it by default in the current working directory. You can change this default as described in “Specifying an Output Directory” on page 6-66. The protected model has the same name as the source model, with the suffix `.mdl.p`. In the MATLAB Directory Browser, a small image of a lock appears in the upper left corner of a protected model’s icon.

Never change the name of a protected model. If you rename a protected model, or change its suffix, the model will be unusable until its original name and suffix are restored. You also cannot change a protected model internally in any way: any internal change will destroy the usability of the file.

Specifying an Output Directory

To specify a directory other than the current working directory to contain the protected model, use the optional `Path` argument:

```
Simulink.ModelReference.protect(model, 'Path', pathname)
```

where *pathname* is a fully qualified pathname of an accessible writeable directory. The command stores the protected model in the indicated directory.

Creating a Harness Model

When the `protect` command protects a model, it can also create a *harness model*. This model contains a Model block that is preconfigured to work with the protected model. This Model block:

- Names the protected model as its referenced model.
- Has the same number of input and output ports as the protected model

- Defines any model reference arguments that the protected model uses, but provides no values

To create a harness model when protecting a model, specify the optional `Harness` argument as `true`. The `protect` command returns the handle of the harness model in the first output argument:

```
[harnessHandle] = Simulink.ModelReference.protect(model, 'Harness', true)
```

The harness model opens as a new untitled model that contains only the described Model block. This block, and any other Model block that references a protected model, shows a small image of a lock in its upper left corner. You can:

- Save the harness model under any name and use it as needed
- Use the handle in `harnessHandle` to manipulate the model programmatically
- Copy the Model block into another model, where it functions as the interface to the protected model

See “Packaging a Protected Model” on page 6-69 for more about how to use a harness model and the Model block that it contains.

Obtaining Object and Variable Names

When a referenced model requires any object or tunable parameter that is defined in the MATLAB base workspace, executing the protected version of the model will require that same entity if it is:

- A global tunable parameter
- A global Data Store Memory
- Any of the following objects used by a signal that connects to a root-level model Inport or Outport:
 - `Simulink.Signal`
 - `Simulink.Bus`
 - `Simulink.Alias`
 - `Simulink.NumericType` that is an alias

When you protect a model, you must obtain the definitions of all necessary base workspace entities and ship them along with the model. The necessary steps for obtaining the definitions are:

- Obtain candidate names
- Prune unneeded names
- Save workspace definitions

See “Packaging a Protected Model” on page 6-69 for information about how to ship the base workspace definitions.

Obtaining Candidate Names. When the `protect` command executes, it always generates a cell array that includes the names of all base workspace entities that the protected model might need. You can obtain this array by accepting the second return value when you execute the command:

```
[~, neededVars] = Simulink.ModelReference.protect(model)
```

The tilde (~) operator discards the first argument, or you could accept it if you create a harness model at the same time. The returned cell array *neededVars* is guaranteed to include the name of every needed base workspace entity.

Pruning Unnecessary Names. The cell array *neededVars* may also include the names of base workspace entities that the protected model does not need. For example, *neededVars* could contain names of workspace objects that define signals that do not connect to any root I/O port within the protected model. The protected version of the model does not need the corresponding definitions, because they do not specify anything about its interface to other model entities.

Leaving unnecessary names in *neededVars* risks disclosing intellectual property, adds unnecessary definitions to the receiver’s model, and increases the likelihood of a name conflict with that model. Because a protected model cannot be changed, resolving such a conflict would require the receiver’s model to change.

To remove unnecessary names, edit the cell array *neededVars* and delete any names that do not correspond to definitions that serve the protected model in one of the capacities listed under “Obtaining Object and Variable Names” on

page 6-67. If you are not sure which names are extra, you can remove any doubtful cases and later test to see whether any missing definitions result.

Saving Workspace Definitions. The cell array derived in the previous steps contains only the names of base workspace entities. The protected model needs the definitions themselves, in a separate file that can be shipped along with the model. To create this file, execute:

```
save(myFile.mat, neededVars{:})
```

where *myFile* is any file (or path) name, and *neededVars* is the (possibly pruned) second return value of the `protect` command. The `{:}` operator converts the cell array *neededVars* into a list of comma-separated names, which become arguments to `save`. When the command executes, it evaluates each name, thus obtaining the corresponding definition from the base workspace, and stores all the definitions in *myFile.mat*. The receiver of the protected model can then restore the base workspace definitions by loading *myFile.mat*.

Packaging a Protected Model

A protected model consists of the model itself, optionally a harness model, and any needed definitions saved in a MAT file, as described in “Saving Workspace Definitions” on page 6-69. The Model Protection facility does not require packaging these files in any specific way. For example, they could be:

- Provided as three separate files.
- Combined into a ZIP or other container file.
- Packaged using a manifest. See “Exporting Files in a Manifest” on page 19-42.
- Provided in some other standard or proprietary format specified by the receiver.

All that matters is that the receiver has the information necessary to access any packing format used and retrieve the original file(s). One way to ensure this result is to use the Simulink Manifest Tools, as described in “Model Dependencies” on page 19-29.

Testing a Protected Model

To test a protected model, you enact the same sequence that the receiver will enact, as described in “Using a Protected Model” on page 6-70. Since you are also the supplier, both the original and the protected model may exist on the MATLAB path. If a Model block’s **Model name** field names the model without providing a suffix, the unprotected (.mdl) version will take precedence over the protected (.mdlp) model, regardless of their relative positions on the MATLAB path. Explicitly specify the .mdlp suffix in the Model block’s **Model name** field if needed to override this default when testing a protected model.

If you want to ensure that the protected model gives the same results as the source model, you can create a parent model that contains two Model blocks. One Model block references the source model, in Accelerator mode, and the other references the protected model. You can then log the outputs of the blocks and compare the logged data to see that the two versions of the block behave identically. See “Logging Signals” on page 15-3 for more information.

Using a Protected Model

Using a protected model does not require a Real-Time Workshop license. Using a protected model requires that the model:

- Be available somewhere on the MATLAB path.
- Be referenced by a Model block in a model that executes in Normal mode.
- Receive from the Model block the values needed by any defined model arguments.
- Connect via the Model block to input and output signals that match those of the protected model.

A typical workflow for meeting these requirements is:

- 1** If necessary, unpack the files according to any accompanying directions.
- 2** If a MAT-file containing workspace definitions is provided, load the MAT-file.
- 3** If a harness model is provided, copy the Model block into a Normal-mode model.

- 4 Connect signals to the Model block that match its input and output port requirements.
- 5 Provide any needed model argument values. See “Assigning Model Argument Values” on page 6-31.

You should now be able to simulate the model that references the protected model, which will produce the same outputs that it did when used in Accelerator mode in the source model. Many variations on these instructions are possible, such as:

- Use your own Model block rather than the Model block in the harness model.
- Start with the harness model, add more constructs to it, and use it in your model.
- Use the protected model as a variant, as described in “Using Model Reference Variants” on page 6-38.
- Apply a mask to the Model block that references the protected model. See Chapter 9, “Working with Block Masks”.
- Configure a callback function such as **LoadFcn** to load the MAT file automatically. See “Using Callback Functions” on page 4-54.

To facilitate locating protected models:

- The MATLAB Directory Browser shows a small image of a lock in the upper left corner of a protected model’s icon
- A Model block that references a protected model shows a small image of a lock in the block’s upper left corner.
- When the Model block **Browse** button shows a protected model, the `.mdl` suffix and lock icon appear.

When you change a Model block to reference a protected model, the block’s **Simulation mode** becomes Accelerator and cannot be changed.

Model Protection Limitations

Protected models are subject to all limitations listed in “Limitations on All Model Referencing” on page 6-78 and “Limitations on Accelerator Mode Referenced Models” on page 6-81. The following limitations also apply to protected models.

- A model that is to be protected cannot use a non-inlined S-function directly or indirectly.
- Simulating a protected model requires that all models superior to it execute in Normal mode.

Protected models must also meet certain requirements, as described in “Model Protection Requirements” on page 6-65.

Inheriting Sample Times

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.

Without the ability to inherit sample times, a Model block could not be placed in a triggered, function call, or iterator subsystem. 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, but could be reused with different sample times, for example, discrete at 0.1, discrete at 0.2, triggered, and so on.

A referenced model inherits its sample time if and only 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 6-75.

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, you must ensure that blocks connected to a referenced model with intrinsic sample times operate at the same rates as the 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
- 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 set to Time-based)
- Stateflow (only when the chart uses the reserved word `t` to reference time)
- Step
- To File
- To Workspace (only when logging to StructureWithTime format)
- Transport Delay
- Variable Time Delay
- Variable Transport Delay

Some blocks other than Simulink blocks may 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, including any S-Function blocks, whose outputs are functions of the inherited simulation time. If so, Simulink specifies a default sample time and displays an error if you have set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane 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, and therefore 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 *Writing S-Functions* 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.

Refreshing Model Blocks

Refreshing a Model block updates its internal representation to reflect changes in the interface of the model that it references. For example, you must refresh a Model block if its referenced model has gained or lost a port. When more than one Model block references a model whose interface has changed, you must refresh all the Model blocks. Changes that do not affect a referenced model's interface to its parent do not require refreshing the block.

To refresh all Model blocks in a model, select **Refresh Model Blocks** from the model's **Edit** menu. To update a specific Model block, select **Refresh** from the block's context menu. You can also refresh a model by starting a simulation or generating code.

Simulink provides diagnostics that you can use 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”**

Simulink Model Referencing Limitations

In this section...
“Introduction” on page 6-78
“Limitations on All Model Referencing” on page 6-78
“Limitations on Normal Mode Referenced Models” on page 6-81
“Limitations on Accelerator Mode Referenced Models” on page 6-81
“Limitations on PIL Mode Referenced Models” on page 6-84

Introduction

The following Simulink limitations apply to model referencing. In addition, a model reference hierarchy must satisfy all the requirements listed in “Simulink Model Referencing Requirements” on page 6-20.

Limitations on All Model Referencing

Index Base Limitations

In the following 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, like the Assignment block, or produce indexes, like the For Iterator block.

- If a root-level input port of the referenced model is connected 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.
- If a root-level output port of the referenced model is connected 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.

In these cases, the lack of propagation can cause Simulink to fail to detect incompatible index connections.

General Reusability Limitations

If a referenced model has any of the following properties, 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. If the parameter is not set correctly, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model contains any To File blocks, to avoid multiple instances writing to the same file
- The model references another model which has been 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 events
 - Machine-parented data

Block Mask Limitations

- Mask callbacks cannot add Model blocks or change Model block name or simulation mode. Violating this requirement generates an error.
- If a mask specifies the name of the model referenced by a Model block, the name of the referenced model must be provided directly by the mask, rather than being obtained by evaluating a workspace variable.
- The mask workspace of a Model block is not visible to the model that it references. Any variable used by the referenced model must resolve to a workspace defined in the referenced model, or to the MATLAB base workspace.

See for information about creating and using block masks.

Simulink Tool Limitations

- Working with the Simulink Debugger in a parent model, you can set breakpoints at Model block boundaries, allowing you to look at the block's

input and output values, but you cannot set a breakpoint inside the submodel that the Model block references. See Chapter 26, “Simulink Debugger” for more information.

Stateflow Limitations

- A model that contains a Stateflow chart cannot be referenced multiple times in the same model reference hierarchy if:
 - The Stateflow chart contains exported graphical functions.
 - The Stateflow model contains machine-parented data or events.

Other Limitations

- Referenced models cannot use asynchronous rates internally. However, a function-call model referenced in a top model can be triggered by an asynchronous source within the top model. See “Defining Function-Call Models” on page 6-34 for more information.
- A referenced model can input or output only those user-defined data types that are fixed-point or defined by `Simulink.DataType` or `Simulink.Bus` objects.
- Model blocks referencing models that contain assignment blocks that are not in an iterator subsystem cannot be placed in an iterator subsystem.
- If you want to initialize the states of a model that references other models with states, you must specify the initial states in structure format.
- The Model Browser does not display Model blocks in its tree view. Use the Model Explorer to browse a referenced model hierarchy.
- A referenced model cannot directly access the signals in a multi-rate bus. Connecting Multi-Rate Buses to Referenced Models describes a technique for overcoming this limitation.
- A continuous sample time cannot be propagated to a Model block that is sample-time independent.
- Goto/From blocks cannot cross model reference boundaries.
- You cannot print a referenced model from a top model.

Limitations on Normal Mode Referenced Models

Simulink Tool Limitations

- Enabling the Simulink Profiler on a parent model does not enable profiling for referenced models. Profiling must be enabled separately for each submodel. See “Capturing Performance Data” on page 27-29.

Other Limitations

- When the same submodel appears more than once in a hierarchy, at most one of these instances can specify Normal mode. All the rest must specify Accelerator mode.

Limitations on Accelerator Mode Referenced Models

Customization Limitations

- Accelerator mode simulation ignores custom code settings in the **Configuration Parameter** dialog box and custom code blocks when generating the simulation target for a referenced model.
- Some restrictions exist on grouped custom storage classes in referenced models. See “Custom Storage Class Limitations” for details.
- Data type replacement is not supported for simulation target code generation for referenced models. Also, the custom code is ignored in Simulation Target > Custom code

Data Logging Limitations

- To Workspace blocks, Scope blocks, and all types of runtime display, such as Port Values Display, have no effect when specified in referenced models executing in Accelerator mode. The result during simulation is the same as if the constructs did not exist.
- Referenced models executing in Accelerator mode cannot log data to MAT-files. If data logging is enabled for a referenced model, Simulink disables the option before code generation and re-enables it afterwards.

Accelerator Mode Reusability Limitations

If a referenced model has any of the following properties, and the model executes in Accelerator mode, 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, in either Normal mode or Accelerator mode. If the parameter is not set correctly, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model contains a subsystem that is marked as function
- The model contains an S-function that is:
 - Inlined but has not set the option `SS_OPTION_WORKS_WITH_CODE_REUSE`
 - Not inlined
- The model contains a function-call subsystem that:
 - Has been forced by Simulink to be a function
 - Is called by a wide signal

S-Function Limitations

- If a 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 will not inline the S-function unless this flag is set.
- The Real-Time Workshop S-function target does not support model referencing.
- A referenced model cannot use noninlined S-functions in the following cases:
 - The model uses a variable-step solver.
 - The S-function was generated by Real-Time Workshop.
 - 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, make copies of the referenced model, assign different names to the copies, and reference a different copy at each location that needs the model.

Simulink Tool Limitations

- Simulink tools that require access to a model's internal data or configuration (including Model Coverage, the Report Generator, the Simulink Debugger, and the Simulink Profiler) 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.

Stateflow Limitations

A Stateflow chart in a referenced model that executes in Accelerator mode cannot call MATLAB functions.

Subsystem Limitations

- If a subsystem contains Model blocks, you cannot build a subsystem module by right-clicking the subsystem or by using **Tools > Real-Time Workshop > Build subsystem**.
- If you generate code for an atomic subsystem as a reusable function, inputs or outputs that connect the subsystem to a referenced model can affect code reuse, as described in “Reusable Code and Referenced Models”.

Target Limitations

- The Real-Time Workshop `grt_malloc` targets do not support model referencing.
- The Real-Time Workshop S-function target does not support model referencing.

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, as described in “Specifying the Simulation Mode” on page 6-13.
- When the `sim` command executes a referenced model in Accelerator mode, the source workspace is always the MATLAB base workspace (`SrcWorkspace` is `base`) even if a `simset` options structure specifies some other source workspace.
- Errors can occur if a Model block is part of a cycle; the Model block is a direct feedthrough block; and an algebraic loop results. See “Model Blocks and Direct Feedthrough” for details.
- Accelerator mode does not support the **External mode** option. If the option is enabled, it is ignored in Accelerator mode.

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 new model, you cannot use that model as a PIL mode referenced model until you have saved the model to disk. You can work around this limitation by setting the model to Normal mode, as described in “Specifying the Simulation Mode” on page 6-13.

For more information, see “Creating Model Components” in the Real-Time Workshop documentation, and “Verifying Compiled Object Code with Processor-in-the-Loop Simulation” in the Real-Time Workshop Embedded Coder documentation.

Working with Blocks

- “About Blocks” on page 7-2
- “Adding Blocks” on page 7-4
- “Editing Blocks” on page 7-9
- “Working with Block Parameters” on page 7-15
- “Changing a Block’s Appearance” on page 7-34
- “Displaying Block Outputs” on page 7-41
- “Controlling and Displaying the Sorted Order” on page 7-46
- “Accessing Block Data During Simulation” on page 7-56

About Blocks

In this section...

“What Are Blocks?” on page 7-2

“Block Data Tips” on page 7-2

“Virtual Blocks” on page 7-2

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.

Block Data Tips

Information about a block is displayed in a pop-up window when you allow the pointer to hover over the block in the diagram view. To disable this feature or control what information a data tip includes, select **Block data tips options** from the Simulink **View** menu.

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.

Block Name	Condition Under Which Block 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” in the <i>Simulink Reference</i> .
Terminator	Always virtual.
Trigger	Virtual when the output port is <i>not</i> present.

Adding Blocks

In this section...
“Ways to Add Blocks” on page 7-4
“Adding Blocks by Browsing or Searching with the Library Browser” on page 7-5
“Copying Blocks from a Model” on page 7-5
“Adding Frequently Used Blocks” on page 7-6
“Adding Blocks Programmatically” on page 7-8

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.

Adding 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 “Populating a Model” on page 4-4 for more information.

Copying Blocks from a Model

To copy a block from a model in the Model Editor:

- 1 Select the block you want to copy.
- 2 Choose **Edit > Copy**.
- 3 If you have multiple model windows open, make the target model window the active window.
- 4 Choose **Edit > Paste**.

When you copy a block, the parameters use default values.

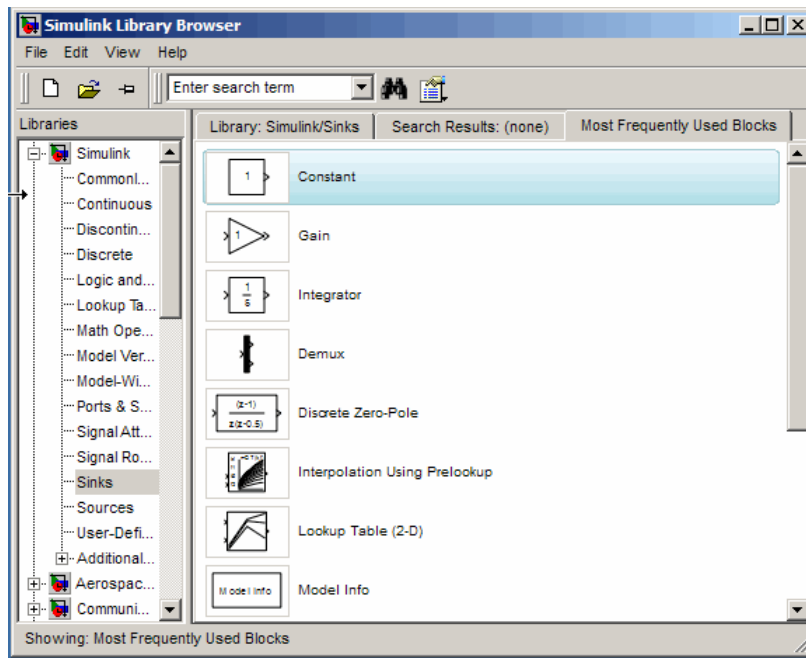
See “Copying and Moving Blocks from One Window to Another” on page 7-9 for details.

Adding Frequently Used Blocks

When using the same block repeatedly, you can save time by using the **Most Frequently Used Blocks** feature in the Library Browser or Model Editor.

Selecting Your Most Frequently Used Blocks From the Library Browser

Open the **Most Frequently Used Blocks** pane with the third tab in the Library Browser.



To add a block to a model, select the block from the list and do either one of the following:

- Drag the block into your model.
- Select **Add to <model_name>** at the top of the context menu for the block.

Selecting Your Most Frequently Used Blocks from the Model Editor

In the Model 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 in the Most Frequently Used Blocks Lists

The Most Frequently Used Blocks pane 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 (In other words, 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.

Adding 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.

Editing Blocks

In this section...

“Copying and Moving Blocks from One Window to Another” on page 7-9

“Moving Blocks in a Model” on page 7-10

“Copying Blocks in a Model” on page 7-13

“Deleting Blocks” on page 7-14

Copying and Moving Blocks from One Window to Another

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 “Browsing Block Libraries” on page 4-5 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 **Copy** and **Paste** commands from the **Edit** menu:

- 1 Select the block you want to copy.
- 2 Choose **Copy** from the **Edit** menu.

3 Make the target model window the active window.

4 Choose **Paste** from the **Edit** menu.

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 “Manipulating Block Names” on page 7-39.

When you copy a block, the new block inherits all the original block’s parameter values.

For more ways to add blocks, see “Adding Blocks” on page 7-4.

add blocks

Moving Blocks in a Model

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. If you need information about how to select more than one block, see “Selecting Multiple Objects” on page 4-7.

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.

Moving blocks from one window to another is similar to copying blocks, except that you hold down the **Shift** key while you select the blocks.

You can use the **Undo** command from the **Edit** menu to remove an added block.

Aligning Blocks

You can use tools to manually align blocks. These tools include a grid snap feature and smart guides that indicate when a block center or port aligns with the center or port of another block. You can also use commands that align a group of blocks automatically (see “Aligning, Distributing, and Resizing Groups of Blocks Automatically” on page 4-24 for details).

Grid Snap. Simulink uses an invisible five-pixel grid to simplify the alignment of blocks. When you move a block to a new location, the block snaps to the nearest line on the grid. You can display the grid and change its spacing.

To display the grid, enter the following command at the MATLAB command prompt.

```
set_param('<model name>', 'showgrid', 'on')
```

The default width of the grid is 20 pixels. To change the grid spacing, enter

```
set_param('<model name>', 'gridspacing', <number of pixels>)
```

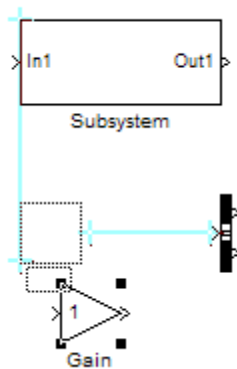
For example, to change the grid spacing to 25 pixels, enter

```
set_param('<model name>', 'gridspacing', 25)
```

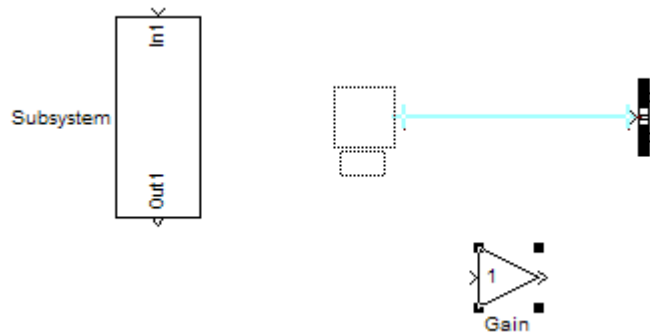
Note The new spacing must be a multiple of five pixels to ensure that the displayed grid aligns with the invisible snap grid.

For either of the above commands, you can also select the model, then enter `gcs` instead of `<model name>`.

Smart Guides. When you move a block, smart guides appear by default to indicate when the block ports, center, or edges are aligned with the ports, centers, and edges of other blocks in the same diagram. For example, the following figure shows a snapshot of a Gain block that you drag from one position in a diagram to another. The dotted outline indicates the position to which the Gain block has been dragged. The blue smart guides indicate that if you drop the Gain block at this position, its left edge will be aligned with the left edge of the Subsystem block and its output port will be aligned with the input port of the Mux 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's center aligns with the Subsystem's center and the Gain block's output port aligns with the Mux block's input port. However, because ports take precedence over centers, Simulink draws a guide only for the ports.



You can turn smart guides off or on (the default) by selecting **Format > Smart Guides**.

Positioning Blocks Programmatically

You can position (and resize) a block programmatically, using its `Position` parameter. For example, the following command

```
set_param(gcb, 'Position', [5 5 20 20]);
```

moves the currently selected block to a location 5 points down and 5 points to the right of the top left corner of the block diagram and sets the block's height and width to 15 points, respectively.

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.

Copying Blocks in a Model

You can copy blocks in a model as follows. While holding down the **Ctrl** key, select the block with the left mouse button, then drag it to a new location. You can also do this by dragging the block using the right mouse button. Duplicated blocks have the same parameter values as the original blocks. Sequence numbers are added to the new block names.

Note The model editor 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, Gain24...Gain23.

Deleting Blocks

To delete one or more blocks, select the blocks to be deleted and press the **Delete** or **Backspace** key. You can also choose **Clear** or **Cut** from the **Edit** menu. 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.

You can use the **Undo** command from the **Edit** menu to replace a deleted block.

Working with Block Parameters

In this section...

“About Block Parameters” on page 7-15

“Mathematical Versus Configuration Parameters” on page 7-15

“Setting Block Parameters” on page 7-16

“Specifying Numeric Parameter Values” on page 7-17

“Checking Parameter Values” on page 7-19

“Changing the Values of Block Parameters During Simulation” on page 7-23

“Inlining Parameters” on page 7-25

“Block Properties Dialog Box” on page 7-27

“State Attributes Pane of Block Parameters Dialog Box” on page 7-33

About Block Parameters

All Simulink blocks have attributes that you can specify. Some user-specifiable attributes are common to all Simulink blocks, for example, a block’s name and foreground color. Other attributes are specific to a block, for example, the gain of a Gain block. Simulink associates a variable, called a block parameter, with each user-specifiable attribute of a block. You specify the attribute by setting its associated parameter to a corresponding value. 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'. The Simulink parameter reference lists the names, usages, and valid settings for Simulink block parameters (see “Common Block Parameters” and “Block-Specific Parameters”).

Mathematical Versus Configuration Parameters

Block parameters fall into two broad categories. A *mathematical parameter* is a parameter used to compute the value of a block’s output, for example, the Gain parameter of a Gain block. All other parameters are *configuration parameters*, for example, a Gain block’s Name parameter. In general, you can change the values of mathematical but not configuration parameters during simulation (see “Changing the Values of Block Parameters During Simulation” on page 7-23).

Setting 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 Model Editor's **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 "Changing a Block's Appearance" on page 7-34 for more information).

- **Block Properties** dialog box

Specifies various attributes that are common to all blocks (see "Block Properties Dialog Box" on page 7-27 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 "Displaying a Block's Parameter Dialog Box" on page 7-16 for information on displaying a block's parameter dialog box. For information on the parameter dialog of a specific block, see "Blocks — Alphabetical List" in the online Simulink reference.

- 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 "The Model Explorer" on page 19-2 for more information.

Displaying a Block's Parameter Dialog Box

To display a block's parameter dialog box, double-click the block in the model or library window. You can also display a block's parameter dialog box by selecting the block in the model's block diagram and choosing **BLOCK Parameters** from the model window's **Edit** menu or from the block's context (right-click) menu, where **BLOCK** is the name of the block you selected, e.g., **Constant Parameters**.

Note Double-clicking a block to display its parameter dialog box works for all blocks with parameter dialog boxes except for Subsystem blocks. You must use the Model Editor's **Edit** menu or the block's context menu to display a Subsystem block's parameter dialog box.

Specifying Numeric 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.

- “Using Workspace Variables in Parameter Expressions” on page 7-17
- “Resolving Variable References in Block Parameter Expressions” on page 7-18
- “Using Parameter Objects to Specify Parameter Values” on page 7-18
- “Determining Parameter Data Types” on page 7-19

Using 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 “Resolving Symbols” on page 4-75 and “Specifying Numeric Values with Symbols” on page 4-77.

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 “Changing the Values of Block Parameters During Simulation” on page 7-23.

Note If 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 “Parameter Considerations” in the Real-Time Workshop documentation.

Resolving 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 “Resolving Symbols” on page 4-75 and “Specifying Numeric Values with Symbols” on page 4-77 for more information.

Using 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`. For more information, see “Resolving Symbols” on page 4-75 and “Specifying Numeric Values with Symbols” on page 4-77.

Using parameter objects to specify parameters can facilitate tuning parameters in some applications (see “Using a Parameter Object to Specify a Parameter As Noninlined” on page 7-26 and “Parameterizing Model References” on page 6-27 for more information).

Note Do not use expressions of the form `p.Value` where `p` is a parameter object in parameter expressions. Such expressions cause evaluation errors when Simulink compiles the model.

Determining 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”).

Obtaining Parameter Information. You can use `get_param` to find the system and block parameter values for your model. See “Model and Block Parameters” for a list of 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')
```

Checking Parameter Values

Several 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. See the following topics for more information:

- “Blocks That Perform Parameter Range Checking” on page 7-20
- “Specifying Ranges for Parameters” on page 7-20

- “Performing Parameter Range Checking” on page 7-21

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
Lookup Table	Table data
Lookup Table (2-D)	Table data
Lookup Table (n-D)	Table data
Relay	Output when on Output when off
Repeating Sequence Interpolated	Vector of output values
Repeating Sequence Stair	Vector of output values
Saturation	Upper limit Lower limit

Specifying 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 scalar, real number with double data

type. The default value, [], is equivalent to $-\text{Inf}$ for the minimum value and Inf for the maximum value. 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 10-28).

Note You cannot specify the minimum or maximum value as NaN.

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 $(\text{sqrt}(a^2+b^2))$

Performing 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.
- When you start a simulation or select **Update Diagram** from the Simulink **Edit** menu, 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 “Specifying Ranges for Parameters” on page 7-20) and the block data type. That is, Simulink performs the following check:

$$\text{DataTypeMin} \leq \text{MinValue} \leq \text{VALUE} \leq \text{MaxValue} \leq \text{DataTypeMax}$$

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.

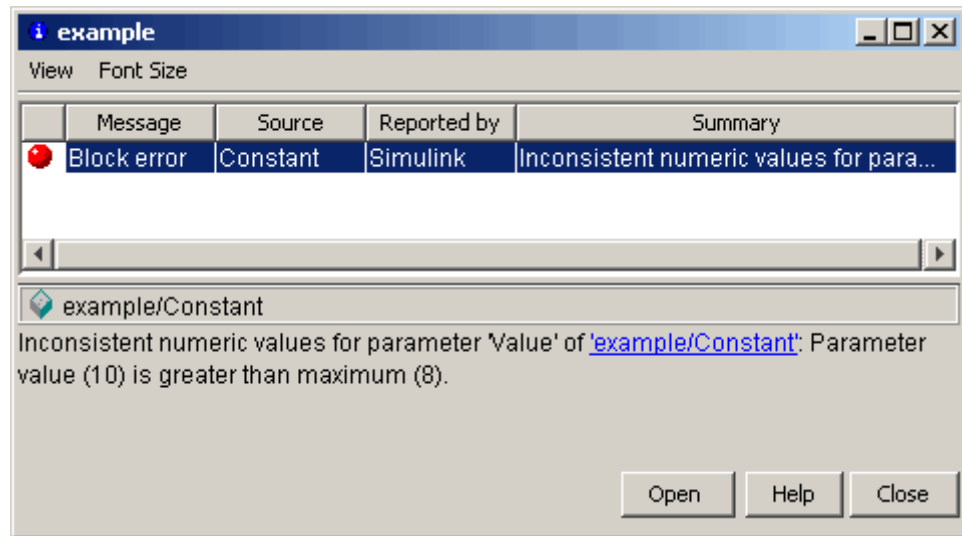
When Simulink detects a parameter value that violates the check, it displays an error message. For example, consider a model that contains a Constant block whose

- **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 “Updating a Block Diagram”). Simulink displays the following error message:



Changing the Values of Block Parameters During Simulation

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.

Note You cannot tune inline parameters. See “Inlining Parameters” on page 7-25 for more information.

Tuning a Block Parameter

You can use a block's dialog box or the Model Explorer to modify the tunable parameters of any block, except a source block (see “Changing Source Block

Parameters During Simulation” on page 7-24). 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 “Updating a Block Diagram” on page 1-27).

Changing Source Block Parameters During Simulation

Opening the dialog box of a source block with tunable parameters (see “Source Blocks with Tunable Parameters” on page 7-24) causes a running simulation to pause. While the simulation is paused, you can edit the parameter values displayed on the dialog box. However, you must close the dialog box to have the changes take effect and allow the simulation to continue. Similarly, starting a simulation causes any open dialog boxes associated with source blocks with tunable parameters to close.

Note If you enable the **Inline parameters** option, Simulink does not pause the simulation when you open a source block’s dialog box because all of the parameter fields are disabled and can be viewed but cannot be changed.

The Model Explorer disables the parameter fields that it displays in the list view and the dialog pane for a source block with tunable parameters while a simulation is running. As a result, you cannot use the Model Explorer to change the block’s parameters. However, while the simulation is running, the Model Explorer displays a **Modify** button in the dialog view for the block. Clicking the **Modify** button opens the block’s dialog box. Note that this causes the simulation to pause. You can then change the block’s parameters. You must close the dialog box to have the changes take effect and allow the simulation to continue. Your changes appear in the Model Explorer after you close the dialog box.

Source Blocks with Tunable Parameters. Source blocks with tunable parameters include the following blocks.

- Simulink source blocks, including
 - Band-Limited White Noise
 - Chirp Signal
 - Constant
 - Pulse Generator
 - Ramp
 - Random Number
 - Repeating Sequence
 - Signal Generator
 - Sine Wave
 - Step
 - Uniform Random Number
- User-developed masked subsystem blocks that have one or more tunable parameters and one or more output ports, but no input ports.
- S-Function and M-file (level 2) S-Function blocks that have one or more tunable parameters and one or more output ports but no input ports.

Inlining Parameters

The **Inline parameters** optimization (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 this option is on, the parameters appear in the generated code as inlined 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. Furthermore, to ensure that simulation faithfully models the generated code, Simulink prevents you from changing the values of block parameters during simulation when the **Inline parameters** option is on.

Specifying Some Parameters as Nonlinear

Suppose that you want to take advantage of the **Inline parameters** optimization while retaining the ability to tune some of your model's parameters. You can do this by declaring some parameters as *nonlinear*, 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.

Note The documentation for the Real-Time Workshop 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.

Note 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. See “Tunable Workspace Parameter Data Type Considerations” for more information.

Using a Parameter Object to Specify a Parameter As Nonlinear.

If you use a parameter object to specify a parameter's value (see “Using Parameter Objects to Specify Parameter Values” on page 7-18), you can also use the object to specify the parameter as nonlinear. To do this, set the parameter object's `RTWInfo.StorageClass` property to any value but 'Auto' (the default).

```
K=Simulink.Parameter;  
K.RTWInfo.StorageClass = 'SimulinkGlobal';
```

If you set the `RTWInfo.StorageClass` property to any value other than `Auto`, you should not include the parameter in the tunable parameters table in the model's **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.

Block Properties Dialog Box

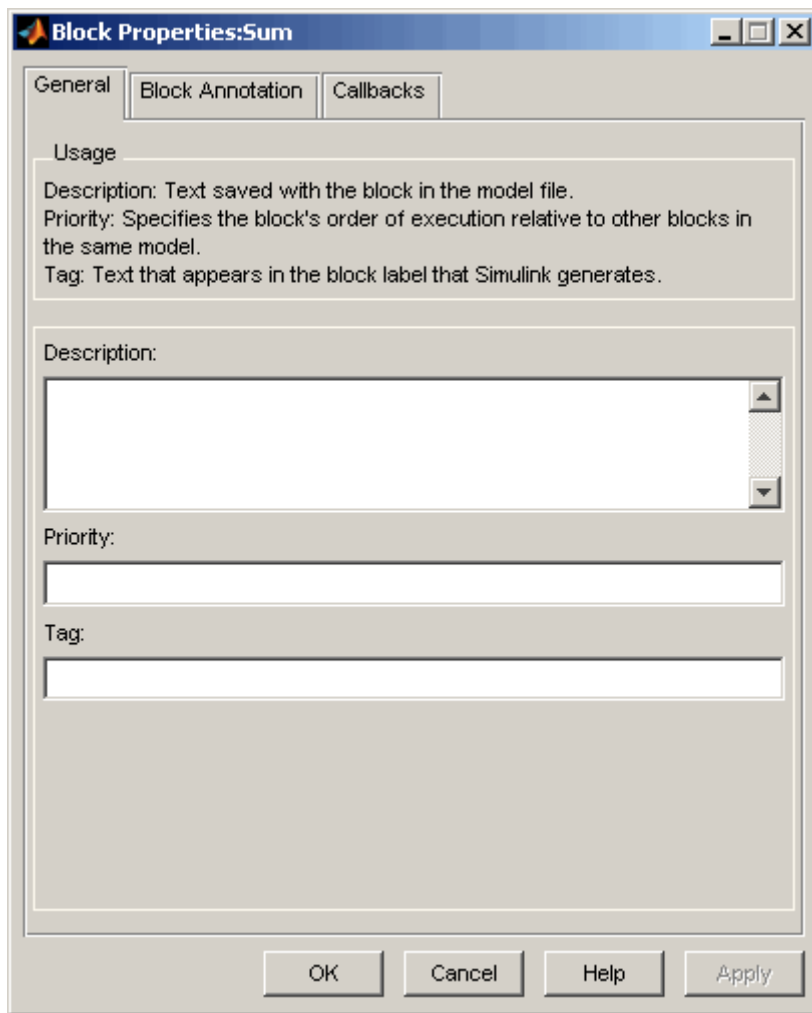
This dialog box lets you set a block's properties. To display this dialog, select the block in the model window and then select **Block Properties** from the **Edit** menu.

The dialog box contains the following tabbed panes:

- “General Pane” on page 7-27
- “Block Annotation Pane” on page 7-29
- “Callbacks Pane” on page 7-31

General Pane

This pane allows you to set the following properties.



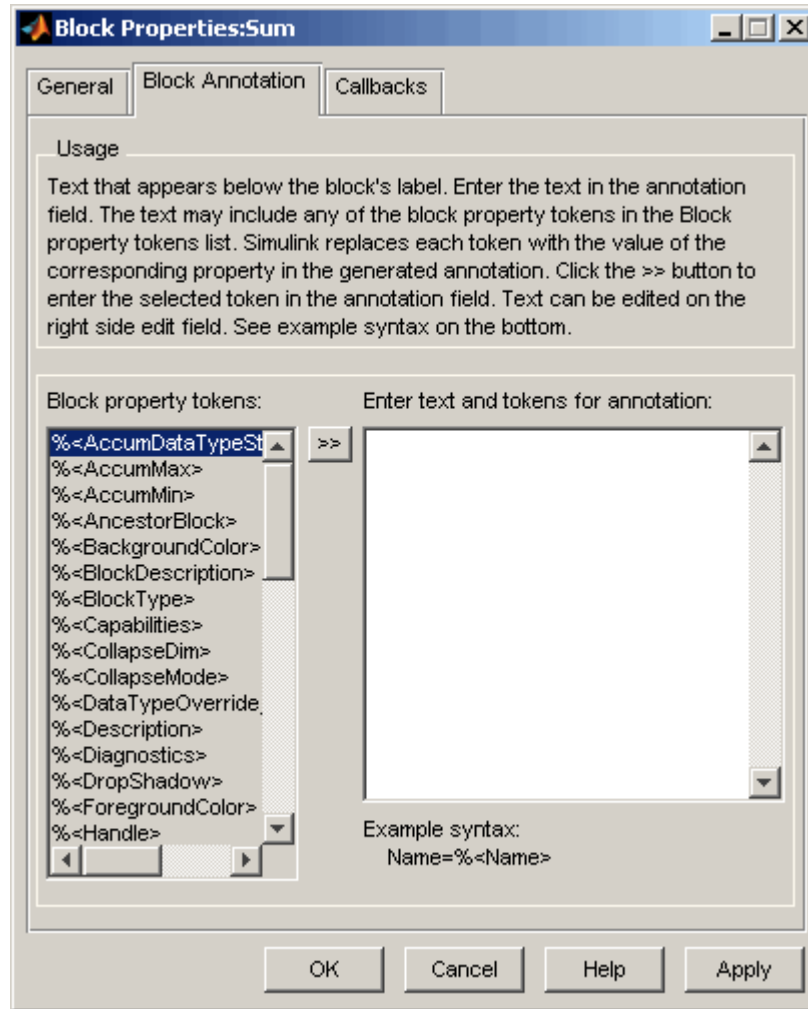
Description. Brief description of the block’s purpose.

Priority. Execution priority of this block relative to other blocks in the model. See “Assigning Block Priorities” on page 7-52 for more information.

Tag. Text that is assigned to the block's Tag parameter and saved with the block in the model. You can use the tag to create your own block-specific label for a block.

Block Annotation Pane

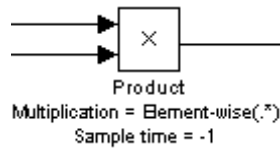
The block annotation pane allows you to display the values of selected block parameters in an annotation that appears beneath the block's icon.



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, the Simulink software 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 model editor window, the annotation appears as follows:

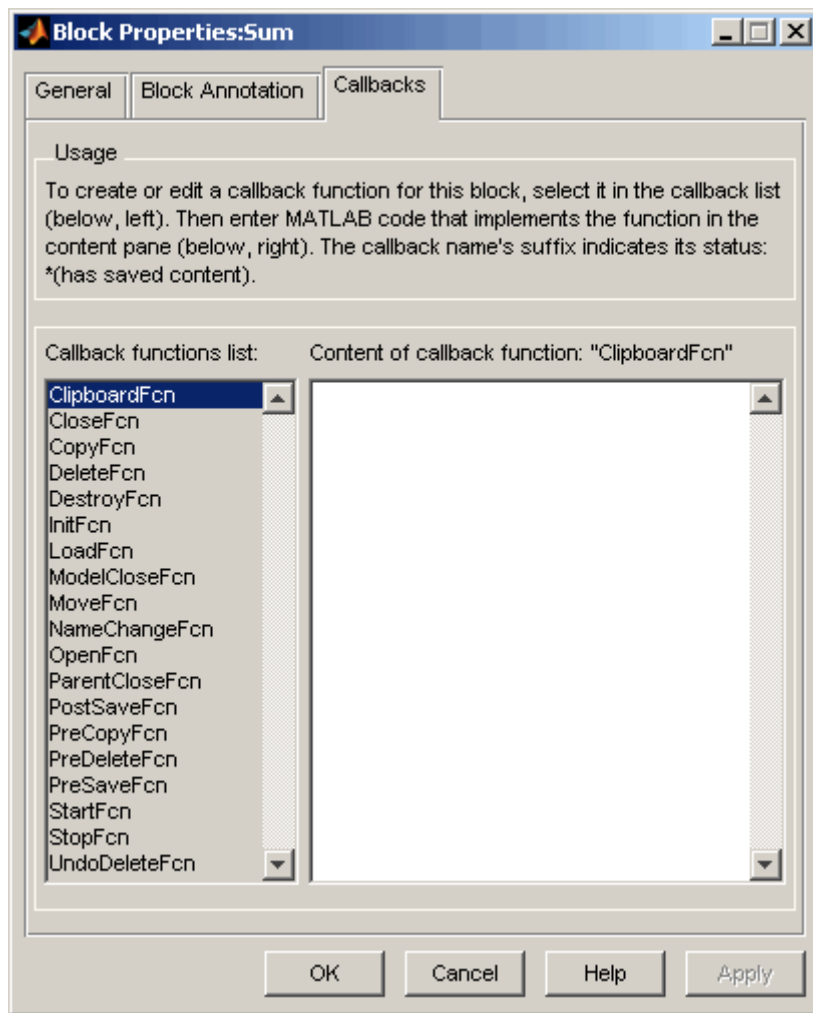


The block property token list on the left side of the pane lists all the parameters that are valid for the currently selected block (see “Model and Block Parameters” in the *Simulink Reference*). To add one of the listed tokens to the text field on the right side of the pane, select the token and then click the button between the list and the text field.

You can also create block annotations programmatically. See “Creating Block Annotations Programmatically” on page 7-33.

Callbacks Pane

The **Callbacks Pane** allows you to specify implementations for a block’s callbacks (see “Using Callback Functions” on page 4-54).



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.

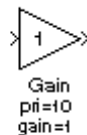
Creating Block Annotations Programmatically

You can use a block's `AttributesFormatString` parameter to display selected block parameters beneath the block as an “attributes format string,” i.e., a string that specifies values of the block's attributes (parameters). “Model and Block Parameters” in *Simulink Reference* describes 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, specifying the attributes format string

```
pri=%<priority>\ngain=%<Gain>
```

for a Gain block displays



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.

State Attributes Pane of Block Parameters Dialog Box

Use the **State Attributes** pane of a block parameters dialog box to specify Real-Time Workshop code generation options for blocks with discrete states. See “Block State Storage and Interfacing Considerations” in the *Real-Time Workshop User's Guide* for more information.

Changing a Block's Appearance

In this section...

“Changing a Block's Orientation” on page 7-34

“Resizing a Block” on page 7-37

“Displaying Parameters Beneath a Block” on page 7-38

“Using Drop Shadows” on page 7-38

“Manipulating Block Names” on page 7-39

“Specifying a Block's Color” on page 7-40

Changing a Block's 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 flipping it 180 degrees around its horizontal or vertical axis.

- “How to Rotate a Block” on page 7-34
- “How to Flip a Block” on page 7-36

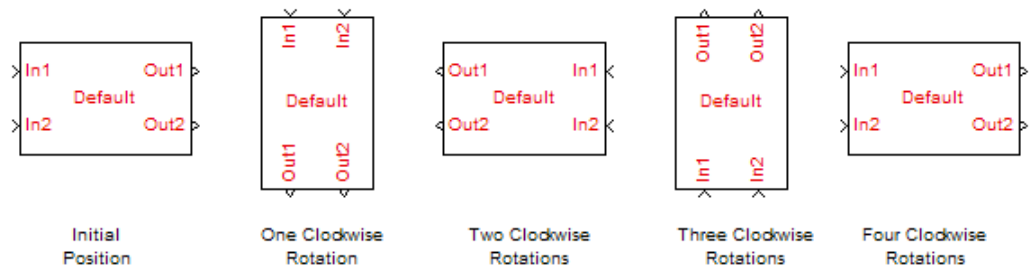
How to Rotate a Block

You can rotate a block 90 degrees by selecting one of these commands from the **Format** menu:

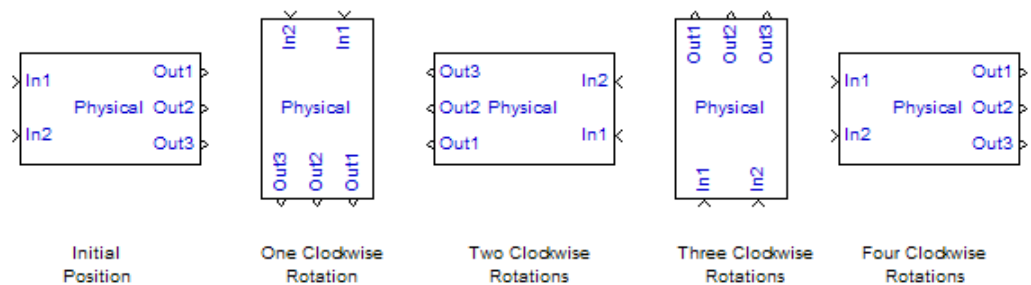
- **Rotate Block > Clockwise (Ctrl+R)**
- **Rotate Block > Counterclockwise**

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's 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 Block > Clockwise** command on a block with the default rotation policy.



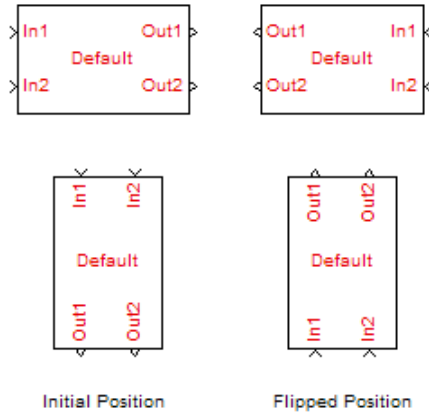
A masked block can optionally specify that its ports not be reordered after a clockwise rotation (see “Port Rotation”). Such a block is said to have a *physical port rotation type*. This policy facilitates layout of diagrams in mechanical and hydraulic systems modeling and other applications where diagrams 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

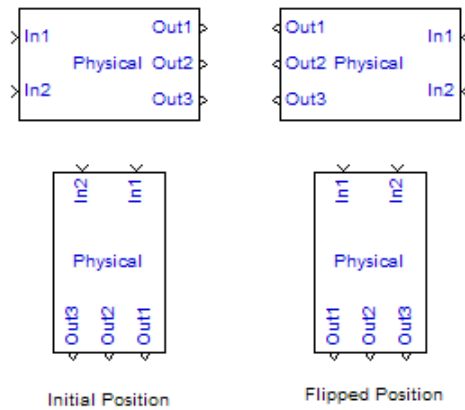
Simulink provides a set of commands that allow you to flip a block 180 degrees about its horizontal or vertical axis. The commands effectively move a block's input and output ports to opposite sides of the block or reverse the ordering of the ports, depending on the block's port rotation type.

A block with the default rotation type has one flip command: **Format > 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.

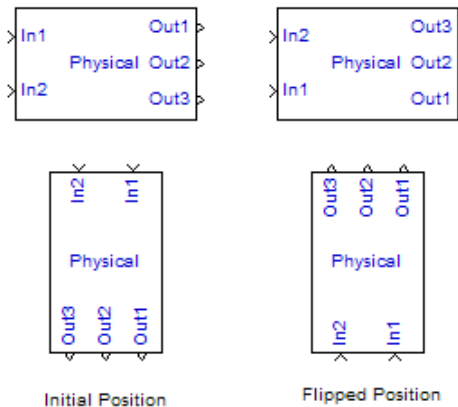


A block with a physical rotation type has two flip commands: **Format > Flip Block > Left-Right** or **Format > Flip Block > Up-Down**. The effect of these commands depends on the initial position of the ports.

Left-Right Flip. If the block's ports initially reside on the left and right sides of the block, the **Left-Right** command reverses the sides on which the input and ports are located. If the ports are located on the top and bottom of the block, the command reverses the order of the ports without changing their sides.



Up-Down Flip. If the block's ports reside on the top and bottom of the block, the **Up-Down** command reverses the sides on which the input and ports are located. If the ports are located on the left and right sides of the block, the command reverses the order of the ports without changing their sides.

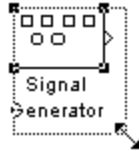


Resizing a Block

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.

This figure shows a block being resized:



Tip Use the model editor's resize blocks commands to make one block the same size as another (see "Aligning, Distributing, and Resizing Groups of Blocks Automatically" on page 4-24).

Displaying Parameters Beneath a Block

You can cause Simulink to display one or more of a block's parameters beneath the block. You specify the parameters to be displayed in the following ways:

- By entering an attributes format string in the **Attributes format string** field of the block's **Block Properties** dialog box (see "Block Properties Dialog Box" on page 7-27)
- By setting the value of the block's `AttributesFormatString` property to the format string, using `set_param`

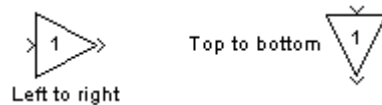
Using Drop Shadows

You can add a drop shadow to a block by selecting the block, then choosing **Show Drop Shadow** from the **Format** menu. When you select a block with a drop shadow, the menu item changes to **Hide Drop Shadow**. The following figure shows a Subsystem block with a drop shadow:



Manipulating 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, i.e., /, 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.

Changing Block Names

You can edit a block name in one of these ways:

- To replace the block name, click the block name, double-click or drag the cursor to select the entire name, 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.

When you click the pointer anywhere else in the model or take any other action, the name is accepted or rejected. If you try to change the name of a block to a name that already exists or to a name with no characters, Simulink displays an error message.

You can modify the font used in a block name by selecting the block, then choosing the **Font** menu item from the **Format** menu. Select a font from the **Set Font** dialog box. This procedure also changes the font of any text that appears inside the block.

You can cancel edits to a block name by choosing **Undo** from the **Edit** menu.

Note If you change the name of a library block, all links to that block become unresolved.

Changing the Location of a Block Name

You can change the location of the name of a selected block in two ways:

- By dragging the block name to the opposite side of the block.
- By choosing the **Flip Name** command from the **Format** menu. 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 7-34.

Changing Whether a Block Name Appears

To change whether the name of a selected block is displayed, choose a menu item from the **Format** menu:

- The **Hide Name** menu item hides a visible block name. When you select **Hide Name**, it changes to **Show Name** when that block is selected.
- The **Show Name** menu item shows a hidden block name.

Specifying a Block’s Color

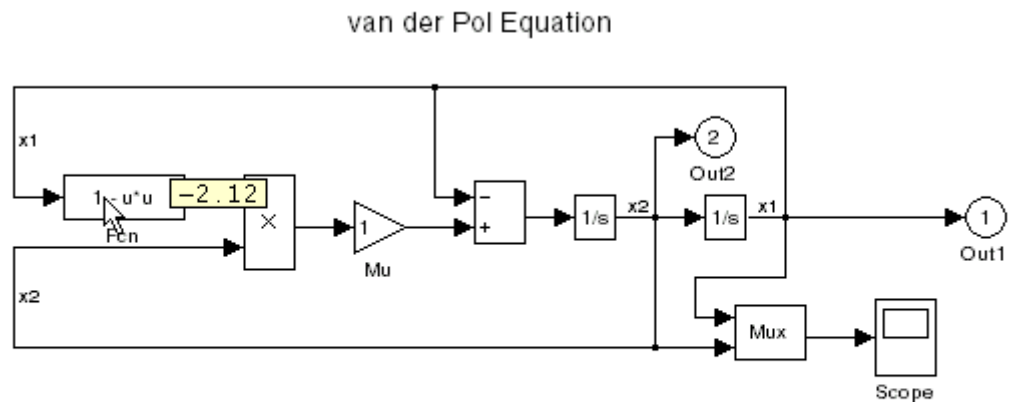
See “Specifying Block Diagram Colors” on page 4-9 for information on how to set the color of a block.

Displaying Block Outputs

In this section...
“Block Output Data Tips” on page 7-41
“Setting Block Output Data Tip Options” on page 7-42
“Enabling Block Output Display” on page 7-42
“Controlling the Block Output Display” on page 7-43
“Port Value Display Limitations” on page 7-44

Block Output Data Tips

For many blocks, Simulink can display block output (port values) 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.



The van der Pol Equation
(Double-click on the '?' for more info)



Double-click
here for
Simulink Help

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

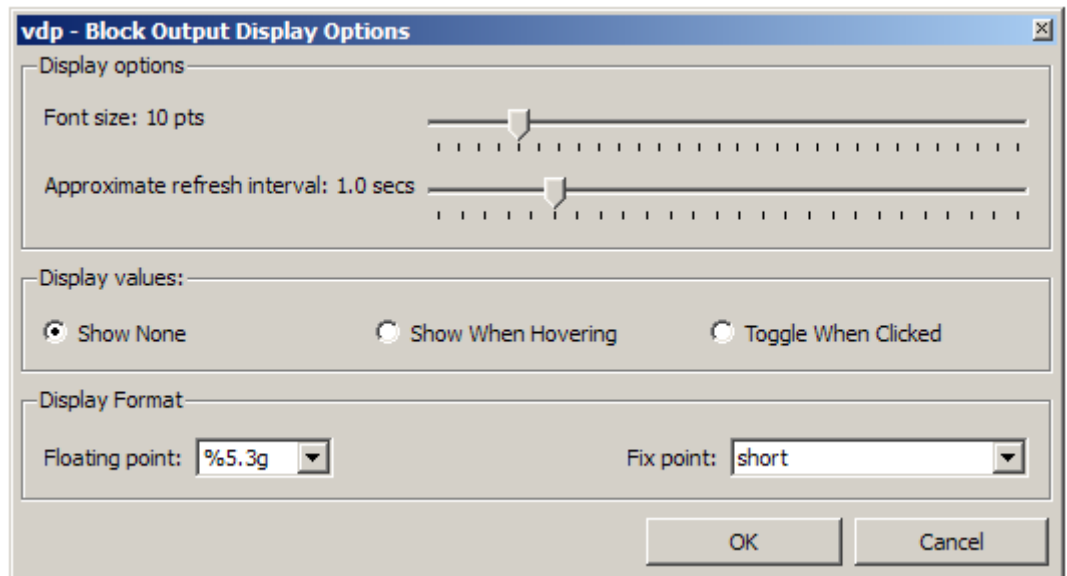
Note The block output display feature has limitations for models with:

- Accelerated modes
- Signal storage reuse
- Signals with some complex data types, such as bus signals

See “Port Value Display Limitations” on page 7-44.

Setting Block Output Data Tip Options

You can set data tip options by selecting **Port Values > Options** from the Model Editor **View** menu and using the Block Output Display Options dialog.



Enabling Block Output Display

To turn block output display on or off, select one these options:

- **Show None**

Turns off block output display.

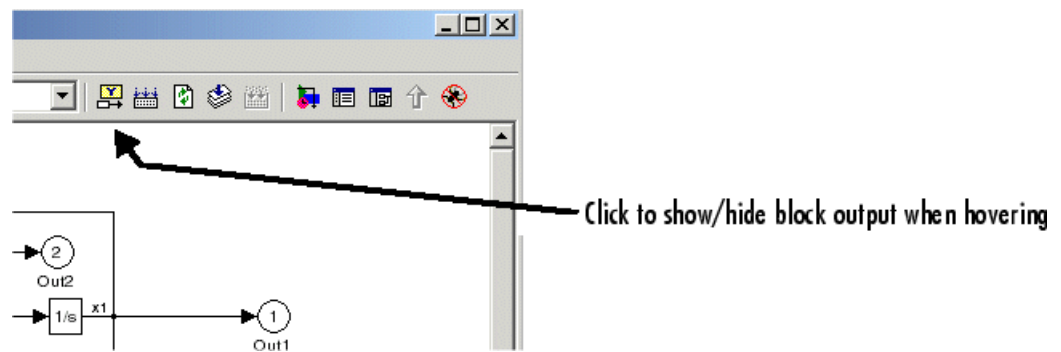
- **Show When Hovering**

Displays output port values for the block under the mouse cursor.

- **Toggle When Clicked**

Displays output port values when you select a block. Reselecting that block turns off the display.

The Microsoft Windows operating system Simulink version also has a Model Editor output display button to control block output display when hovering.



Controlling the Block Output Display

You can specify how the block output display is formatted and how frequently the display updates.

Block Output Data Tip Display Characteristic	What You Specify
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 Display Format list.

Block Output Data Tip Display Characteristic	What You Specify
Fixed-point data format	Choose a fixed-point format from the Display Format 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 Refresh interval slider to the right.

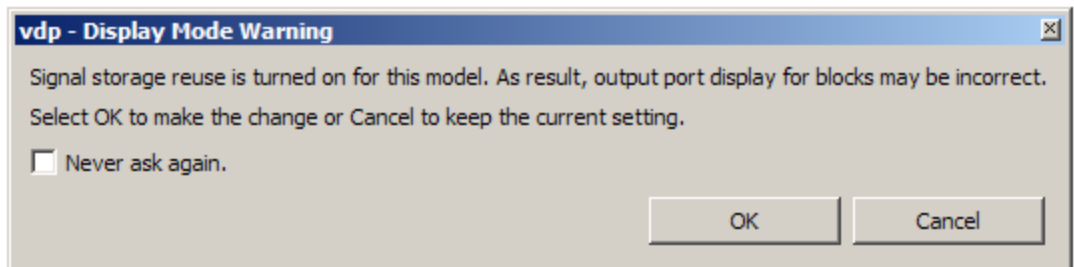
Port Value Display Limitations

Accelerated Modes

Port values do not update in a model that simulates in any accelerated mode. The limitation exists whether the model itself specifies accelerated simulation or the model is subordinate to a model that specifies accelerated simulation. See Chapter 27, “Accelerating Models”, “Accelerator Mode” on page 6-12, and “Testing and Refining Concept Models With Standalone Rapid Simulations” for more information.

Signal Storage Reuse

A model with **Signal storage reuse** enabled displays a warning when you set **Display values** to Show When Hovering or Toggle When Clicked .



Click the **OK** button if you want the new Display values option to take effect, despite the possibility that values for output ports could be incorrect.

Signal Data Types

The port value displays 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, no data displays for signals with some complex data types, such as bus signals.

Controlling and Displaying the Sorted Order

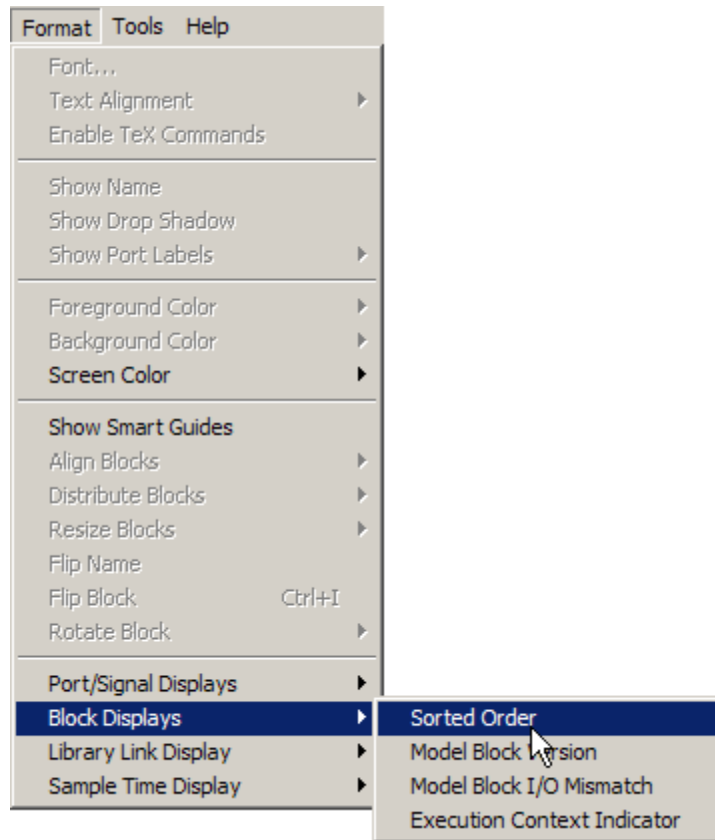
In this section...
“What Is Sorted Order?” on page 7-46
“Displaying the Sorted Order” on page 7-46
“Sorted Order Notation” on page 7-48
“How Simulink Determines the Sorted Order” on page 7-51
“Assigning Block Priorities” on page 7-52
“Rules for Priorities” on page 7-53

What Is Sorted Order?

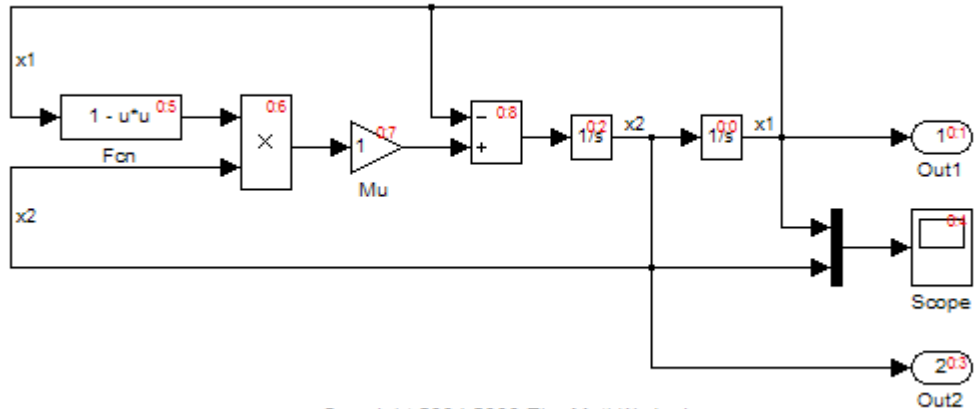
During the initialization phase of simulation, Simulink determines the order in which to invoke the block methods (see “Block Methods” on page 2-17) during simulation. This ordering is the *sorted order*. You cannot set this order, but you can assign priorities to nonvirtual blocks to indicate to Simulink their relative order. Simulink then tries to honor your priority settings when it creates the sorted order according to the sorting rules. To confirm the results of any priorities that you have set or to debug your model, you can display and review the sorted order of your nonvirtual blocks and subsystems.

Displaying the Sorted Order

To display the sorted order of a system, from the **Format** menu, select **Block Displays > Sorted Order**. As a result, Simulink displays a notation in the top right corner of each nonvirtual block and each nonvirtual subsystem in the block diagram.



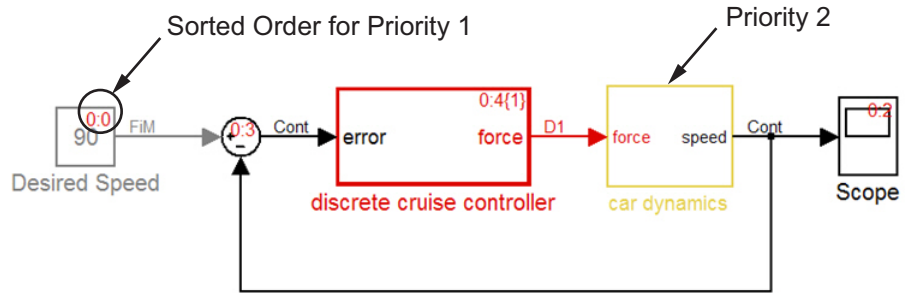
The following figure shows the sorted order for the van der Pol Equation model. The integrator with the sorted order 0:0 runs first followed by Out1 of order 0:1. The remaining blocks run in numeric order from 0:2 to 0:8.



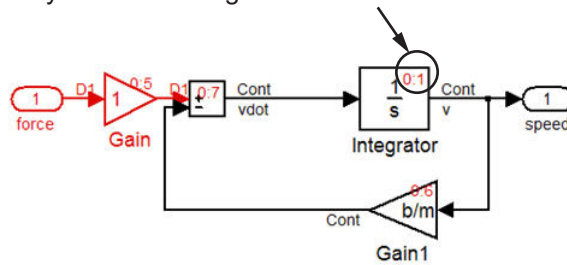
Copyright 2004-2008 The MathWorks, Inc.

Sorted Order Notation

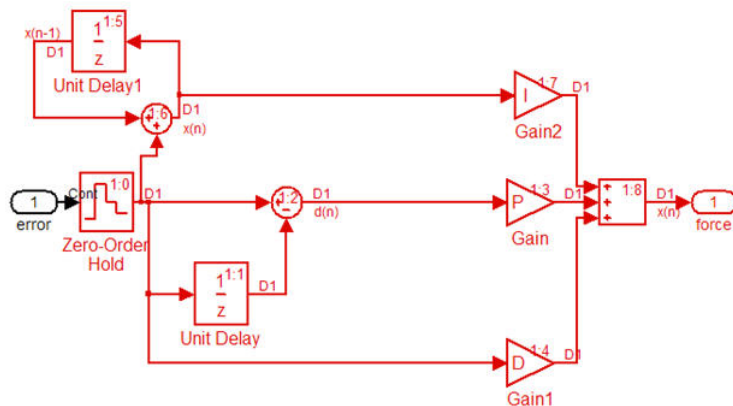
The following model consists of a root-level system and two subsystems — an atomic, nonvirtual subsystem named “Discrete Cruise Controller” and a virtual subsystem called “Car Dynamics.” Both subsystems produce graphical hierarchy in the model. The atomic subsystem also provides execution hierarchy. (See “Creating Subsystems” on page 4-37.) The sorted order of the blocks is shown for the case when the Desired Speed (Constant) block is set with a priority of one and the Car Dynamics (virtual) subsystem is set with a priority of two.



Priority 2 causes Integrator block to run second



Car Dynamics Subsystem



Discrete Cruise Controller Subsystem

The sorted order notation for most blocks has the format $s:b$, where s is a system index that prefixes all blocks belonging to the execution context of that system. (See “Conditional Execution Behavior” on page 5-25 for more information.) The b specifies the block position within the sorted order for the same execution context. For nonvirtual subsystems, the notation becomes $s:b\{s_i\}$ where s and b have the same meaning as before and where s_i represents the index of the execution context of a given subsystem.

For example the sorted order $0:4\{1\}$ appears on the Discrete Cruise Controller subsystem. The 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. The 4 indicates that the atomic subsystem is the fifth block that Simulink executes relative to the blocks within the root level. And the 1 represents the subsystem index. This index does not indicate the relative order in which the atomic subsystem runs. Like s , s_i is a means of identifying or referencing the subsystem. Also, s_i becomes s within the respective subsystem; each of the sorted orders in the Discrete Cruise Controller subsystem are of the form $1:b$.

Because the sorted order of a Function-Call Subsystem cannot be determined at compile time, Simulink replaces the block position b with a single letter. 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 a subsystem that connects to more than one initiator, s is no longer unique; therefore, Simulink uses the letter notation $M:F\{s_i\}$.

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 the letter B stands for “bus.” See “Bus-Capable Blocks” on page 12-9 for more information.

Virtual blocks, such as the Mux block, exist only graphically and do not execute. Consequently, they are not part of a sorted order and do not display any sorted order notation. Similarly, the “Car Dynamics” subsystem is a virtual subsystem and thus is only a graphical entity created for organizational purposes. Unlike an atomic subsystem, a virtual subsystem does not execute as a unit and thus, like a virtual block, is not part of the sorted order. Instead, the blocks within the virtual subsystem are part of the root-level system sorted order, and therefore share the 0 index.

How Simulink Determines the Sorted Order

Simulink uses the following basic rules to sort blocks:

- If a block drives the direct-feedthrough port of another block, it must appear in the sorted order ahead of the block it drives. (See “About Direct-Feedthrough Ports” on page 7-51.)

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 which 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.

The result of applying these rules is a sorted order in which blocks without direct-feedthrough ports appear at the beginning of the list in no particular order, followed by blocks with direct-feedthrough ports arranged such that they can supply valid inputs to the blocks which they drive. This is illustrated in the previous block diagrams (Car Dynamics Subsystem on page 7-49). The Integrator and the Constant blocks do not have direct-feedthrough and thus appear at the beginning of the sorted order of the root-level system. Also, all of the Gain blocks, which have direct-feedthrough ports, run before the Sum blocks that they drive.

About Direct-Feedthrough Ports

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 its inputs. 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 the Gain, Product, and Sum blocks. Examples of blocks that have non-direct-feedthrough inputs include the Integrator block (its output is a function purely of its state), the Constant block (it does not have an input), and the Memory block (its output is dependent on its input from the previous time step).

About Algebraic Loops

During the sorting process, Simulink checks for and flags the occurrence of algebraic loops, that is, signal loops in which a direct-feedthrough output of a block connects directly or indirectly to a direct-feedthrough input of the same block. Such loops seemingly create a deadlock condition because the block needs the value of the direct-feedthrough input to compute its output.

However, an algebraic loop can represent a set of simultaneous algebraic equations (hence the name) where the block inputs and outputs are the unknowns. These equations can have valid solutions at each time step. Accordingly, Simulink assumes that loops involving direct-feedthrough ports represent a solvable set of algebraic equations. Simulink attempts to solve the equations each time it needs the block output during a simulation. For more information, see “Algebraic Loops” on page 2-35.

Assigning Block Priorities

You can assign a priority to a nonvirtual blocks 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.

You can assign block priorities programmatically or interactively.

- To set priorities programmatically, use the command

```
set_param(b, 'Priority', 'n')
```

where *b* is the block path and *n* is any valid integer. (Negative integers and 0 are valid priority values.)

- To set the priority of a block interactively, enter the priority in the **Priority** field of the **Block Properties** dialog box. (See “Block Properties Dialog Box” on page 7-27.) You can interactively set the priority of any subsystem using the same method

Returning to the car model, the Constant block (Desired Speed) has been set to priority 1 by the user, while the virtual subsystem has been set to priority 2. Since the Constant block has a higher priority, it runs before the Integrator

block of the subsystem. Whereas if you do not assign any priorities, the Integrator block runs first, followed by the Scope block and then the Constant block.

Rules for Priorities

Simulink honors the block priorities that you specify only if they are consistent with the Simulink block sorting algorithm. In assessing your priority assignments, Simulink attempts to create a sorted order such that the priorities you set for the individual blocks within the root system or within a nonvirtual subsystem are honored relative to one another. Consider, for example, if you set the following priorities in the car model:

In Discrete Cruise Controller:

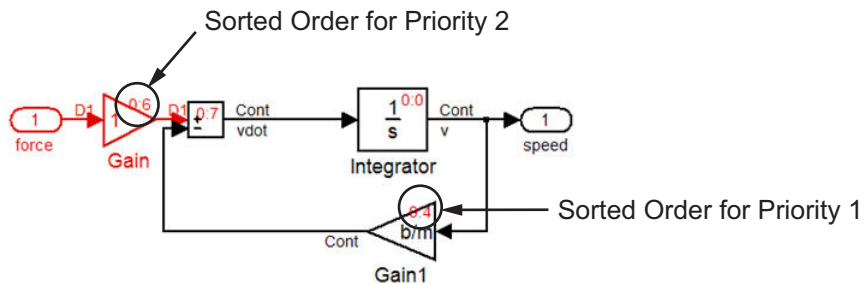
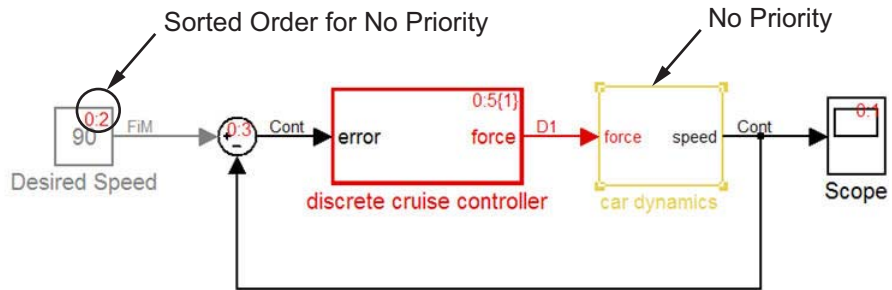
Block	Priority
Gain	3
Gain1	2
Gain2	1

In Car Dynamics:

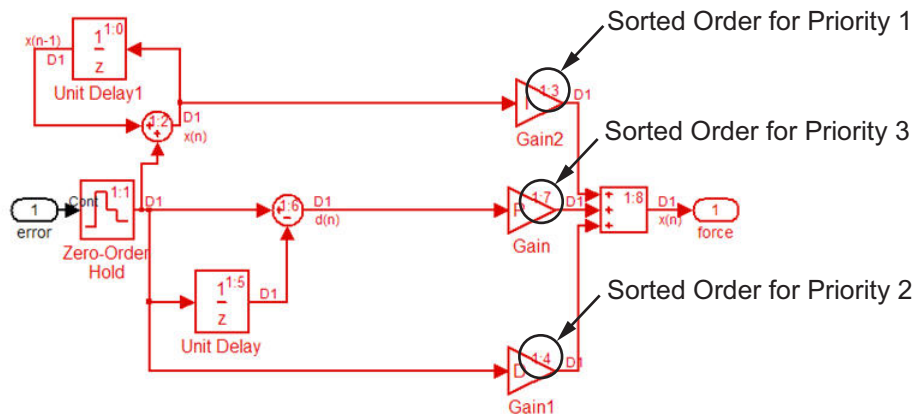
Block	Priority
Gain	2
Gain1	1

The results, as shown below, illustrate the three rules pertaining to priorities:

- Priorities are relative: The priority of a block is relative to the priority of the blocks within the same system or subsystem.
- Priorities are hierarchal.
- A lack of priority does not necessarily result in a low priority (higher sorting order) for a given block.



Car Dynamics Subsystem



Discrete Cruise Controller Subsystem

The application of the first rule is evident in the Discrete Cruise Controller subsystem. The sorted order values of the Gain, Gain1, and Gain2 blocks reflect the respective priorities assigned: Gain2 has highest priority and comes first; Gain1 has second priority and comes second. However note that the Gain blocks are not the first, second, and third blocks to run. Nor do they have consecutive sorted orders. In other words, the sorted order values do not necessarily correspond to the priority values; however, Simulink arranges the blocks such that their priorities are honored relative to each other.

The sorted orders of the two subsystems illustrate the result of the second rule. You are allowed to 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 with those of any other subsystem. Thus the atomic subsystem has an independent sorted order with the blocks arranged consecutively from 0 to 8. In contrast, the blocks within the virtual subsystem are part of the primary or root system hierarchy and thus are part of the root-level sorted order.

Because of the third rule, the Integrator block in the Car Dynamics subsystem has no priority yet runs first. Similarly the Unit Delay, the Zero-Order Hold, and the Unit Delay1 blocks have no priority, yet they all execute before the Gain blocks. Simulink places these blocks first because they do not have direct-feedthrough ports.

If a model has two atomic subsystems A and B, you can assign priorities of 1 and 2 respectively to A and B and thereby cause all of the blocks in A to run before any of the blocks in B. (Recall that the blocks within an atomic subsystem execute as a single unit, so the subsystem has its own sorted order.)

Finally, if Simulink is unable to honor a block priority, it displays a **Block Priority Violation** diagnostic message.

Accessing Block Data During Simulation

In this section...

“About Block Run-Time Objects” on page 7-56

“Accessing a Run-Time Object” on page 7-56

“Listening for Method Execution Events” on page 7-57

“Synchronizing Run-Time Objects and Simulink Execution” on page 7-58

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 M-file S-functions (see “Writing S-Functions in M” 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 “Working with Data Objects” on page 13-26) 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.

Accessing 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.

Note Virtual blocks (see “Virtual Blocks” on page 7-2) 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 or pause a model, all existing handles for run-time objects become empty.

Listening 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 *Simulink Reference* 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

Run-time objects can be used at the MATLAB command line to obtain the value of a block's output by entering the following commands.

```
rto = get_param(gcf, 'RuntimeObject')
rto.OutputPort(1).Data
```

However, the displayed data may not be the block's true 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 M-file S-function or in an event listener callback. When called at the MATLAB command line, 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, turn off the **Signal storage reuse** option (see “Signal storage reuse”) on the **Optimization** pane in the **Configuration Parameters** dialog box.

Working with Block Libraries

- “About Block Libraries” on page 8-2
- “Working with Reference Blocks” on page 8-3
- “Working with Library Links” on page 8-6
- “Creating Block Libraries” on page 8-14
- “Adding Libraries to the Library Browser” on page 8-24

About Block Libraries

A *block library* is a collection of blocks that serve as prototypes for cloning blocks to Simulink models. Simulink provides a Library Browser that you can use to display block libraries, search for blocks by name, and clone library blocks into models. All installed libraries appear in the Library Browser when you open it. See “Populating a Model” on page 4-4 for information about how to use the Simulink Library Browser.

Simulink comes with two built-in block libraries: the Simulink block library and the Real-Time Workshop block library. The latter is included with Simulink to support sharing models that contain Real-Time Workshop blocks. Many additional products and associated block libraries are available from The MathWorks™. See the MathWorks Web Site for product information. You cannot change a built-in block library in any way.

When you clone a block from a library into a model, Simulink does not copy the library block itself. Instead, Simulink places a *reference block* in the model, and connects the reference block to the library block using a *library link*. The library block is then the *prototype block*, and the prototype representation in the model (via the reference block) is a *block instance*. The appearance and behavior of a linked block are the same as if you had actually copied it to the model. For most purposes, you can ignore the underlying and link and reference block structure and just think of a block cloned from a library as a *block instance*.

Working with Reference Blocks

In this section...

“About Reference Blocks” on page 8-3

“Creating a Reference Block” on page 8-3

“Updating a Reference Block” on page 8-4

“Modifying Reference Blocks” on page 8-4

“Finding a Reference Block’s Library Block Prototype” on page 8-5

“Getting Information About Library Blocks Referenced by a Model” on page 8-5

About Reference Blocks

A *reference block* is an instance of a block type in a model that contains a link to a 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 reference block to update whenever the corresponding prototype in the library changes (“Updating a Reference Block” on page 8-4). This ensures that your model always uses the latest version of the block.

Note The data tip for a reference block shows the name of the library block it references (see “Block Data Tips” on page 7-2).

You can change the values of a reference block’s parameters but you cannot mask the block or edit its mask. Also, you cannot set callback parameters for a reference block. If the reference block’s prototype is a subsystem, you can make nonstructural changes to the contents of the referenced subsystem (see “Modifying Reference Blocks” on page 8-4).

Creating a Reference Block

To create a reference block in a model or another library:

- 1 Open your model.

- 2** Open the Simulink Library Browser (see “About the Library Browser” on page 4-4).
- 3** Use the Library Browser to find the library block that serves as a prototype of the block you want to create (see “Browsing Block Libraries” on page 4-5 and “Searching Block Libraries” on page 4-5).
- 4** Drag the library block from the Library Browser’s **Library** pane and drop it into your model.

Updating a Reference Block

Simulink updates out-of-date reference blocks in a model or library at these times:

- When the model or library is loaded
- When you select **Update Diagram** from the **Edit** menu or run the simulation
- When you use the `find_system` command
- When you query the `LinkStatus` parameter of a block, using the `get_param` command (see “Determining Link Status” on page 8-10)

Note Querying the `StaticLinkStatus` parameter of a block does not update any out-of-date reference blocks.

Modifying Reference Blocks

You cannot make structural changes to reference 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 reference block’s link to its library prototype (see “Disabling Links to Library Blocks” on page 8-7).

You can, however, change the values of any masked subsystem reference 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 link to a library block from a reference block whose parameter values differ from those of the corresponding library block is

called 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 model (.mdl) file. When you reopen the system, Simulink copies the library subsystem into the loaded model and applies the saved changes.

Tip To determine whether a reference block's parameter values differ from those of its library prototype, open the reference block's block diagram in an editor window. The title bar of the editor window displaying the subsystem displays "parameterized link" if the reference block parameter values differ from the library block's parameter values.

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.

Finding a Reference Block's Library Block Prototype

To find the source library and block linked to a reference block, select the reference block. Then choose **Go To Library Link** from the **Link Options** submenu of the model window's **Edit** or context menu. 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 opens it and selects the library block.

Getting Information About Library Blocks Referenced by a Model

Use the `libinfo` command to get information about reference blocks in a system.

Working with Library Links

In this section...

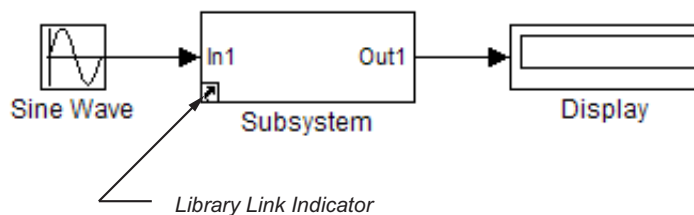
- “Displaying Library Links” on page 8-6
- “Disabling Links to Library Blocks” on page 8-7
- “Restoring Disabled or Parameterized Links” on page 8-7
- “Determining Link Status” on page 8-10
- “Breaking a Link to a Library Block” on page 8-11
- “Fixing Unresolved Library Links” on page 8-12

Displaying 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, and from the **Format** menu, select **Library Link Display**.
- 2** From the submenu, select either **User** (displays only links to user libraries) or **All** (displays all links).

The library link indicator is an arrow in the bottom left corner of each block.



The color of the link arrow indicates the status of the link.

Color	Status
Black	Active link
Grey	Inactive link
Red	Active and modified (parameterized link)

Disabling Links to Library Blocks

To make a structural change to a linked subsystem block, you need to disable the link between the block and the library block that serves as its prototype.

Note When you use the Model Editor to make a structural change (such as editing the diagram) to a local copy of a subsystem block with an active library link, Simulink offers to disable the library link for you. 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 change is undefined.

To disable a link:

- 1 In the Model Editor window, select a subsystem block.
- 2 From the **Edit** menu, select **Link Options**, and then select **Disable link**.

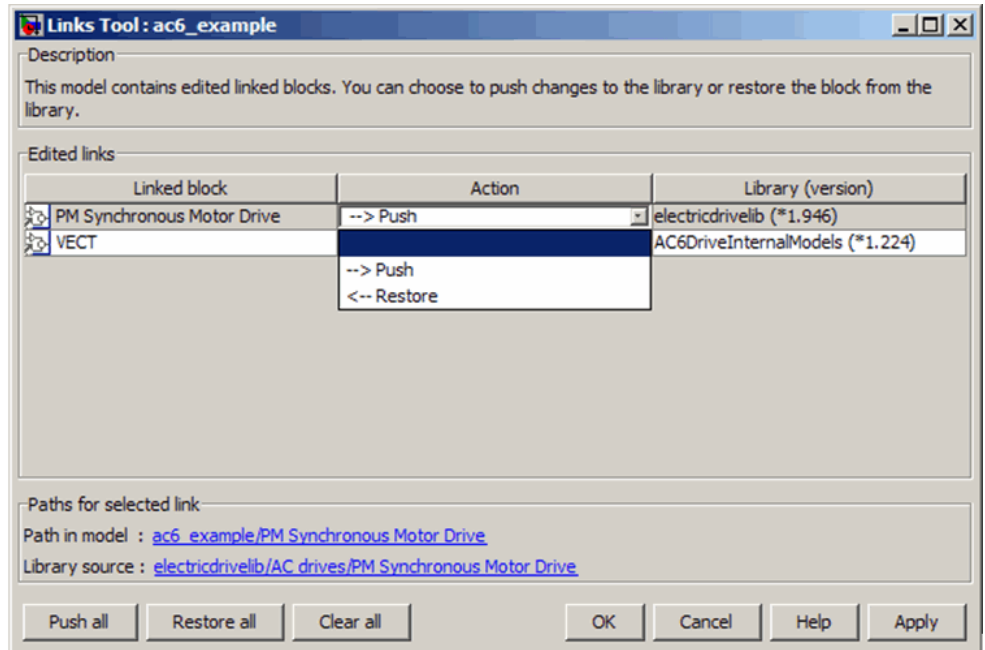
The model breaks the library link and grays out the library link indicator. When a library block is disabled and it is within another library block (a child of a parent library block), the model also disables the parent block containing the child block.

Restoring Disabled or Parameterized Links

After you make structural changes to a linked subsystem block, you need to restore the link to its 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 subsystem block with a disabled library link.
- 2 From the **Edit** menu, select **Link Options**, and then select **Resolve link**.

The Links Tool window opens.



The table in the Edited links panel has the following columns:

- **Linked block** — List of linked blocks. The list of edited links includes library links with structural changes, parameterized library links, and library links that were actively chosen to be resolved.
 - **Action** — Choose an action to perform on the linked block or library.
 - **Library** — List of library names and version numbers.
- 3 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.

- 4 From the **Action** list, choose one of two actions.


Action Choice	Simulink Action
Push	Updates the library version with the changes you made in the model version of that subsystem.
Restore	Replaces the version of the subsystem in the model with the version in the library.

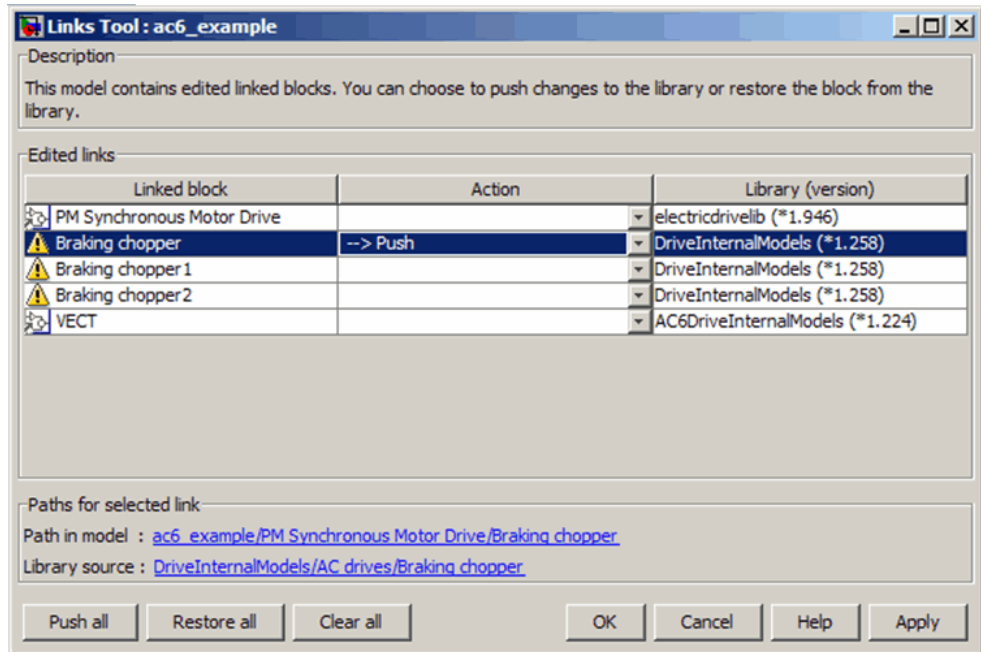
Choosing the **Push all**, **Restore all** or **Clear all** button selects an action for all linked blocks.

- 5 Click **OK** or **Apply**.

Simulink restores the library links to the selected subsystems in the model. The subsystem versions in the library and the model now match. If the link for a child that was structurally changed is restored, and there were no structural changes with its parent, the parent link is also restored.

Changes pushed 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 (In this case, Braking chopper, Braking chopper 1, or Braking chopper 2) 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.



Determining Link Status

All blocks have a `LinkStatus` parameter and a `StaticLinkStatus` parameter that indicate whether the block is a reference block. The parameters can have these values.

Note Using `get_param` to query a block's `LinkStatus` also resolves any out-of-date block references. It is, therefore, useful to update library links in a model programmatically. Conversely, querying the `StaticLinkStatus` property does not resolve any out-of-date references. 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.

Status	Description
none	Block is not a reference block.

Status	Description
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	Link disabled.
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. The function <code>get_param</code> never return <code>Restore</code> . It is only used with <code>set_param</code> .
Propagate	Restores an inactive or disabled link to a library block and pushes any changes made to the local copy to the library. The function <code>get_param</code> never return <code>Propagate</code> . It is only used with <code>set_param</code> .

Breaking a Link to a Library Block

You can break the link between a reference block and its library block to cause the reference 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 Example” on page 9-5 model as a standalone model, without the libraries.

To break the link between a reference block and its library block, first disable the link. Then select the block and choose **Break Link** from the **Link Options** menu. You can also break the link between a reference block and its library block from the command line by changing the value of the `LinkStatus` parameter to `'none'` using this command:

```
set_param('refblock', 'LinkStatus', 'none')
```

You can also break links to library blocks when saving the model, by supplying arguments to the `save_system` command. See `save_system` in the Simulink reference documentation.

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 third-party libraries and optional Simulink blocksets.

Fixing 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 reference 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
"reference-block-path".
```

The unresolved reference block appears like this (colored red).



To fix a bad link, you must do one of the following:

- Delete the unlinked reference block and copy the library block back into your model.
- Add the directory that contains the required library to the MATLAB path and select **Update Diagram** from the **Edit** menu.
- Double-click the unlinked reference block to open its dialog box (see the Bad Link block reference page). On the dialog box that appears, correct the pathname in the **Source block** field and click **OK**.

Creating Block Libraries

In this section...
“Creating a Library” on page 8-14
“Creating a Sublibrary” on page 8-15
“Modifying a Library” on page 8-15
“Locking Libraries” on page 8-16
“Making Backward-Compatible Changes to Libraries” on page 8-16

Creating a Library

You can create your own block library and add it to the Simulink Library Browser (see “Adding Libraries to the Library Browser” on page 8-24).

Tip If your library contains many blocks, consider grouping the blocks into a hierarchy of sublibraries (see “Creating a Sublibrary” on page 8-15).

To create a library:

- 1 Select **Library** from the **New** submenu of the **File** menu.

Simulink creates a model (*.mdl) 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 must provide a mask (see Chapter 9, “Working with Block Masks”) for the block. You can also provide a mask for a subsystem in a library but you do not need to do so in order to create links to it in models.

- 3 Save the library’s model file under a new name.

Creating a Sublibrary

Creating a sublibrary entails inserting a reference in the model (.mdl) 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 Unlock the parent library (see “Modifying a Library” on page 8-15).
- 3 Add a Subsystem block to the parent library.
- 4 Delete the subsystem’s default input and output ports.
- 5 Create a mask for the subsystem that displays text or an image that conveys the sublibrary’s purpose.
- 6 Set the subsystem’s OpenFcn parameter to the name of the sublibrary’s model file.
- 7 Save the parent library.

Modifying a Library

When you open a library, it is automatically locked and you cannot modify its contents. To unlock the library, select **Unlock Library** from the **Edit** menu. Closing the library window locks the library.

Locking Libraries

To lock a block library, save and close the library or set its Lock parameter to 'on' at the MATLAB command line, using the `set_param` command. 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. You must then relock the library from the MATLAB command line to prevent further changes.

Making 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

Library forwarding tables enable Simulink to update models to reflect changes in the names or locations of the library blocks that they reference. For example, suppose that you rename a block in a library. You can use a forwarding table for that library to enable Simulink to update models that reference the block under its old name to reference it under its new name.

Simulink allows you to associate a forwarding table with any library. The forwarding table for a library specifies 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.

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 “Example: Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes” on page 8-17).

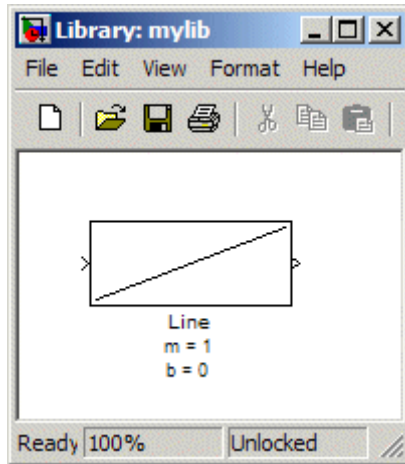
To create aliases for a masked block's mask parameters, use the `set_param` command to set the block's `MaskVarAliases` parameter to a cell array that specifies the names of the aliases in the same order as the mask names appear in the block's `MaskVariables` parameter.

Example: 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.

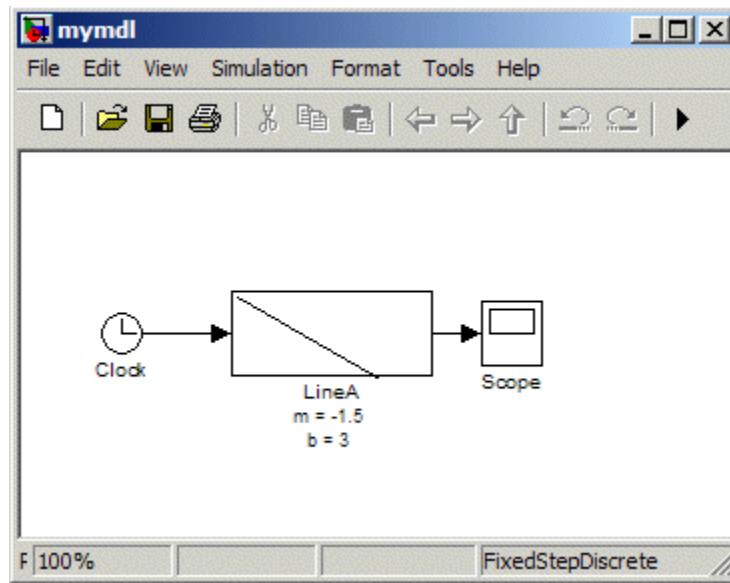
- 1 Create a library named `mylib`.
- 2 Create the masked subsystem described in “Masked Subsystem Example” on page 9-5 in `mylib`.
- 3 Name the masked subsystem `Line`.
- 4 Set the masked subsystem's annotation property (see “Block Annotation Pane” on page 7-29) to display the value of its `m` and `b` parameters, i.e., to

```
m = %<m>  
b = %<b>
```

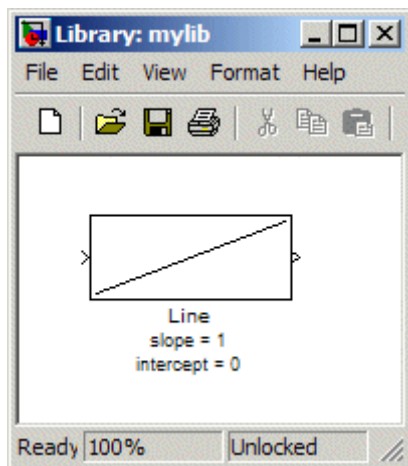
The library appears as follows:



- 5** Save mylib.
- 6** Create a model named mymdl.
- 7** Create an instance of the Line block in mymdl.
- 8** Rename the instance LineA.
- 9** Change the value of LineA's m parameter to -1.5 .
- 10** Change the value of LineA's b parameter to 3 .
- 11** Set LineA's annotation property to display the values of its m and b parameters.

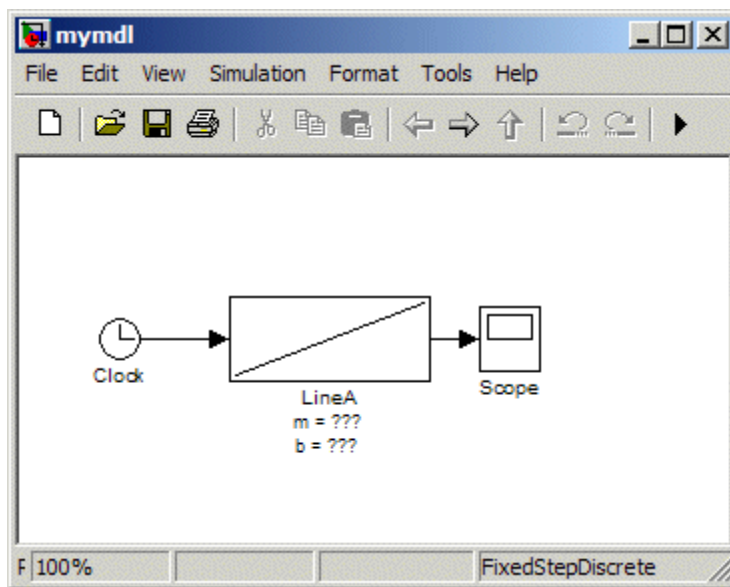


- 12** Configure mymdl to use a fixed-step, discrete solver with a step size of 0.1 second.
- 13** Save mymdl.
- 14** Simulate mymdl.
Note that the model simulates without error.
- 15** Close mymdl.
- 16** Unlock mylib.
- 17** Rename the m parameter of the Line block in mylib to slope.
- 18** Rename Line's b parameter to intercept.
- 19** Change Line's mask icon and annotation properties to reflect the parameter name changes.



20 Save mylib.

21 Reopen mymdl.



Note that LineA's 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.

22 Close mymdl.

23 Unlock mylib.

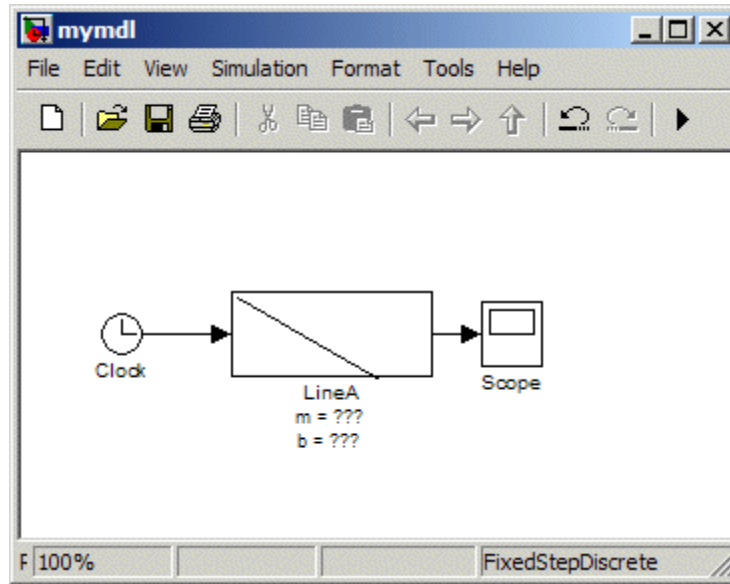
24 Select the Line block in mylib.

25 Execute the following command at the MATLAB command line.

```
set_param(gcb, 'MaskVarAliases', {'m', 'b'})
```

This specifies that m and b are aliases for the Line block's slope and intercept parameters.

26 Reopen mymdl.



Note that LineA's appearance 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 LineA's slope parameter.

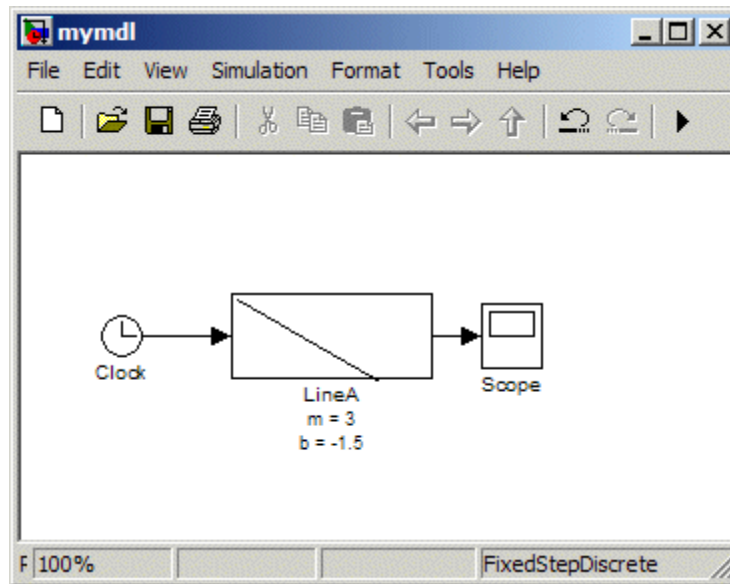
- 27** Change LineA's block annotation property to reflect LineA's parameter name changes, i.e., replace

```
m = %<m>
b = %<b>
```

with

```
m = %<slope>
b = %<intercept>
```

LineA now appears as follows.



Note that LineA's annotation shows that, thanks to parameter aliasing, Simulink has correctly applied the parameter values stored for LineA in mymdl's model file to the block's renamed parameters.

Adding Libraries to the Library Browser

In this section...

“How to Display a Library in the Library Browser” on page 8-24

“Example of a Minimal `sblocks.m` File” on page 8-24

“Adding More Descriptive Information in `sblocks.m`” on page 8-25

How to Display a Library in the Library Browser

- 1 Create a directory 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 directory on the MATLAB path. Two top-level libraries cannot exist in the same directory.

- 2 Create or copy the top-level library and its sublibraries into the directory you created in the MATLAB path.
- 3 In the directory for the top-level library, include a `sblocks.m` file.

The approach you use to create the `sblocks.m` file depends on your requirements for describing the library:

- If a minimal `sblocks.m` file meets your needs, then create a new `sblocks.m` file, based on the example below
- If you want to describe the library more fully, consider copying an existing `sblocks.m` file to use as a template, editing the copy to describe your library (see below).

Example of a Minimal `sblocks.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 `sblock.m` file.

```
% sblocks.m defines a block library
function blkStruct = sblocks
```

```
% 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;
% End of sblocks.m
```

Adding 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`.

Working with Block Masks

- “What Are Masks?” on page 9-2
- “Why Use a Mask?” on page 9-3
- “Masked Subsystem Example” on page 9-5
- “Roadmap for Masking Blocks” on page 9-8
- “Mask Terminology” on page 9-10
- “Creating a Block Mask” on page 9-11
- “Masking a Model Block” on page 9-28
- “Masks on Blocks in User Libraries” on page 9-29
- “Operating on Existing Masks” on page 9-31
- “Roadmap for Dynamic Masks” on page 9-34
- “Calculating Values Used Under the Mask” on page 9-35
- “Controlling Masks Programmatically” on page 9-39
- “Understanding Mask Code Execution” on page 9-44
- “Creating Dynamic Mask Dialog Boxes” on page 9-51
- “Creating Dynamic Masked Subsystems” on page 9-55
- “Best Practices for Using M-Code in Masks” on page 9-59

What Are Masks?

A mask is a custom user interface for a Simulink block. The mask hides the native user interface of the underlying block, substituting an icon and a parameters dialog box defined by the mask. You can apply a mask to any Subsystem block, Model block, or S-Function block. The block can optionally reside in a user-defined library.

Masking a block changes only the block's user interface, not its underlying characteristics. For example, masking a nonatomic subsystem does not make it act as an atomic subsystem, and masking a virtual block does not convert it to a nonvirtual block. You cannot save a mask separately from the block that it masks, or create a freestanding mask definition and apply it to more than one block.

A mask's icon and parameters dialog box can provide any capability that a block's native icon and dialog box can provide. When you set mask parameter values, the mask can use the values to dynamically change the mask's icon and dialog box, and to calculate values to be used under the mask. A mask on a subsystem can dynamically change the subsystem to reflect mask parameter values.

Why Use a Mask?

Masks can provide interface customization, logic encapsulation, and data hiding. For example, although defining a subsystem simplifies a model's graphical appearance, the simplification does not provide a user interface that is specific to the subsystem. The lack of a subsystem-specific interface has several possible disadvantages, including:

- The subsystem's icon is generic: it does not indicate the meaning of the subsystem, and cannot change to provide information about parameter settings in the subsystem.
- The subsystem parameters that need to be changed to control behavior may be distributed among many subsystem blocks, making the parameters hard to find.
- The parameter names in the blocks comprising the subsystem reflect only the generic purpose of each block, rather than the parameter's purpose in the context of the model.
- Making any change to the subsystem requires exposing its complete contents in an editor, which makes the subsystem vulnerable to unintended changes.
- Because the subsystem has no separate parameters dialog box, no Block Help information is available for the subsystem, as it would be for a built-in block.

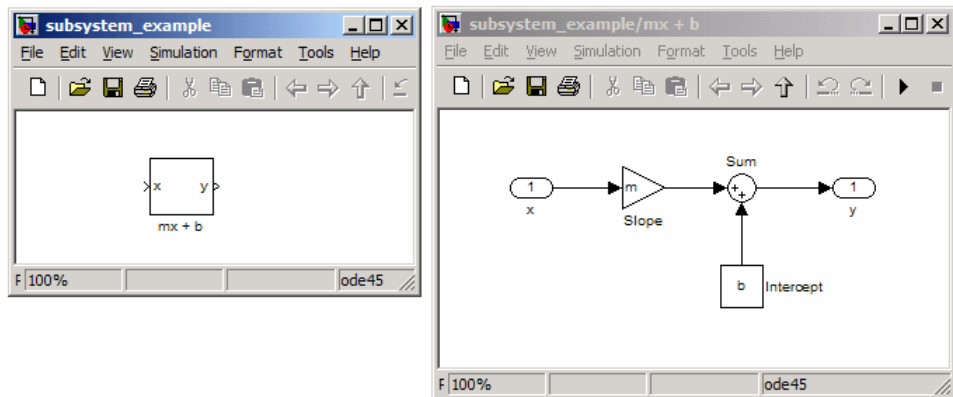
Masking a subsystem can address all of these problems. A subsystem mask can:

- Display a meaningful icon that updates dynamically and reflects values within the subsystem.
- Define customized user-settable parameters whose names reflect the purpose of the subsystem.
- Provide a parameters dialog box that gives access only to parameters that should be exposed.
- Shows customized Block Help and Online Help information that is specific to the subsystem.

The mask thereby encapsulates the subsystem under an interface that reflects the subsystem as the user sees it, as distinct from the subsystem's architecture, and provides the look and feel of a built-in Simulink block. Similar advantages apply to masks on Model blocks and S-Function blocks.

Masked Subsystem Example

The next figure shows a Subsystem block that models the equation for a line, $y = mx + b$. The techniques for creating such a subsystem appear in “Creating Subsystems” on page 4-37. The subsystem is not atomic, so the Subsystem block is just a graphical abbreviation for the underlying blocks:



Double-clicking an unmasked Subsystem block opens a separate window that displays the contents of the subsystem, as shown in the above figure. This subsystem contains a Gain block, named **Slope**, whose **Gain** parameter is specified as m , and a Constant block, named **Intercept**, whose **Constant value** parameter is specified as b . These parameters represent the slope and intercept of a line. To examine this model, click `subsystem_example` or execute:

```
run([docroot ' /toolbox/simulink/ug/examples/masking/subsystem_example.mdl'])
```

The organization of this model has several disadvantages:

- Although the block gives some indication of its purpose by showing the calculation that it performs, the icon adds no specific information.
- Only the values of the **Gain** and **Constant value** parameters are relevant outside the subsystem, but editing the subsystem displays its entire contents.

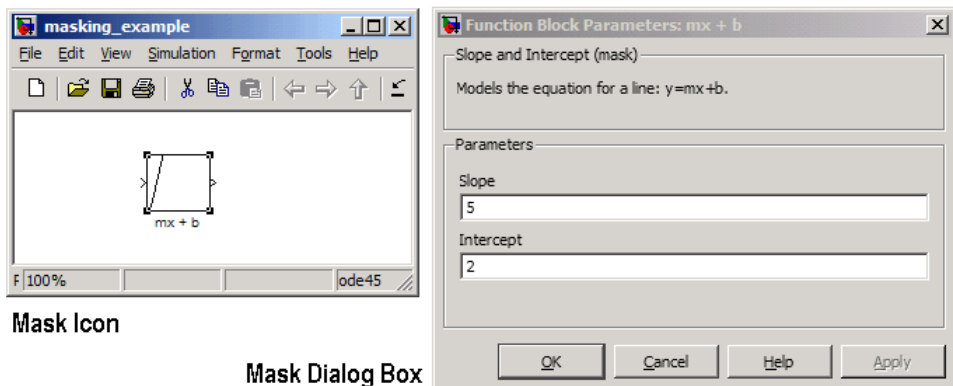
- The names of the block parameters within the subsystem relate to Simulink architecture rather than to the purpose of the subsystem.

These problems can be overcome by applying a mask to the Subsystem block. This mask can:

- Provide an icon that reflects the subsystem's purpose
- Show only the parameters that are used outside the subsystem
- Give those parameters names that relate to the subsystem's purpose

The next figure shows the above model with a mask applied to the Subsystem block. To examine this model, click `masking_example` or execute:

```
run([docroot ' /toolbox/simulink/ug/examples/masking/masking_example.mdl'])
```



Mask Icon

Mask Dialog Box

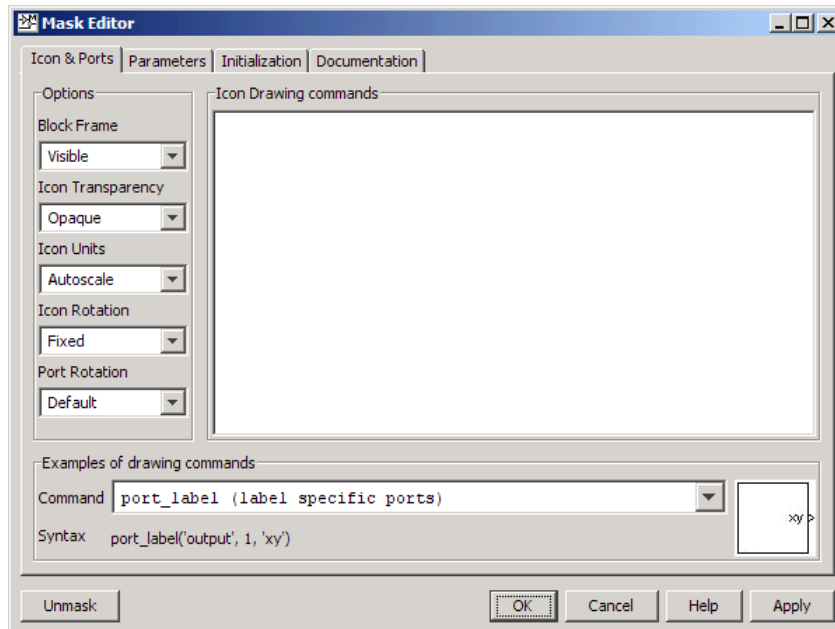
Masking the subsystem has created a self-contained functional unit. A customized mask icon replaces the default Subsystem block icon. Double-clicking the mask icon opens a customized Mask Parameters dialog box, which replaces the Subsystem block dialog box. The Mask Parameters dialog box shows two parameters: **Slope**, which provides the value of m in the subsystem Gain block, and **Intercept**, which provides the value of b in the subsystem Constant block.

These parameters are the only constructs within the subsystem that are of interest outside the subsystem, so the mask has been designed to hide everything else. The mask parameters are currently set to 5 and 2,

respectively. These values are not part of the mask itself, but were entered just as if they were the values of ordinary block parameters. The parameter values are available to all the blocks in the underlying subsystem. The mask icon displays a line that reflects the current value of the **Slope** parameter.

Roadmap for Masking Blocks

The “Simulink Mask Editor” provides the capabilities necessary for creating, changing, and removing a Subsystem, Model, or S-Function block mask. When first opened, the Mask Editor looks like this:



The Mask Editor provides four tabbed panes, which you can use to perform any masking operation:

- “Icon & Ports Pane” — Specifies commands that draw and control the icon that represents the masked block.
- “Parameters Pane” — Specifies mask parameters and code that executes when a parameter value changes.
- “Initialization Pane” — Specifies code that executes as needed to initialize the mask and underlying block.
- “Documentation Pane” — Specifies documentation that describes the block in its dialog box and in Online Help.

Use the preceding links as needed to access the Mask Editor reference. Definitions and instructions for masking a block appear in:

- “Mask Terminology” on page 9-10
- “Creating a Block Mask” on page 9-11
- “Masking a Model Block” on page 9-28

If the block to be masked resides in a user-defined library, you will also need the information in:

- “Masks on Blocks in User Libraries” on page 9-29

After you have defined a block mask, you can perform various operations on it, as described in:

- “Viewing Mask Parameters” on page 9-31
- “Looking Under a Block Mask” on page 9-31
- “Changing a Block Mask” on page 9-31
- “Removing and Caching a Mask” on page 9-32
- “Restoring a Cached Mask” on page 9-33
- “Permanently Deleting a Mask” on page 9-33

If you want to define a mask that can use mask parameter values to dynamically change the mask’s icon and dialog box, or calculate values to be used under the mask, first read the sections listed above as needed, then read:

- “Roadmap for Dynamic Masks” on page 9-34

All dynamic masking techniques require M-code programming, which is described in *MATLAB Programming Fundamentals*.

Mask Terminology

The rest of this chapter assumes that you know the following definitions:

Mask Icon — An icon that replaces a masked block's standard icon when the masked block appears in a model. You can accept a default icon or provide drawing code that defines the icon.

Mask Parameters — Optional parameters that link to block parameters that exist under the mask. Setting a mask parameter sets the associated block parameter.

Mask Initialization Code — Optional MATLAB code that runs when needed to initialize the masked subsystem. You can use initialization code to modify a masked subsystem to reflect current parameter values.

Mask Callback Code — Optional MATLAB code that runs whenever a mask parameter value changes. You can use callback code to modify a mask dialog box to reflect current parameter values.

Mask Documentation — Optional information that names and briefly describes the masked block, and provides Help information about how to use it.

Mask Dialog Box — A dialog box that replaces the masked block's standard parameters dialog box. The mask dialog box displays a block description, contains controls that enable a user to set mask parameter values, and provides Help information.

Mask Workspace — If the mask uses mask parameters or initialization code, it has a mask workspace that holds parameter and temporary values used by the mask.

Creating a Block Mask

In this section...

- “Introduction” on page 9-11
- “Opening the Mask Editor” on page 9-12
- “Defining a Mask Icon” on page 9-13
- “Understanding Mask Parameters” on page 9-16
- “Defining Mask Parameters” on page 9-18
- “Defining Mask Documentation” on page 9-21
- “Using a Block Mask” on page 9-25

Introduction

This section describes techniques for masking any type of block. The techniques are the same whether you mask a Subsystem block, Model block, or S-Function block. The masked block can optionally be included in a user-defined library, as described in “Masks on Blocks in User Libraries” on page 9-29. When you mask a Model block, one special consideration applies, as described in “Masking a Model Block” on page 9-28. Masking an S-Function block requires no special provisions.

In practice, most masks are subsystem masks, because Model blocks and S-Function blocks already provide a high degree of encapsulation. For simplicity, the instructions and examples in this section refer specifically to a subsystem, and the mask does not dynamically change its dialog box or the subsystem under the mask. Instructions for masks that make dynamic changes begin in “Roadmap for Dynamic Masks” on page 9-34, which assumes that you understand everything in this section.

This section demonstrates only some of the capabilities of the Mask Editor. See “Simulink Mask Editor” for complete reference information. The section assumes that you know the definitions in “Mask Terminology” on page 9-10 and are familiar with the “Masked Subsystem Example” on page 9-5.

This section illustrates the described operations by constructing the mask described in “Masked Subsystem Example” on page 9-5. You can read the

section and apply the operations to your own model, or you can use the section as a tutorial, as follows:

- 1 Open the unmasked source model by clicking `subsystem_example` or executing:

```
run([docroot ' /toolbox/simulink/ug/examples/masking/subsystem_example.mdl'])
```

- 2 Save the model in a writable working directory.

- 3 Optionally open the completed example by clicking `masking_example` or executing:

```
run([docroot ' /toolbox/simulink/ug/examples/masking/masking_example.mdl'])
```

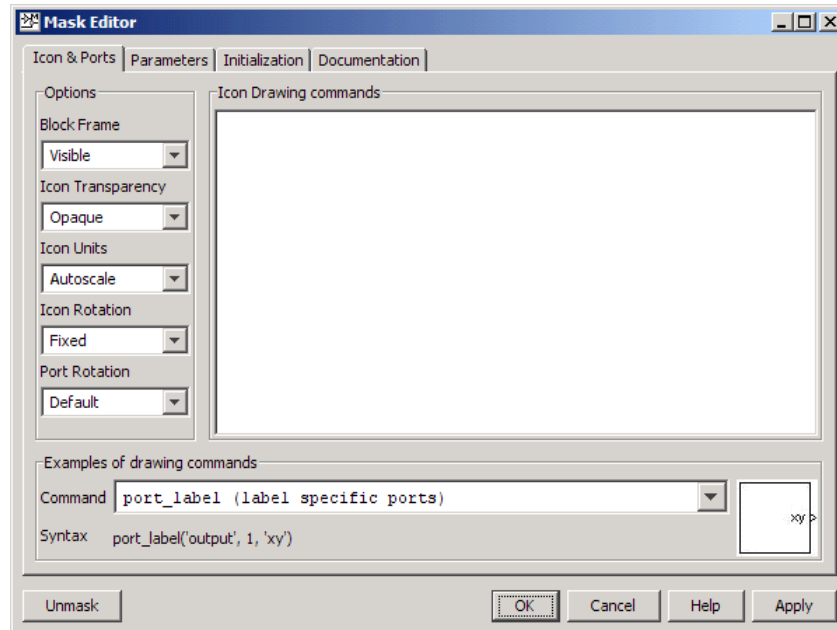
- 4 Execute the instructions in this section on the `subsystem_example` to implement the capabilities shown in the `masking_example`.

Opening the Mask Editor

The Simulink Mask Editor provides all capabilities necessary for masking a block. To invoke the editor on an unmasked block:

- Select the block.
- Choose **Mask *BlockType*** from the **Edit** menu or the block's context menu.

The Mask Editor opens:



You can define mask components in any order, as long as the final result is correct. You can also open the Mask Editor on block that is already masked, as described in “Changing a Block Mask” on page 9-31. Click **Apply** to make changes within the model, or **OK** to make changes within the model and close the editor. To permanently save changes, you must save the model itself. If you close the model without saving it, all unsaved Mask Editor changes are lost.

Defining a Mask Icon

A mask icon replaces a masked block’s standard icon when the masked block appears in a model. Mask icons can contain descriptive text, graphics, images, state equations, one or more plots, or a transfer function. Simulink uses code that you supply to draw the mask icon.

Use the Mask Editor’s **Icon & Ports** pane to specify the icon for a masked block. This pane is always selected by default when you open the editor. The previous figure shows its initial appearance. See the “Icon & Ports Pane”

reference for information about all **Icon & Ports** pane capabilities. This section gives an example of their use.

To specify the contents of a mask icon, enter one or more of the commands described in “Mask Icon Drawing Commands” in the **Drawing Commands** pane. You must use only these commands to draw a mask icon. An error occurs if a command not listed in “Mask Icon Drawing Commands” appears in the **Drawing Commands** pane. The drawing commands can access all variables defined in the mask workspace. To see examples of drawing command syntax, use the pulldown in the **Examples of drawing commands** pane.

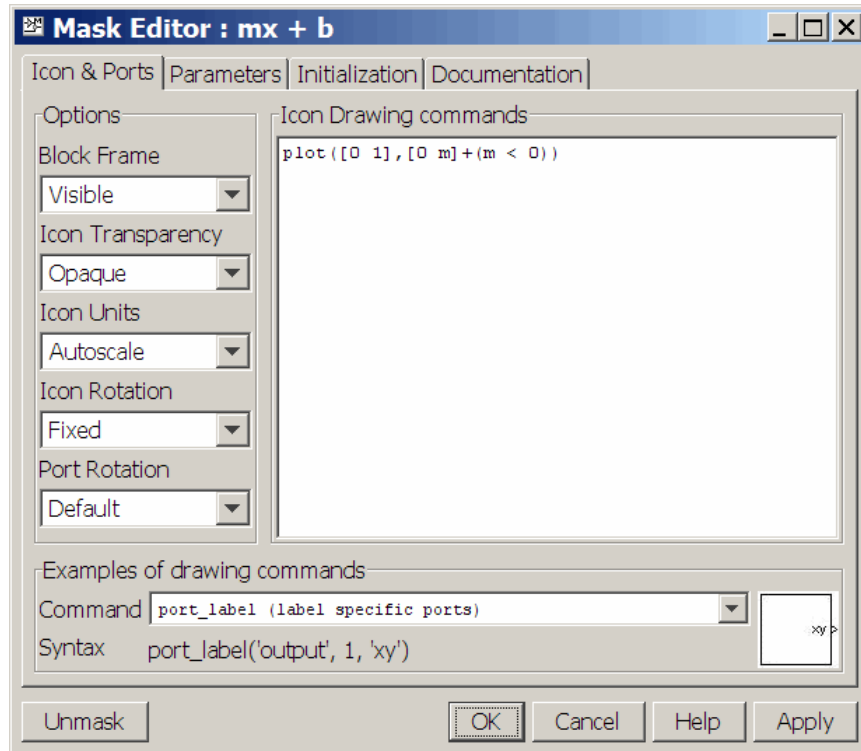
The mask icon in `masking_example` displays a line that reflects the slope of the line modeled the masked subsystem. To define this icon:

- 1 Type the following command into the **Drawing Commands** pane:

```
plot([0 1],[0 m]+(m<0))
```

- 2 Under **Icon Options**, set **Units** to Normalized.

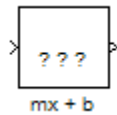
The **Icon** pane now looks like this:



3 Click **Apply**.

This `plot` command plots a line from $(0,0)$ to $(1,m)$. If the slope of the line is negative, the line is shifted up by 1 to keep it within the visible drawing area of the block. Setting **Icon Units** to **Normalized** tells Simulink to draw the icon in a frame whose bottom-left corner is $(0,0)$ and whose top-right corner is $(1,1)$.

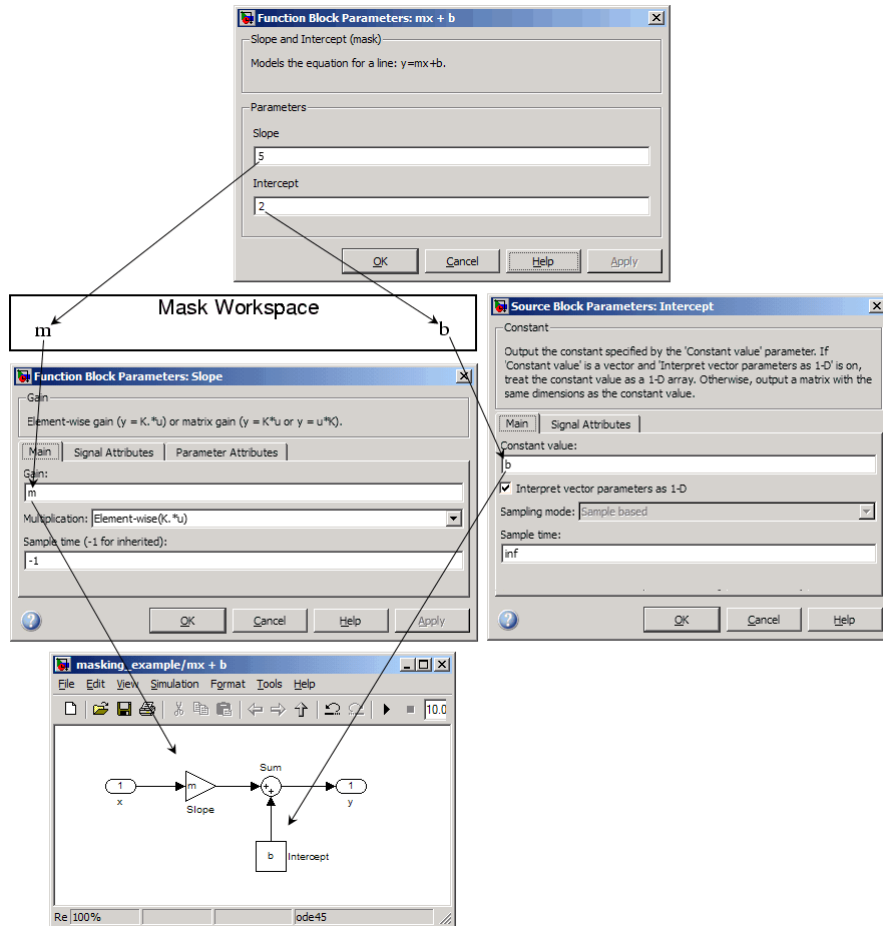
If Simulink cannot evaluate all commands in the **Drawing Commands** pane to obtain a displayable result, the content of the icon is three question marks. In `masking_example`, three question marks appear at this point because the variable `m` has not yet been defined:



The pull-downs under **Icon Options** allow you to specify the icon frame visibility, icon transparency, drawing context, icon rotation, and port rotation. See the “Icon & Ports Pane” reference for information about these and all other **Icon & Ports** pane capabilities.

Understanding Mask Parameters

Mask parameters provide the connection between the Mask Parameters dialog box and the underlying block. Each mask parameter appears in the Mask Parameters dialog box as a control: an edit box, a pop-up, or a checkbox. Setting the value of this control sets the value of an associated mask parameter that is defined in the mask workspace. The underlying block specifies that parameter as the value of one or more block parameters, and thus accesses the value of the corresponding control in the Mask Parameters dialog box. The next figure illustrates this relationship for the `masking_example`:



In this figure, the Mask Parameters dialog box (top) contains edit-box controls for two parameters: **Slope**, which is associated with the mask workspace variable m , and **Intercept**, which is associated with the mask workspace variable b . In the subsystem (bottom) the Gain block named Slope specifies its **Gain** parameter as m , and the Constant block named Intercept specifies its **Constant value** parameter as b , as shown in the two Block Parameters dialogs (middle).

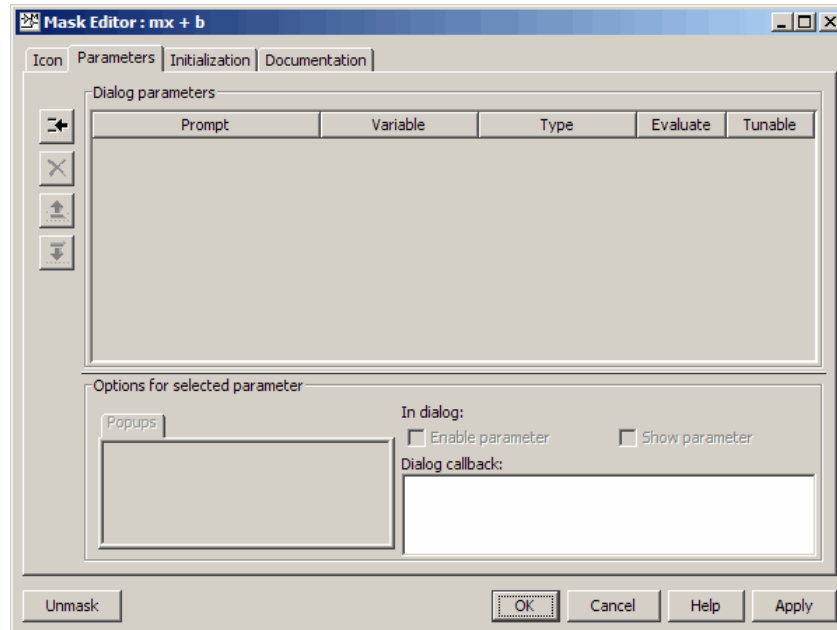
When you set the values of **Slope** and **Intercept** in the Mask Parameters dialog box, Simulink automatically assigns the mask parameter values to the corresponding mask workspace variables, **m** and **b**. Thus in the **Mask Workspace** above, **m = Slope = 5**, and **b = Intercept = 2**.

Before simulation begins, Simulink tries to resolve the Gain block's **Gain** parameter **m** and the Constant block's **Constant Value** parameter **b** to numeric values by searching up the workspace hierarchy, as described in "Resolving Symbols" on page 4-75. Because a mask exists, Simulink searches the mask workspace first. There it finds **m** and **b** defined, so it successfully resolves the two block parameters: **Gain = m = Slope = 5**, and **Constant value = b = Intercept = 2**.

If no mask existed, and hence no mask workspace, but the subsystem itself was unchanged, Simulink would perform the same hierarchical search seeking a value for **m** and **b**, and the search for each value would have to succeed before simulation could begin. Thus the existence of a mask affects hierarchical search only by providing a mask workspace that Simulink searches first.

Defining Mask Parameters

Be sure you have read "Understanding Mask Parameters" on page 9-16 before reading this section. Use the Mask Editor **Parameters** pane to specify block mask parameters:



See the “Parameters Pane” reference for information about all **Parameters** pane capabilities. This section gives an example of their use.

The parameters pane defines both the individual mask parameters and the organization of the Mask parameters dialog box. For each mask parameter, a row exists in the **Parameters** pane **Dialog parameters** table. The fields in this row are:

- **Prompt** — A prompt that represents the parameter in the Mask Parameters dialog box
- **Variable** — A variable in the mask workspace that corresponds to the mask parameter
- **Type** — The type of control that represents the mask parameter in the mask dialog box.
- **Evaluate** — Whether the value of the mask parameter is used literally or evaluated

- **Tunable** — Whether the parameter is editable (tunable) while simulation is running
- **Tab** — The name of the tab, if any, on which the control appears in the mask dialog box

Parameters appear in the Mask Parameters dialog box in the same order that the corresponding parameter definitions appear in the **Dialog parameters** table. To define the mask parameters used in the `masking_example`:



- 1 Click the **Add** button at the upper left of the **Parameters** pane to create a new parameter:

Prompt	Variable	Type	Evaluate	Tunable
		edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

- 2 Edit the **Prompt** field to specify **Slope** and the **Variable** field to specify **m**:

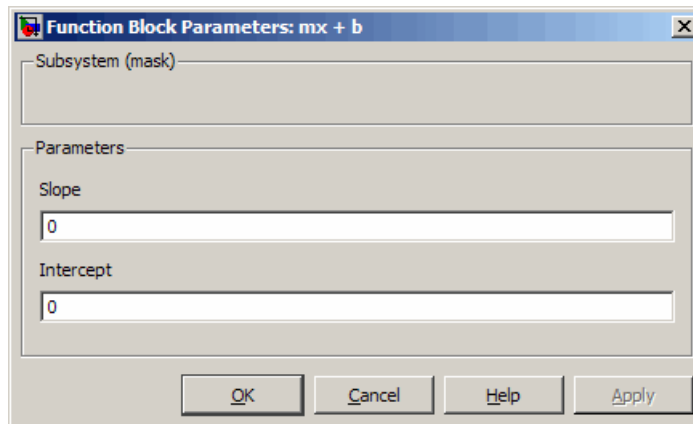
Prompt	Variable	Type	Evaluate	Tunable
Slope	m	edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

- 3 Click **Add** to create a second parameter, and specify its **Prompt** as **Intercept** and its **Variable** as **b**.

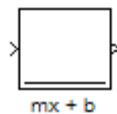
Prompt	Variable	Type	Evaluate	Tunable
Slope	m	edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Intercept	b	edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Prompt	Variable	Type	Evaluate	Tunable
Slope	m	edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Intercept	b	edit	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Double-click the masked subsystem to see the current Mask Parameters dialog box:



The variables `m` and `b` now exist in the mask workspace. The GUI parameters in the Mask Parameters dialog box are mapped to the corresponding workspace variables. By default, these variables are of type `double` and have a default value of `0`. The mask icon now shows an initial **Slope** of `0`:



See the “Parameters Pane” reference for information about all **Parameters** pane capabilities.

Defining Mask Documentation

A block mask can display three types of documentation. They are all optional, but in practice essentially all masks provide them.

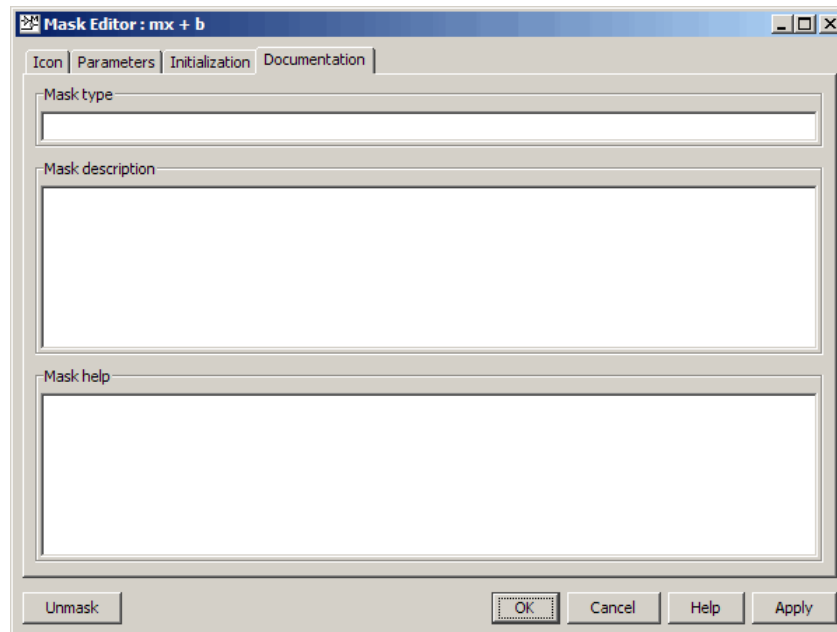
Mask type — A name that appears at the top of the masked block's Mask Parameters dialog box. When Simulink displays a masked block's Mask Parameters dialog box, it suffixes (mask) to the mask type. No newlines can appear in the text.

Mask description — Summary text that describe the block's purpose or function. The description appears in the masked block's dialog box under the mask type. Newlines and multiple blanks can appear for formatting.

Mask help — Provides additional information that appears when the user clicks the **Help** button on the mask dialog box. The text can be:

- Plain text. Use newlines and multiple blanks for formatting.
- HTML text and graphics. Any standard HTML tag can appear.
- A URL that points to information, which can be text or HTML.
- A web or eval command that returns text or HTML to display.

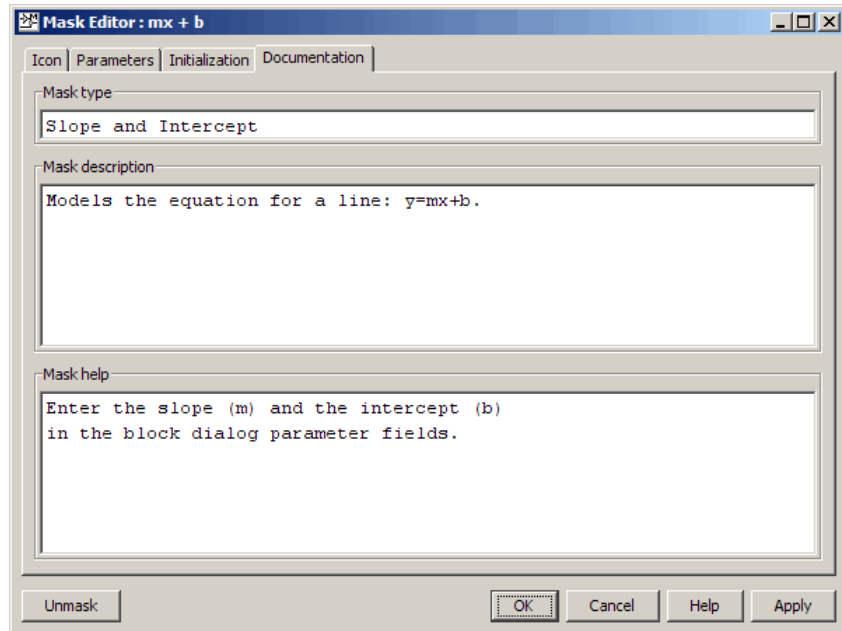
Use the Mask Editor's **Documentation** pane to specify the **Mask type**, **Mask description**, and **Mask Help** for a masked block:



See the “Documentation Pane” reference for information about all **Documentation** pane capabilities. This section gives an example of their use.

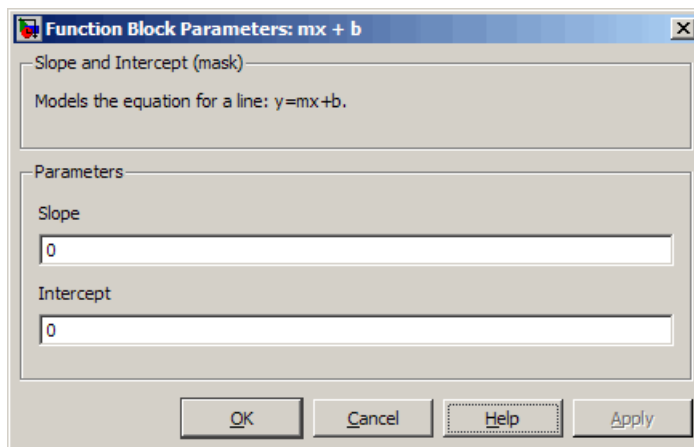
To specify mask documentation, enter text in the three fields of the **Documentation** pane. To define the `masking_example` documentation:

- 1 Enter the text shown below in the **Mask type**, **Mask description**, and **Mask help** fields:

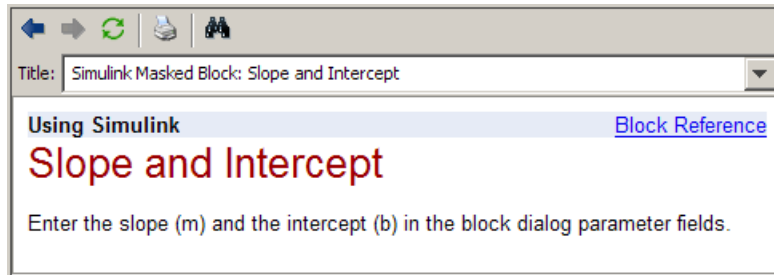


2 Click Apply.

The Mask Parameters dialog box closes if it is open. Double-click the masked subsystem to open the dialog box:



The **Mask type** and **Mask description** now show the specified text. Simulink suffixes (**mask**) to the Mask type to indicate that the dialog box shows a mask rather than a native block dialog. To see the Mask Help, click the **Help** button. The Help text appears in the Online Help browser:



The full capabilities of MATLAB Online Help are available after you access the browser by clicking **Help** in a Mask Parameters dialog box.

See the “Documentation Pane” reference for information about all **Documentation** pane capabilities.

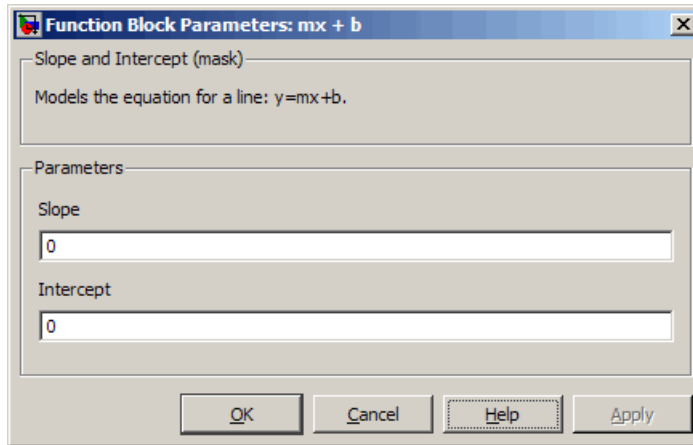
Using a Block Mask

Once you have completed the operations described above, click **OK** to apply any final changes to the mask and close the Mask Editor. If you want to save the mask for later examination, save the model. If you close a model without saving it, all unsaved Mask Editor changes are lost.

You can now use the masked Subsystem block $mx + b$ exactly as you could if it were an instance of a built-in block whose capabilities are those of the subsystem and whose interface is that of the mask:

- 1 Double-click the masked block $mx + b$.

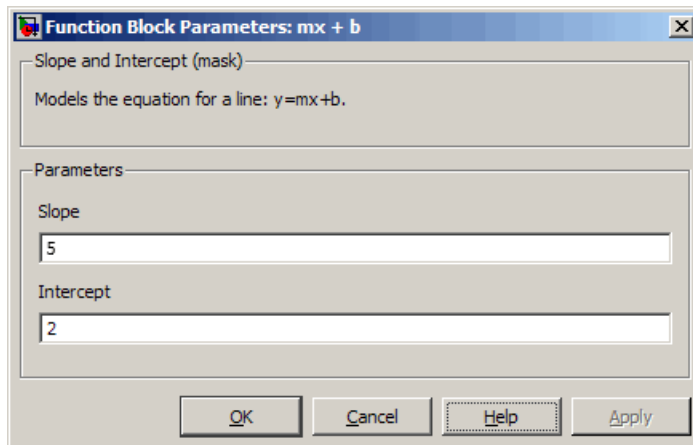
The Mask Parameters dialog box opens:



2 Change the parameter values so that **Slope** = 5 and **Intercept** = 2.

3 Click **Apply**.

The Mask Parameters dialog box and the mask icon now look like this:



When you click **Apply**, Simulink assigns the value of **Slope** to the mask workspace parameter **m**, and the value of **Intercept** to the mask workspace parameter **b**, as described in “Understanding Mask Parameters” on page 9-16. The blocks in the subsystem that reference **m** and **b** will use those values

during simulation. The mask icon now reflects **Slope = 5** via the drawing command `plot([0 1],[0 m]+(m<0))`, as described in “Defining a Mask Icon” on page 9-13. The mask documentation is as defined in “Defining Mask Documentation” on page 9-21.

Masking a Model Block

You can use all techniques described in “Roadmap for Masking Blocks” on page 9-8 to mask a model block, with one special requirement. If a mask specifies the name of the model referenced by a Model block, or by any Model block in a masked subsystem, the name of the referenced model must be given literally, rather than obtained by evaluating a workspace variable. This requirement exists because Simulink updates model reference targets before evaluating block parameters. Two ways to enforce the requirement are:

- Use a Pop-Up control to specify the name of the referenced model. Clear the control’s **Evaluate** parameter to cause the control to provide a textual rather than a numeric value. This technique restricts the user to specifying one of a predefined set of models.
- Use an Edit control to specify the name of the referenced model. Clear the control’s **Evaluate** parameter to cause the name that the user provides to be interpreted literally. This technique allows the user to specify any model as the referenced model.

See “Defining Mask Parameters” on page 9-18 for information about defining Pop-Up and Edit controls.

Note The mask workspace of a Model block is not visible to the model that it references. Any variables used by the referenced model must resolve to workspaces defined in the referenced model, or to the MATLAB base workspace.

Masks on Blocks in User Libraries

In this section...

“About Masks and User-Defined Libraries” on page 9-29

“Masking a Block for Inclusion in a User Library” on page 9-29

“Masking a Block that Resides in a User Library” on page 9-29

“Masking a Block Copied from a User Library” on page 9-30

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 “Creating Block Libraries” on page 8-14.

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:

- 1 Open the library that contains the block.

2 Choose **Edit > Unlock Library**.

You can now 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 Parameters 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. Saving the library will automatically relock it. Complete information about user libraries, appears in Chapter 8, "Working with Block Libraries". Additional information relating to masked library blocks appears in "Creating Block Libraries" on page 8-14.

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.

Operating on Existing Masks

In this section...

- “Changing a Block Mask” on page 9-31
- “Viewing Mask Parameters” on page 9-31
- “Looking Under a Block Mask” on page 9-31
- “Removing and Caching a Mask” on page 9-32
- “Restoring a Cached Mask” on page 9-33
- “Permanently Deleting a Mask” on page 9-33

Changing 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 Choose **Edit Mask** from the **Edit** menu or the block’s context menu.

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.

Viewing Mask Parameters

To display a masked block’s Mask Parameters dialog box, either double-click the block or select it and choose **Mask Parameters** from the **Edit** menu or the block’s context menu.

To display the Block Parameters dialog box that double-clicking would display if no mask existed, select the masked block and choose **BlockType Parameters** from the **Edit** menu or the block’s context menu.

Looking Under a Block Mask

To see the block diagram under a masked Subsystem block, or the model referenced by a masked Model block, select the block and choose **Look Under**

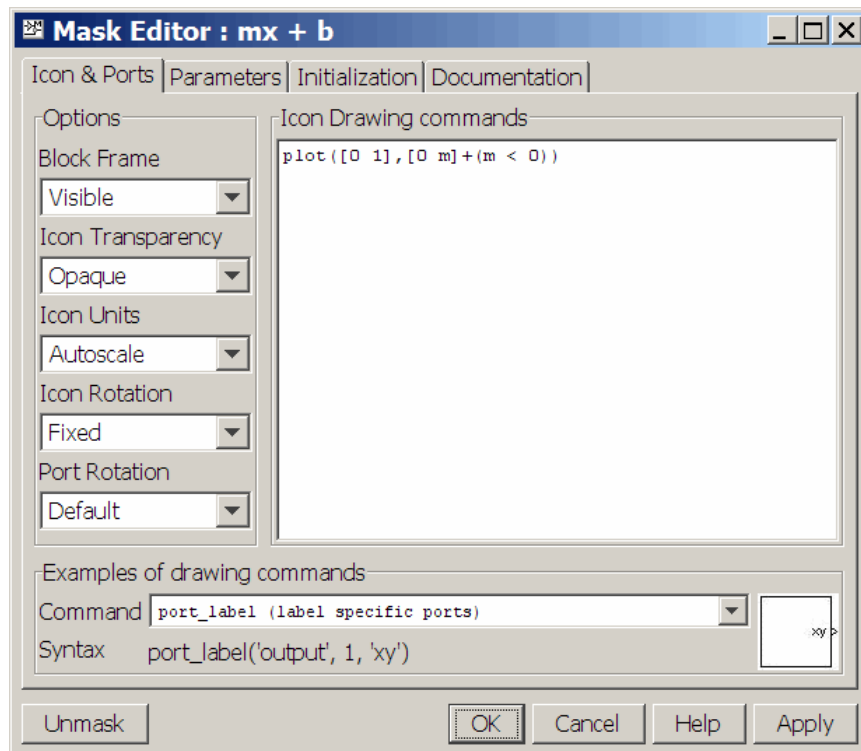
Mask from the **Edit** menu or the block's context menu. The block diagram or referenced model opens in a separate window.

Removing and Caching a Mask

To remove a mask from a block and cache it for possible restoration later:

- 1 Select the block.
- 2 Choose **Edit Mask** from the **Edit** menu or the block's context menu.

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 “Restoring a Cached Mask” on page 9-33, or delete it, as described in “Permanently Deleting a Mask” on page 9-33. The removed cached mask has no further effect unless you restore it.

Restoring a Cached Mask

As long as a model remains open, you can restore a mask that you removed as described in “Removing and Caching a Mask” on page 9-32.

- 1 Select the block.
- 2 Choose **Mask *BlockType*** from the **Edit** menu or the block’s context menu.

The Mask Editor reopens, showing the cached masked definition.

- 3 Modify the definition if needed, using the techniques in “Roadmap for Masking Blocks” on page 9-8.
- 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 Deleting a Mask

To delete a mask permanently, first remove it as described in “Removing and Caching a Mask” on page 9-32, 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.

Roadmap for Dynamic Masks

You can use the Mask Editor to perform all of the mask operations referenced in “Roadmap for Masking Blocks” on page 9-8 without writing M-code. You can also use M-code to perform the following dynamic mask operations:

- Calculate values assigned under the mask rather than using the values that the user specified
- Change a Mask Parameters dialog box to reflect parameter values specified by the user
- Change the architecture of a masked subsystem to reflect parameter values set by the user

In order to perform any of these operations, you will need to understand all topics listed in “Roadmap for Masking Blocks” on page 9-8 that relate to your situation, and in particular the `masking_example` shown in “Masked Subsystem Example” on page 9-5 and constructed in “Creating a Block Mask” on page 9-11. You can then read the following sections as needed:

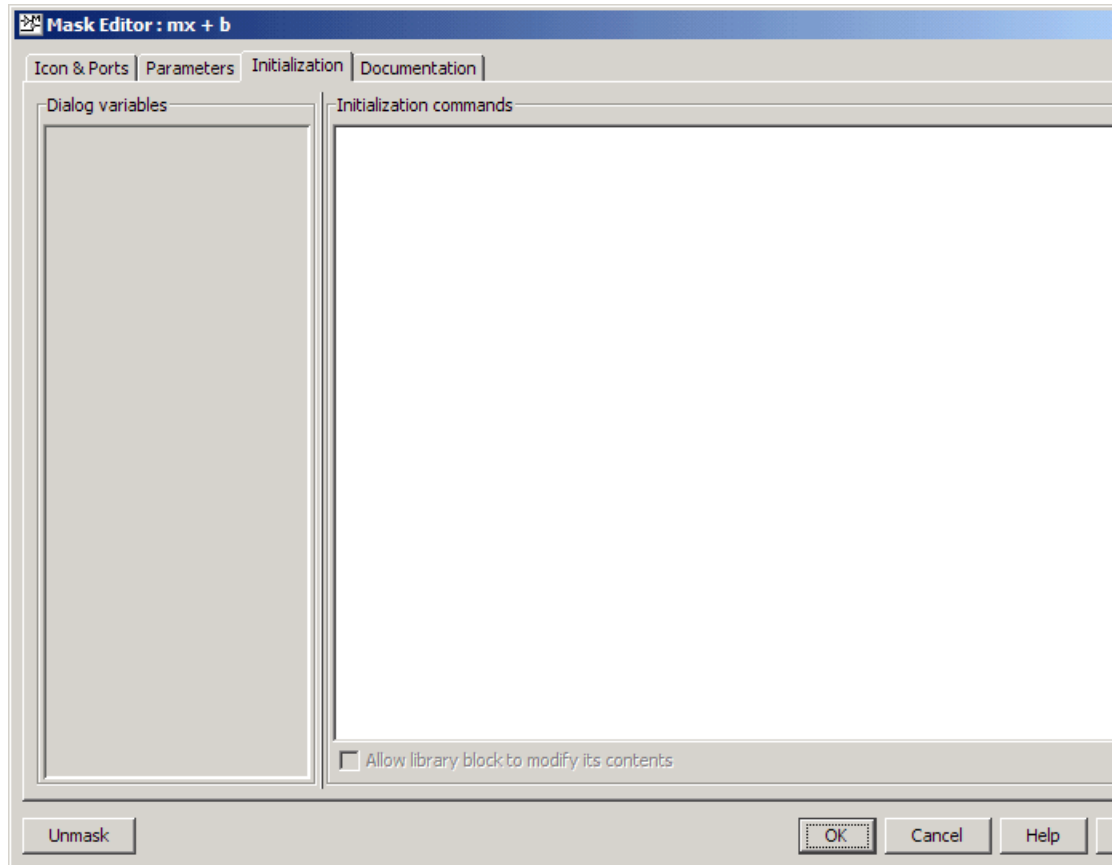
- “Calculating Values Used Under the Mask” on page 9-35
- “Creating Dynamic Mask Dialog Boxes” on page 9-51
- “Creating Dynamic Masked Subsystems” on page 9-55

These operations require using the “Parameters Pane” and the “Initialization Pane” to specify M-code. For information about M-code programming, see *MATLAB Programming Fundamentals*. For a description of efficient ways to organize M-code used in a mask, see “Best Practices for Using M-Code in Masks” on page 9-59.

Calculating Values Used Under the Mask

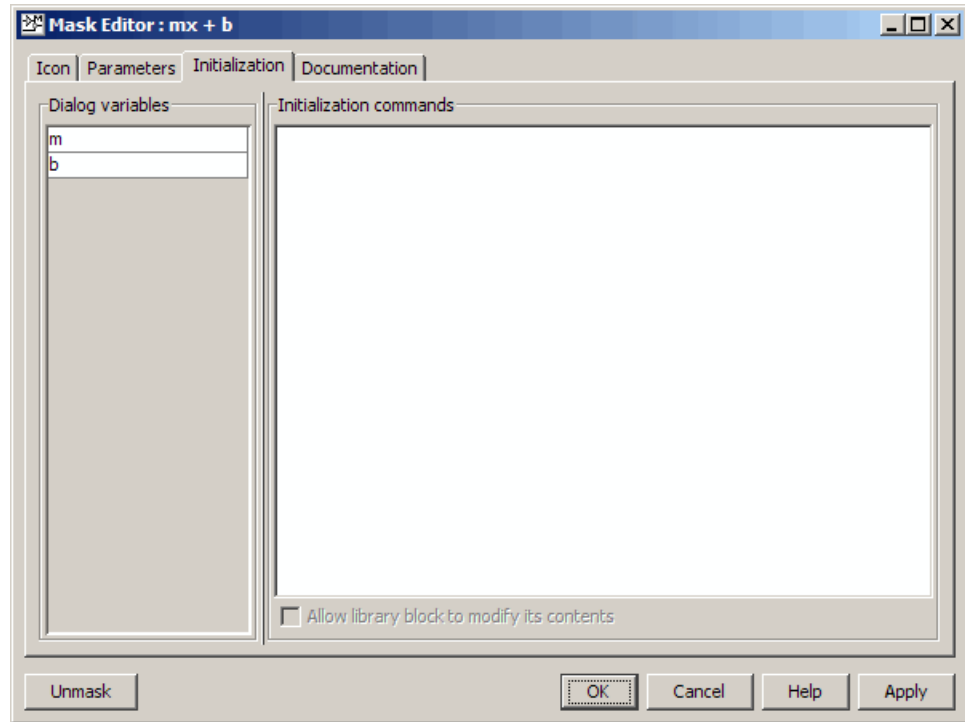
The `masking_example` assigns the values input using the Mask Parameters dialog box directly to block parameters underneath the mask, as described in “Understanding Mask Parameters” on page 9-16. The assignment occurs because the block parameter and the mask parameter have the same name, so the search that always occurs when a block parameter needs a value finds the mask parameter value automatically, as described in “Resolving Symbols” on page 4-75.

You can use the Mask Editor to insert any desired calculation between a value in the Mask Parameters 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.

In order to calculate a value for a block parameter, you must 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 for the `masking_example` 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 a masked block's initialization code 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 mask's initialization code:

```
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.

Controlling Masks Programmatically

In this section...

“Predefined Masked Dialog Parameters” on page 9-40

“Notes on Mask Parameter Storage” on page 9-43

The Simulink software defines a set of masked block parameters that define the current state of the masked block’s dialog. You can use the Mask Editor to inspect and set many of these parameters. The Simulink `get_param` and `set_param` commands also let you inspect and set mask dialog parameters. The `set_param` command allows you to set parameters and hence change a dialog’s appearance while the dialog is open. This in turn allows you to create dynamic masked dialogs.

For example, you can use the `set_param` command in mask callback functions to be invoked when a user changes the values of user-defined parameters. The callback functions in turn can use `set_param` commands to change the values of the masked dialog’s predefined parameters and hence its state, for example, to hide, show, enable, or disable a user-defined parameter control.

Use the 'mask' option of the `open_system` command to open a block’s Mask Parameters dialog box at the MATLAB command line or in an M program.

You can use `get_param` and `set_param` to access this mask parameter.

You can customize every feature of the Mask Parameters dialog box, including which parameters appear on the dialog box, the order in which they appear, parameter prompts, the controls used to edit the parameters, and the parameter callbacks (code used to process parameter values entered by the user).

However, changing the mask parameter value with `set_param` does *not* change the value of the underlying block variable. You cannot use `get_param` or `set_param` to access the value of the underlying variable, because it is hidden by the mask.

The `set_param` and `get_param` commands are insensitive to case differences in mask variable names. For example, suppose a model named `MyModel`

contains a masked subsystem named A that defines a mask variable named Volume. Then, the following line of code returns the value of the Volume parameter.

```
get_param(MyModel/A, 'voLUME')
```

However, case does matter when using a mask variable as the value of a block parameter inside the masked subsystem. For example, suppose a Gain block inside the masked subsystem A specifies VOLUME as the value of its Gain parameter. This variable name does not resolve in the masked subsystem's workspace, as it contains a mask variable named Volume. If the base workspace does not contain a variable named VOLUME, simulating MyModel produces an error.

Predefined Masked Dialog Parameters

The following predefined parameters are associated with masked dialogs.

MaskCallbacks

The value of this parameter is a cell array of strings that specify callback expressions for the dialog's user-defined parameter controls. The first cell defines the callback for the first parameter's control, the second for the second parameter control, etc. The callbacks can be any valid MATLAB expressions, including expressions that invoke M-file commands. This means that you can implement complex callbacks as M-files.

You can use either the Mask Editor or the MATLAB command line to specify mask callbacks. To use the Mask Editor to enter a callback for a parameter, enter the callback in the **Callback** field for the parameter.

The easiest way to set callbacks for a mask dialog at the MATLAB command line is to first select the corresponding masked dialog in a model or library window and then to issue a `set_param` command at the MATLAB command line. For example, the following code

```
set_param(gcb, 'MaskCallbacks', {'parm1_callback', '', ...  
'parm3_callback'});
```

defines callbacks for the first and third parameters of the masked dialog for the currently selected block. To save the callback settings, save the model or library containing the masked block.

MaskDescription

The value of this parameter is a string specifying the description of this block. You can change a masked block's description dynamically by setting this parameter in a mask callback.

MaskDisplay

The value of this parameter is string that specifies the drawing commands for the block's icon.

MaskEnables

The value of this parameter is a cell array of strings that define the enabled state of the user-defined parameter controls for this dialog. The first cell defines the enabled state of the control for the first parameter, the second for the second parameter, etc. A value of 'on' indicates that the corresponding control is enabled for user input; a value of 'off' indicates that the control is disabled.

You can enable or disable user input dynamically by setting this parameter in a callback. For example, the following command in a callback

```
set_param(gcb, 'MaskEnables', {'on', 'on', 'off'});
```

would disable the third control of the currently open masked block's dialog. Disabled controls are colored gray to indicate visually that they are disabled.

MaskInitialization

The value of this parameter is string that specifies the initialization commands for the mask workspace.

MaskPrompts

The value of this parameter is a cell array of strings that specify prompts for user-defined parameters. The first cell defines the prompt for the first parameter, the second for the second parameter, etc.

MaskType

The value of this parameter is the mask type of the block associated with this dialog.

MaskValues

The value of this parameter is a cell array of strings that specify the values of user-defined parameters for this dialog. The first cell defines the value for the first parameter, the second for the second parameter, etc.

MaskVisibilities

The value of this parameter is a cell array of strings that specify the visibility of the user-defined parameter controls for this dialog. The first cell defines the visibility of the control for the first parameter, the second for the second parameter, etc. A value of 'on' indicates that the corresponding control is visible; a value of 'off' indicates that the control is hidden.

You can hide or show user-defined parameter controls dynamically by setting this parameter in the callback for a control. For example, the following command in a callback

```
set_param(gcb,'MaskVisibilities',{'on','off','on'});
```

would hide the control for the currently selected block's second user-defined mask parameter. The Simulink software expands or shrinks a dialog to show or hide a control, respectively.

Note For a full list of predefined masked block parameters see the Mask Parameters reference page.

Notes on Mask Parameter Storage

- 1 The `MaskPromptString` parameter stores the **Prompt** field values for all mask dialog box parameters as a string, with individual values separated by vertical bars (`|`), for example:

```
"Slope: | Intercept: "
```

- 2 The `MaskStyleString` parameter stores the **Type** field values for all mask dialog box parameters as a string, with individual values separated by commas. The **Popup strings** values appear after the popup type, as shown in this example:

```
"edit, checkbox, popup(red|blue|green) "
```

- 3 The `MaskValueString` parameter stores the values of all mask dialog box parameters as a string, with individual values separated by a vertical bar (`|`). The order of the values is the same as the order in which the parameters appear on the dialog box, for example:

```
"2|5"
```

- 4 The `MaskVariables` parameter stores the **Variable** field values for all mask dialog box parameters as a string, with individual assignments separated by semicolons. A sequence number indicates the prompt that is associated with a variable. A special character preceding the sequence number indicates whether the parameter value is evaluated or used literally. An at-sign (`@`) indicates evaluation; an ampersand (`&`) indicates literal usage. For example:

```
"a=@1;b=&2; "
```

This string defines two **Variable** field values:

- The value entered in the first parameter field is evaluated in the MATLAB workspace, and the result is assigned to variable `a` in the mask workspace.
- The value entered in the second parameter field is not evaluated, but is assigned literally to variable `b` in the mask workspace.

Understanding Mask Code Execution

In this section...

“Specifying Mask Initialization” on page 9-44

“Callback” on page 9-48

“Mask Initialization Code” on page 9-50

“Debugging Masks that Use M-Code” on page 9-50

Specifying Mask Initialization

Drawing Commands Execution

Simulink executes the drawing commands in the **Drawing Commands** pane in the sequence in which the commands appear whenever you:

- Load the model
- Run or update the block diagram
- Apply any changes made in the Mask Parameters dialog box by clicking **Apply** or **OK**.
- Apply any changes made in the Mask Editor by clicking **Apply** or **OK**
- Make changes to the block diagram that affect the appearance of the block, such as rotating the block
- Copy the masked block within the same model or between different models

See “Understanding Mask Code Execution” on page 9-44 for information about the execution order of different types of code in a block mask.

Mask Editor Initialization Pane

- A masked subsystem’s initialization code can refer only to variables in its local workspace.

When the block is referenced within or copied into a model, the specified default values appear on the block’s Mask Parameters dialog box. You cannot

use mask initialization code to change mask parameter default values in a library block or any other block.

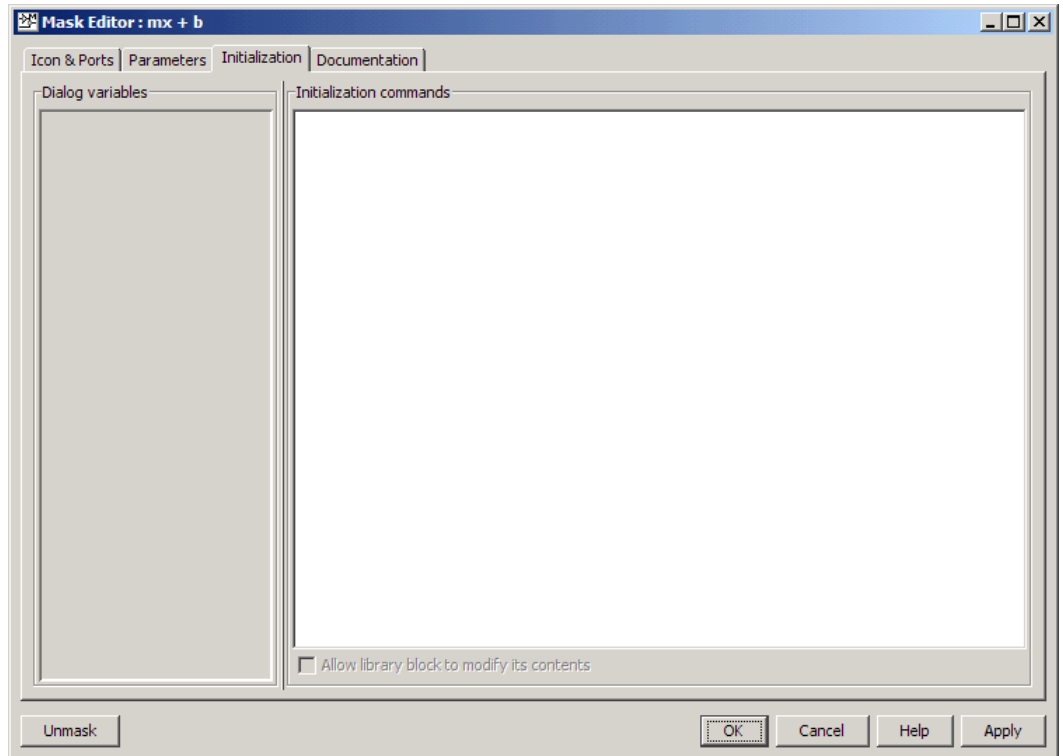
The **Initialization** pane enables you to specify initialization commands (see).

This example demonstrates how to create an improved icon for the `mx + b` sample masked subsystem discussed earlier in this section. First you must enter the following initialization commands to define the data that enables the drawing command to produce an accurate icon regardless of the shape of the block:

```
pos = get_param(gcf, 'Position');
width = pos(3) - pos(1); height = pos(4) - pos(2);
x = [0, width];
if (m >= 0), y = [0, (m*width)]; end
if (m < 0), y = [height, (height + (m*width))]; end
```

The drawing command that generates this icon is `plot(x,y)`.

Use the Mask Editor **Initialization** pane to specify block mask initialization commands:



Reference information about the **Initialization** pane appears in XREF.

Dialog variables

The **Initialization** pane includes the following controls.

Use the Mask Editor **Initialization** pane to enter MATLAB commands that initialize a masked block:

The **Dialog variables** list displays the names of the variables associated with the subsystem's mask parameters, i.e., 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 Command Window. For information on debugging initialization commands, see .

Summary of Initialization Command Execution

When you open a model, initialization commands execute for all masked blocks that are visible because they reside at the top level of the model or in an open subsystem. Initialization commands for masked blocks that are not initially visible execute when you open the subsystem or model that contains the blocks.

When you load a model into memory without displaying the model graphically, no initialization commands initially run for any masked blocks. See “Loading a Model” on page 1-7 and `load_system` for information about loading a model without displaying it.

Initialization commands for all masked blocks in a model 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`, by using the Mask Editor or `set_param`.
- Change the appearance of the masked block, for example by resizing or rotating the block.

- Change the value of a mask parameter by using the block dialog or `set_param`.
- Copy the masked block within the same model or between different models.

Initialization Command Limitations

Mask initialization commands must observe the following rules:

- Do not use initialization code to create dynamic mask dialogs, i.e., dialogs whose appearance or control settings change depending on changes made to other control settings. Instead, use the mask callbacks provided specifically for this purpose (see for more information).
- Avoid prefacing variable names in initialization commands with `L_` and `M_` to prevent undesirable results. These specific prefixes are reserved for use with internal variable names.
- 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 unresolvable 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 unresolvable symbol error.

Callback

Enter the MATLAB code that you want the Simulink software to execute when a user applies a change to the selected parameter, i.e., selects the **Apply** or **OK** button on the Mask Parameters dialog box. You can use such callbacks to create dynamic dialogs, i.e., dialogs whose appearance changes, depending on changes to control settings made by the user, e.g., enabling an edit field when a user checks a check box (see more information).

Note Callbacks are not intended to be used to alter the contents of a masked subsystem. Altering a masked subsystem's contents in a callback, for example by adding or deleting blocks or changing block parameter values, can trigger errors during model update or simulation. If you need to modify the contents of a masked subsystem, use the mask's initialization code to perform the modifications (see).

The callback can create and reference variables only in the block's base workspace. If the callback needs the value of a mask parameter, it can use `get_param` to obtain the value, e.g.,

```
if str2num(get_param(gcf, 'g'))<0
    error('Gain is negative.')
end
```

The Simulink software executes the callback commands when you

- Open the Mask Parameters dialog box. Callback commands execute top down, starting with the top mask dialog parameter
- Modify a parameter value in the Mask Parameters dialog box then change the cursor's focus, i.e., press the **Tab** key or click into another field in the dialog

Note The callback commands are not executed when the parameter value is modified using the `set_param` command.

- Modify the parameter value, either in the Mask Parameters dialog box or via a call to `set_param`, then apply the change by clicking **Apply** or **OK**. Any mask initialization commands execute after the callback commands. (See .)
- Hover over the masked block to see the block's data tip, when the data tip contains parameter names and values. The callback executes again when the block data tip disappears.

Note The callback commands do not execute if the Mask Parameters dialog box is open when the block data tip appears.

For information on debugging dialog callbacks, see .

Mask Initialization Code

The initialization code is MATLAB code that you specify and that the Simulink software runs to initialize the masked subsystem at critical times, such as model loading and the start of a simulation run (see). You can use the initialization code to set the initial values of the masked subsystem's mask parameters.

Debugging Masks that Use M-Code

You can use MATLAB tools to debug mask initialization commands and dialog callbacks in the Mask Editor or the MATLAB Editor/Debugger.

You can debug initialization commands and parameter callbacks entered directly into the Mask Editor in these ways:

- Remove the terminating semicolon from a command to echo its results to the MATLAB Command Window.
- Place a keyboard commands in the code to stop execution and give control to the keyboard.

You can debug initialization commands and parameter callbacks written in M-files using the MATLAB Editor/Debugger in the same way that you would with any other M-file. When debugging M-file initialization commands, you can view the contents of the mask workspace. When debugging M-file parameter callbacks, you can access only the block's base workspace. If you need the value of a mask parameter, use the `get_param` command.

You *cannot* debug icon drawing commands using the MATLAB Editor/Debugger. Use the syntax examples provided in the Mask Editor's **Icon** pane to help solve errors in the icon drawing commands.

Creating Dynamic Mask Dialog Boxes

In this section...

“Setting Nested Masked Block Parameters” on page 9-51

“About Dynamic Masked Dialog Boxes” on page 9-51

“Show parameter” on page 9-52

“Enable parameter” on page 9-52

“Setting Masked Block Dialog Parameters” on page 9-52

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 unresolvable 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 unresolvable symbol error.

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 parameter can cause related mask dialog parameters to be set to appropriate values.

Creating a dynamic masked dialog entails using the Mask Editor in combination with the `set_param` command. Specifically, you use the Mask Editor to define the dialog's parameters, both static and dynamic. For each dynamic parameter, you enter a callback function that defines the dialog's response to changes to that parameter (see). The callback function can in turn use the `set_param` command to set mask dialog parameters that affect the appearance and settings of other controls on the dialog (see). Finally, you save the model or library containing the masked subsystem to complete the creation of the dynamic masked dialog.

Show parameter

The selected parameter appears on the Mask Parameters dialog box only if this option is checked (the default).

Enable parameter

Clearing this option grays the selected parameter's prompt and disables its edit control. This means that the user cannot set the value of the parameter.

Setting Masked Block Dialog Parameters

The following example creates a mask dialog 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 pop-up menu determines the visibility of an edit field used to specify the user-defined gain.

- 1 Mask a subsystem as described in steps one and two in Masking a Subsystem.
- 2 Select the **Parameters** pane on the Mask Editor.

3 Add a parameter.

- Enter **Gain:** in the **Prompt** field
- Enter **gain** in the **Variable** field
- Select **popup** in the **Type** field

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'}),

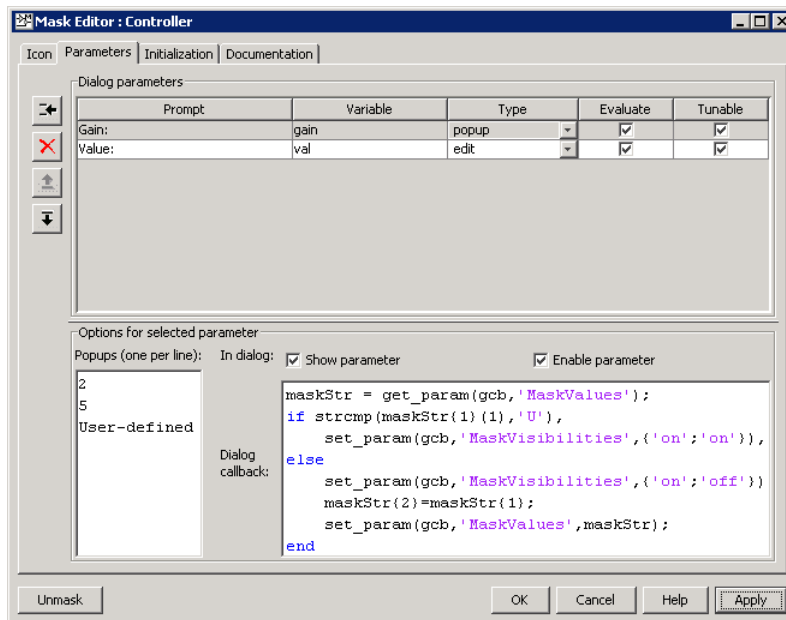
    % Set the string in the Values field equal to the
    % string selected in the Gain pop-up menu.

    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 **Show parameter** in the **Options for selected parameter** group. This turns the visibility of this parameter off, by default.

7 Select **Apply** on the Mask Editor. The Mask Editor now looks like this when the gain parameter is selected and comments are removed from the mask callback code:



Double-clicking on the new masked subsystem opens the Mask Parameters 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.

Creating Dynamic Masked Subsystems

In this section...

“Allow library block to modify its contents” on page 9-55

“Creating Self-Modifying Masks for Library Blocks” on page 9-55

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’s initialization code to modify the contents of the masked subsystem, i.e., 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.

Creating 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 parameter 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 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 “Modifying a Library” on page 8-15).
- 2 Select the block in the library.

- 3 Select **Edit Mask** from the **Edit** menu or the block's context menu. The Mask Editor opens.
- 4 In the Mask Editor's **Initialization** pane, select the **Allow library block to modify its contents** option.
- 5 Enter the code that modifies the masked subsystem in the mask's **Initialization** pane.

Note Do not enter code that structurally modifies the masked subsystem in a dialog parameter callback (see). 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')
```

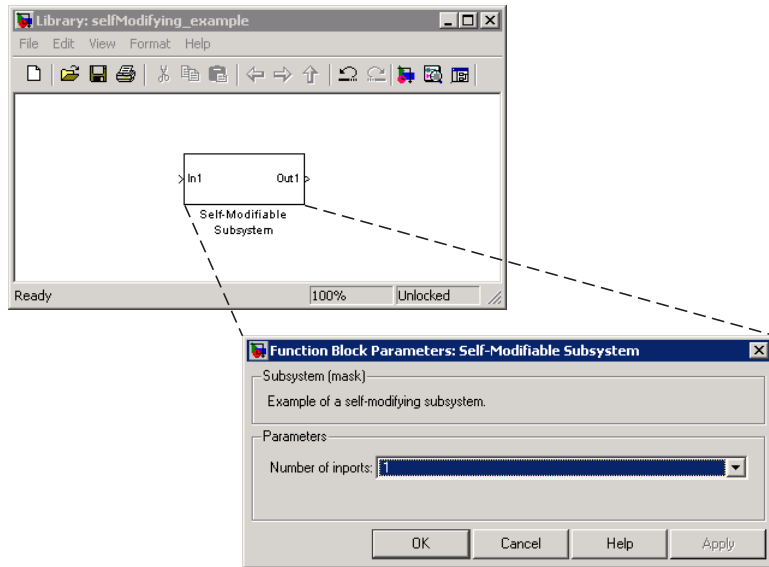
- 2 Specify that the block is self-modifying by using the following command:

```
set_param(block_name, 'MaskSelfModifiable', 'on')
```

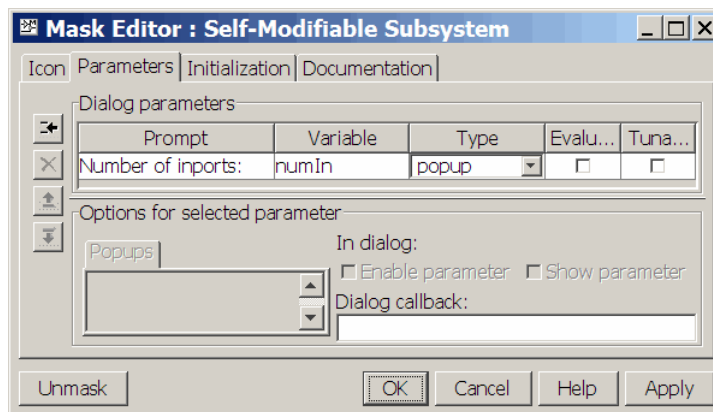
where `block_name` is the full path to the block in the library.

Self-Modifying Mask Example

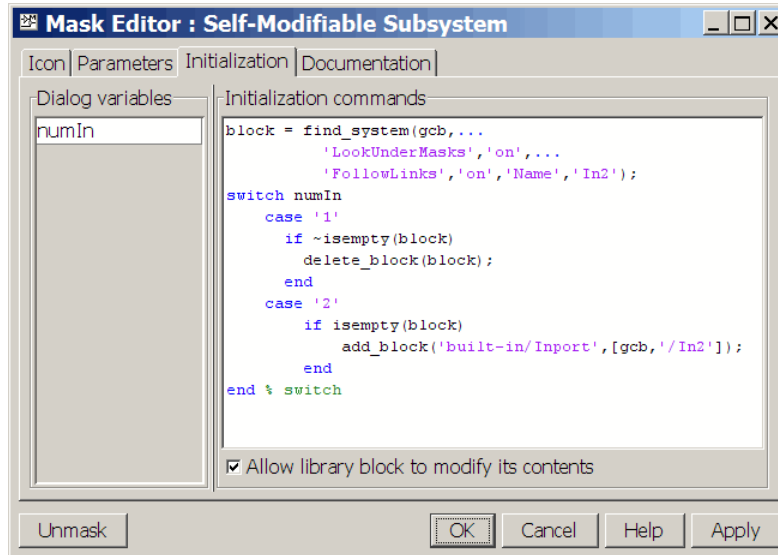
The library `selfModifying_example.mdl` contains a masked subsystem that modifies its number of input ports based on a selection made in the subsystem's Mask Parameters dialog box.



Select the subsystem then select **View Mask** from the **Edit** menu or the block's context menu. The Mask Editor opens. The Mask Editor's **Parameters** pane defines one mask parameter variable numIn that stores the value for the **Number of inputs** option. This Mask Parameters dialog box callback adds or removes Input ports inside the masked subsystem based on the selection made in the **Number of inputs** list.



To allow the dialog callback to function properly, the **Allow library block to modify its contents** option on the Mask Editor's **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 imports** list would produce an error.



Best Practices for Using M-Code in Masks

Working with Signals

- “Signal Basics” on page 10-2
- “Validating Signal Connections” on page 10-20
- “Displaying Signal Sources and Destinations” on page 10-21
- “Determining Output Signal Dimensions” on page 10-25
- “Checking Signal Ranges” on page 10-30
- “Introducing the Signal and Scope Manager” on page 10-37
- “Using the Signal and Scope Manager” on page 10-43
- “The Signal Selector” on page 10-48
- “Initializing Signals and Discrete States” on page 10-53
- “Working with Test Points” on page 10-61
- “Displaying Signal Properties” on page 10-64
- “Working with Signal Groups” on page 10-69

Signal Basics

In this section...

“About Signals” on page 10-2
“Creating Signals” on page 10-3
“Naming Signals” on page 10-3
“Displaying Signal Values” on page 10-5
“Signal Line Styles” on page 10-6
“Signal Labels” on page 10-7
“Signal Data Types” on page 10-8
“Signal Dimensions” on page 10-8
“Complex Signals” on page 10-12
“Virtual Signals” on page 10-12
“Mux Signals” on page 10-15
“Control Signals” on page 10-18
“Composite Signals” on page 10-18
“Signal Glossary” on page 10-19

About 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.

Simulink defines signals as 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 by B depends on the signal output by A.

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).

Note It is tempting but misleading to think of Simulink signals as traveling along the lines that connect blocks the way electrical signals travel along a telephone 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

You can create signals by creating source blocks in 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 the model. See “Sources” for information on blocks that you can use to create signals in a model. You can also use the Signal & Scope Manager to create signals in your model without using blocks. See “Introducing the Signal and Scope Manager” on page 10-37 for more information.

Naming Signals

You can give any signal a name. The syntactic requirements for a signal name vary depending on how the name will be used. The three 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 “Logging Signals” on page 15-3.) 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 15-9.
- 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.

To avoid any doubt about whether a signal name will serve all present and future purposes, make every signal name a legal MATLAB identifier. Otherwise, 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.

Assigning a Signal Name

To assign a name to a signal, double-click the signal. An edit box appears next to the signal near where you double-clicked. Enter the desired name, then click somewhere outside the edit box. The signal now has the specified name, and a label showing that 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.

Another way to name a signal is to right-click the signal, choose **Signal Properties** from the Context menu, enter a name in the **Signal Name** field, then click **OK** or **Apply**. A label showing the name then appears on every branch of the signal. See “Signal Properties Dialog Box” for more information.

You can also use the API to set the name parameter of the port or line that represents the signal:

```
p = get_param(gcf, 'PortHandles')
l = get_param(p.Inport, 'Line')
set_param(l, 'Name', 's9')
```

Changing a Signal Name

To change the name of a signal, click to set the cursor in any label that shows the name, then change the text as needed; or edit the name in the **Signal**

Properties > Signal Name field. All labels automatically update to reflect the change.

To change the location of a label that displays a signal name, drag it with the mouse. You cannot drag a label away from its signal, but only to a different location adjacent to the signal.

Deleting a Signal Name



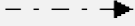
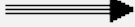
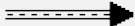

To delete a signal's name, leaving it nameless, delete all characters in the name, in any label on the signal or in the **Signal Properties > Signal Name** field. To delete a label without deleting the signal name, click near the edge of the label to select its surrounding box, then press Delete. The label disappears, but the signal name itself is unaffected.

Displaying Signal Values

As with creating signals, you can use either blocks or 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. See “Sinks” in the Simulink block reference for information on blocks that you can use to display signals in a model.

Signal Line Styles

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 in Simulink diagrams. The signal types and their line styles are as follows:

Signal Type	Line Style	Description
Scalar and Nonscalar		Simulink uses a thin, solid line to represent a diagram's scalar and nonscalar signals.
Nonscalar		When the Wide nonscalar lines option is enabled, Simulink uses a thick, solid line to represent a diagram's nonscalar signals. See also "Using Muxes" on page 10-16.
Control		Simulink uses a thin, dash-dot line to represent a diagram's control signals.
Virtual Bus		Simulink uses a triple line with a solid core to represent a diagram's virtual signal buses. See .
Nonvirtual Bus		Simulink uses a triple line with a dotted core to represent a diagram's nonvirtual signal buses. See .
Variable-Size		Simulink uses a solid wide line with a white dotted core to represent a variable-size signal. See "Variable-Size Signal Basics" on page 11-2.

Other than using the **Wide nonscalar lines** option to display nonscalar signals as thick, solid lines, you cannot customize or control the line style with which Simulink displays signals. See "Wide Nonscalar Lines" on page 10-67 for more information about this option.

Note As you construct a block diagram, Simulink uses a thin, solid line to represent all signal types. The lines are then redrawn using the specified line styles only after you update or start simulation of the block diagram.

Signal Labels

A signal label is text that appears next to the line that represents a signal that has a name. The signal label displays the signal's name. In addition, if the signal is a virtual signal (see “Virtual Signals” on page 10-12) and its **Show propagated signals** property is on (see “Show propagated signals”), the label displays the names of the signals that make up the virtual signal.

Simulink creates a label for a signal when you assign it a name in the “Signal Properties Dialog Box”. You can change the signal's name by editing its label on the block diagram. To edit the label, left-click the label. Simulink replaces the label with an edit field. Edit the name in the edit field, then click outside the label to apply the change.

A signal's label displays the signal's name. A virtual signal's label optionally displays the signals it represents in angle brackets. You can edit a signal's label, thereby changing the signal's name.

To create a signal label (and thereby name the signal), double-click the line that represents the signal. The text cursor appears. Enter the name and click anywhere outside the label to exit label editing mode.

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.

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.

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.

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.

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.

To change the font of a signal label, select the signal, choose **Font** from the **Format** menu, then select a font from the **Set Font** dialog box.

Signal Data Types

Data type refers to the format used to represent signal values internally. The data type of Simulink signals is double by default. However, you can create signals of other data types. Simulink supports the same range of data types as MATLAB. See “Working with Data Types” on page 13-2 for more information.

Signal Dimensions

Simulink blocks can output one-, two-, or multidimensional signals. A one-dimensional (1-D) signal consists of a stream 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 stream of two-dimensional arrays output at a frequency of one 2-D array (matrix) per block sample time. A multidimensional signal consists of a stream of multidimensional (2 or more dimensions) arrays output at a frequency of one array per block sample time (see “Multidimensional Arrays” in the MATLAB Getting Started documentation for information on multidimensional arrays). 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.

Only the following Simulink blocks support multidimensional signals. Simulink supports signals with up to 32 dimensions. Do not use signals with more than 32 dimensions.

- Abs
- Assertion
- Assignment
- Bitwise Operator
- Bus Assignment
- Bus Creator
- Bus Selector
- Check Discrete Gradient
- Check Dynamic Gap
- Check Dynamic Lower Bound
- Check Dynamic Range
- Check Dynamic Upper Bound
- Check Input Resolution
- Check Static Gap
- Check Static Lower Bound
- Check Static Range
- Check Static Upper Bound
- Compare to Constant
- Compare to Zero
- Complex to Magnitude-Angle
- Complex to Real-Imag
- Concatenate
- Constant
- Data Store Memory

- Data Store Read
- Data Store Write
- Data Type Conversion
- Data Type Scaling Strip
- Embedded MATLAB Function
- Environment Controller
- Extract Bits
- From
- From Workspace
- Gain (only if the **Multiplication** parameter specifies Element-wise ($K.*u$))
- Goto
- Ground
- IC
- Inport
- Level-2 M-File S-Function
- Logical Operator
- Magnitude-Angle to Complex
- Manual Switch
- Math Function (no multidimensional signal support for the transpose and hermitian functions)
- Memory
- Merge
- MinMax
- Model
- Multiport Switch
- Outport

- Product, Product of Elements — only if the **Multiplication** parameter specifies `Element-wise(.*)`
- Probe
- Random Number
- Rate Transition
- Real-Imag to Complex
- Relational Operator
- Reshape
- Scope, Floating Scope
- Selector
- S-Function
- Signal Conversion
- Signal Generator
- Signal Specification
- Slider Gain
- Squeeze
- Subsystem, Atomic Subsystem, CodeReuse Subsystem
- Add, Subtract, Sum, Sum of Elements — along specified dimension
- Switch
- Terminator
- To Workspace
- Trigonometric Function
- Unary Minus
- Uniform Random Number
- Unit Delay
- Width
- Wrap to Zero

Simulink blocks vary in the dimensionality of the signals they can accept or output. Some blocks can accept or output signals of any dimensions. Some can accept or output only scalar or vector signals. To determine the signal dimensionality of a particular block, see the block's description in Blocks — Alphabetical List in the online Simulink reference. See “Determining Output Signal Dimensions” on page 10-25 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 simulation. That is, the size of a signal must remain constant while the simulation executes. You can alter a signal's size only after terminating the simulation.

Complex Signals

The values of Simulink signals can be complex numbers. A signal whose values are complex numbers is called a complex signal. You can introduce a complex-valued signal into a model in the following ways:

- 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.

You can 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 in Blocks — Alphabetical List in the online Simulink reference.

Virtual Signals

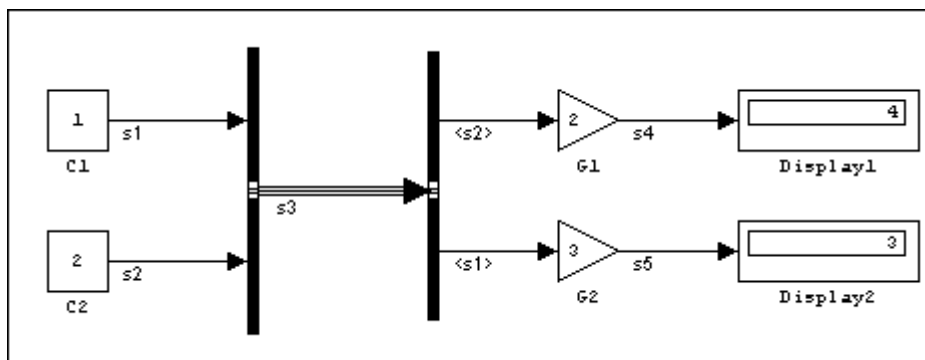
A *virtual signal* is a signal that graphically represents other signals or parts of other signals. Virtual signals are purely graphical entities; they have no mathematical or physical significance. Simulink ignores them when simulating a model, and they do not exist in generated code. Some blocks,

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.

Whenever you update or run a model, Simulink determines the nonvirtual signal(s) represented by the model's virtual signal(s), using a procedure known as *signal propagation*. When running the model, Simulink uses the corresponding nonvirtual signal(s), determined via signal propagation, to drive the blocks to which the virtual signals are connected.

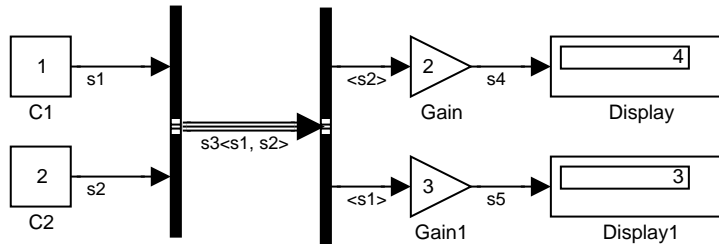
Consider, for example, the following model.



The signals driving Gain blocks G1 and G2 are virtual signals corresponding to signals s2 and s1, respectively. Simulink determines this automatically whenever you update or simulate the model.

Displaying the Nonvirtual Components of Virtual Signals

The **Show propagated signals** option (see “Signal Properties Dialog Box”) displays the nonvirtual signals represented by virtual signals in the labels of the virtual signals.



When you change the name of a nonvirtual signal, Simulink immediately updates the labels of all virtual signals that represent the nonvirtual signal and whose **Show propagated signals** is on, except if the path from the nonvirtual signal to the virtual signal includes an unresolved reference to a library block. In such cases, to avoid time-consuming library reference resolutions while you are editing a block diagram, Simulink defers updating the virtual signal’s label until you update the model’s block diagram either directly (e.g., by pressing **Ctrl+D**) or by simulating the model.

Note Virtual signals can represent virtual as well as nonvirtual signals. For example, you can use a Bus Creator block to combine multiple virtual and nonvirtual signals into a single virtual signal. If during signal propagation, Simulink determines that a component of a virtual signal is itself virtual, Simulink uses signal propagation to determine the nonvirtual components of the virtual component. This process continues until Simulink has determined all nonvirtual components of a virtual signal.

To display the signal(s) represented by a virtual signal, right-click the signal line and select **Signal Properties**. The Signal Properties dialog box (see “Signal Properties Dialog Box”) is displayed. Select the **Show propagated signals** check box, then click **OK**. Simulink displays the signals represented by the virtual signal in brackets in the label. After you select the **Show propagated signals** check box, you can provide a signal name. As a shortcut,

you can also click the signal's label, clear the name (if one exists) and enter an angle bracket (<). Click anywhere outside the signal's label. Simulink exits label editing mode and displays the signals represented by the virtual signal in brackets in the label. After you enable the **Show propagated signals** parameter, you can enter a signal name in the label.

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 should not be confused with 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 has no purpose except to simplify a model's visual appearance. 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, including performing calculations on it. 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. Models can use this capability to perform computations on multiple vectors without the overhead of first 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, by definition, a *virtual* vector signal: its constituent signals 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.

If you want to create a composite signal, in which the constituent signals retain their identities and can have different data types, use a Bus Creator block rather than a Mux block, as described in Chapter 12, “Using Composite

Signals”. Although you can use a Mux block to create a composite signal in some cases, The MathWorks discourages this practice. See “Avoiding Mux/Bus Mixtures” on page 12-57 for more information.

Using Muxes

The Signal Routing library provides two virtual blocks that you can use for implementing muxes:

Mux

Combine several input signals into a mux (virtual vector) signal

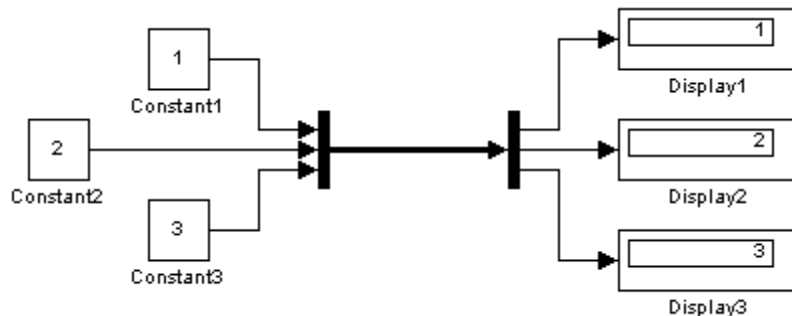
Demux

Extract and output the values in a mux (virtual vector) signal

To implement a mux signal:

- 1 Clone a Mux and Demux block from the Signal Routing library.
- 2 Set the Mux block’s **Number of inputs** and the Demux block’s **Number of outputs** properties 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. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the mux signal, is wide because the model has been built with **Format > Port/Signal Displays > Wide Nonscalar Lines** enabled in the model menu. See “Displaying Signal Properties” on page 10-64 for details.

Signals input to a Mux block can 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.

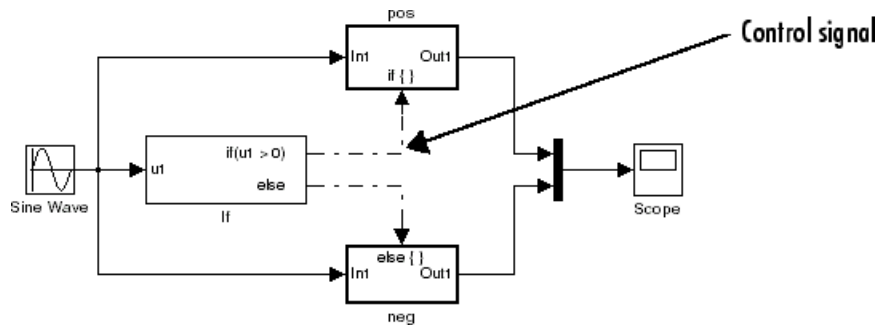
The values in all signals input to a Mux block must have the same data type. If they do not, the block will output a bus rather than a mux unless you have configured the model to disable this practice. The MathWorks discourages using Mux blocks to create buses. See “Avoiding Mux/Bus Mixtures” on page 12-57 for details.

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, but it cannot rearrange the order of those values. See the Demux block documentation for details.

A Demux block can input a bus unless you have configured the model to disable this practice. The MathWorks discourages using Demux blocks to access buses under any circumstances. See “Avoiding Mux/Bus Mixtures” on page 12-57 for details.

Control Signals

A *control signal* is a signal used by one block to initiate execution of another block, e.g., a function-call or action subsystem. When you update or start simulation of a block diagram, Simulink uses a dash-dot pattern to redraw lines representing the diagram's control signals as illustrated in the following example.



Composite Signals

The Simulink software provides capabilities that you can use to 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. Buses can simplify the appearance of a model when many parallel signals exist, and help to clarify generated code. A bus can be either virtual or nonvirtual. See Chapter 12, “Using Composite Signals” for details.

Signal Glossary

The following table summarizes the terminology used to describe signals in the Simulink user interface and documentation.

Term	Meaning
Bus	A Simulink composite signal
Complex signal	Signal whose values are complex numbers.
Composite Signal	A signal made up of other signals, optionally including other composite signals. See Chapter 12, “Using Composite Signals”.
Data type	Format used to represent signal values internally. See “Working with Data Types” on page 13-2.
Matrix	Two-dimensional signal array.
Mux	A virtual vector created with a Mux block.
Real signal	Signal whose values are real (as opposed to complex) numbers.
Scalar	One-element array.
Signal bus	Same as a composite signal
Signal propagation	Process used by the Simulink to determine attributes of signals and blocks, such as data types, labels, sample time, dimensionality, and so on, that are determined by connectivity.
Size	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.
Test point	A signal that must be accessible during simulation. See “Working with Test Points” on page 10-61 for more information.
Vector	One-dimensional signal array.
Virtual signal	Signal that represents another signal or set of signals.
Width	Size of a vector signal.

Validating Signal Connections

Many Simulink blocks have limitations on the types of signals they can accept. Before simulating a model, Simulink checks all blocks to ensure that they can accommodate the types of signals output by the ports to which they are connected. If any incompatibilities exist, Simulink reports an error and terminates the simulation.

To detect signal compatibility errors before running a simulation, choose **Update Diagram** from the Simulink **Edit** menu. Simulink reports any invalid connections found in the process of updating the diagram.

Displaying Signal Sources and Destinations

In this section...

“About Signal Highlighting” on page 10-21

“Highlighting Signal Sources” on page 10-21

“Highlighting Signal Destinations” on page 10-22

“Removing Highlighting” on page 10-23

“Resolving Incomplete Highlighting to Library Blocks” on page 10-23

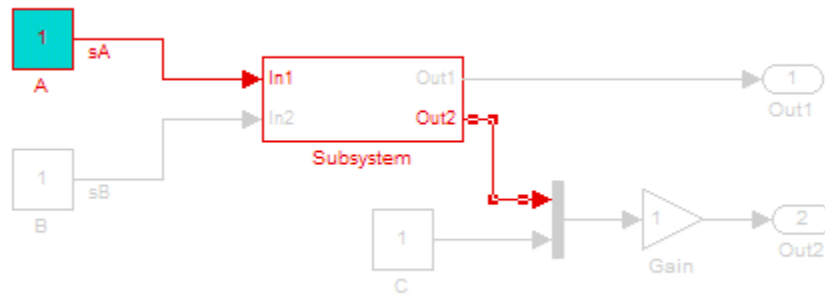
About Signal Highlighting

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 Chapter 12, “Using Composite Signals”.)

Highlighting Signal Sources

To display the source block(s) of a signal, choose the **Highlight 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

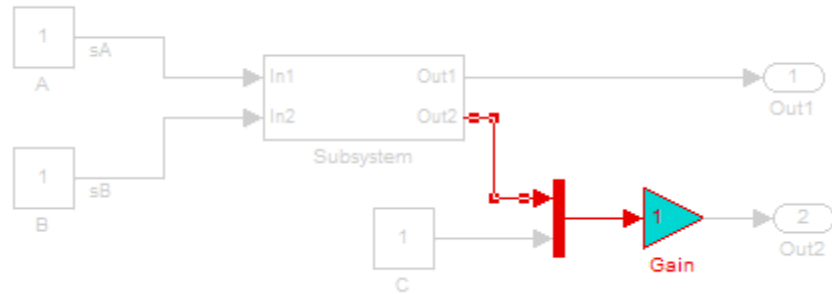


Highlighting Signal Destinations

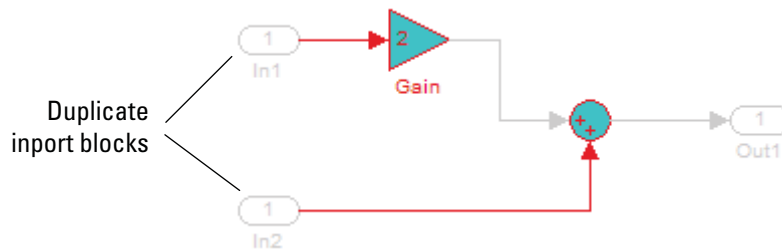
To display the destination blocks of a signal, choose the **Highlight to Destination** option from the context menu for the signal. This option highlights:

- 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 to Destination** option highlights the signal and destination block for In2 and In1, because In1 and In2 are duplicate inport blocks.



Removing Highlighting

To remove all highlighting, choose **Remove Highlighting** from the model's context menu, or choose **View > 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.

Determining Output Signal Dimensions

In this section...

“About Signal Dimensions” on page 10-25

“Determining the Output Dimensions of Source Blocks” on page 10-25

“Determining the Output Dimensions of Nonsource Blocks” on page 10-26

“Signal and Parameter Dimension Rules” on page 10-26

“Scalar Expansion of Inputs and Parameters” on page 10-28

About Signal Dimensions

If a block can emit nonscalar signals, the dimensions of the signals that the block outputs depend on the block’s parameters, if the block is a source block; otherwise, the output dimensions depend on the dimensions of the block’s 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 the “Sources” table in the online Simulink block reference 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 (i.e., not selected in the block’s 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 to specify the dimensions of the signals that they output. You can therefore use them 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 10-26).

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” on page 10-28) thus preserving the general rule.

Block Parameter Dimension Rule

In general, a block’s parameters must have the same dimensions as the corresponding inputs.

Two seeming exceptions exist to this 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 Parameters” on page 10-29) thus preserving the general rule.
- 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.

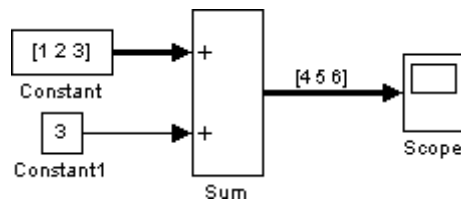
Scalar Expansion of Inputs and Parameters

Scalar expansion is the conversion of a scalar value into a nonscalar array of the same dimensions. Many Simulink blocks support scalar expansion of inputs and parameters. Block descriptions in the *Simulink Reference* indicate whether Simulink applies scalar expansion to a block’s inputs and parameters.

Scalar Expansion of Inputs

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. The elements of an expanded signal equal the value of the scalar from which the signal was expanded.

The following model illustrates scalar expansion of inputs. This model adds scalar and vector inputs. The input from block Constant1 is scalar expanded to match the size of the vector input from the Constant block. The input is expanded to the vector [3 3 3].

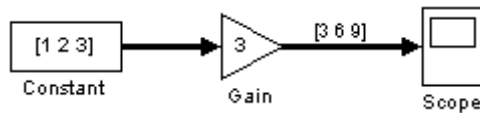


When a block’s output is a function of a parameter and the parameter is nonscalar, Simulink expands a scalar input to match the dimensions of the parameter. For example, Simulink expands a scalar input to a Gain block to match the dimensions of a nonscalar gain parameter.

Scalar Expansion of Parameters

If a block has a nonscalar input and a corresponding parameter is a scalar, Simulink expands the scalar parameter to have the same number of elements as the input. Each element of the expanded parameter equals the value of the original scalar. Simulink then applies each element of the expanded parameter to the corresponding input element.

This example shows that a scalar parameter (the Gain) is expanded to a vector of identically valued elements to match the size of the block input, a three-element vector.



Checking Signal Ranges

In this section...
“About Signal Range Checking” on page 10-30
“Blocks That Allow Signal Range Specification” on page 10-30
“Specifying Ranges for Signals” on page 10-31
“Checking for Signal Range Errors” on page 10-32

About Signal Range Checking

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
- Lookup Table
- Lookup Table (2-D)

- 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

See “Blocks — Alphabetical List” in the *Simulink Reference* for more information about these blocks and their parameters.

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 10-30 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 value, `[]`, is equivalent to `-Inf` for the minimum value and `Inf` for the maximum value. 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 10-28).

Note You cannot specify the minimum or maximum value as NaN.

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 10-31) and the block data type. That is, Simulink performs the following check:

$$\text{DataTypeMin} \leq \text{MinValue} \leq \text{VALUE} \leq \text{MaxValue} \leq \text{DataTypeMax}$$

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.

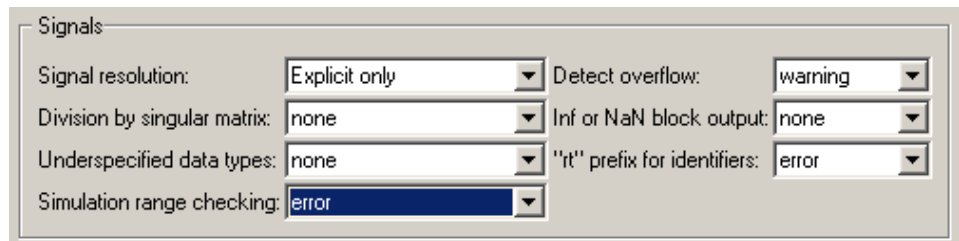
Enabling Simulation Range Checking

To enable the **Simulation range checking** diagnostic:

- 1 In your model window, select **Simulation > 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.

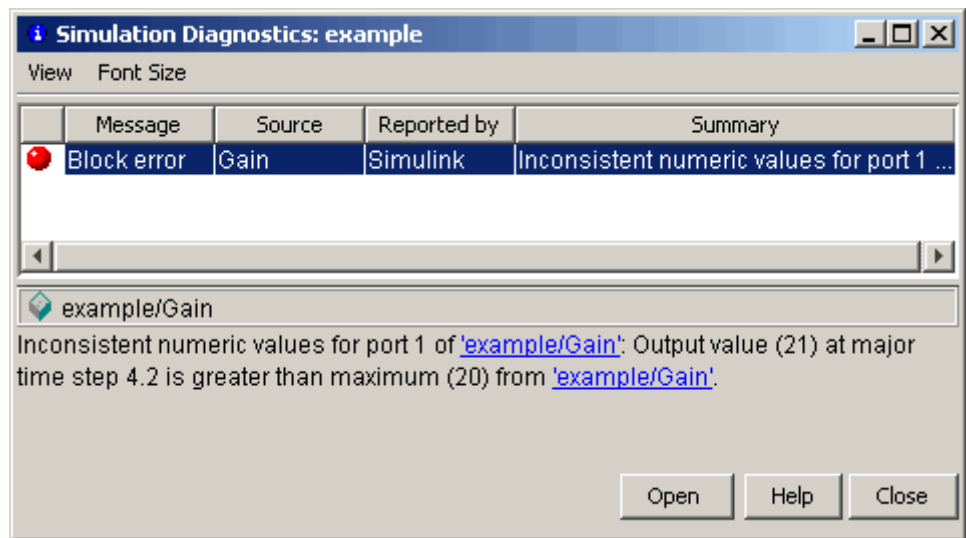
See “Simulation range checking” in the *Simulink Graphical User Interface* documentation for more information.

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 10-33).
- 2 In your model window, select **Simulation > Start** 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:

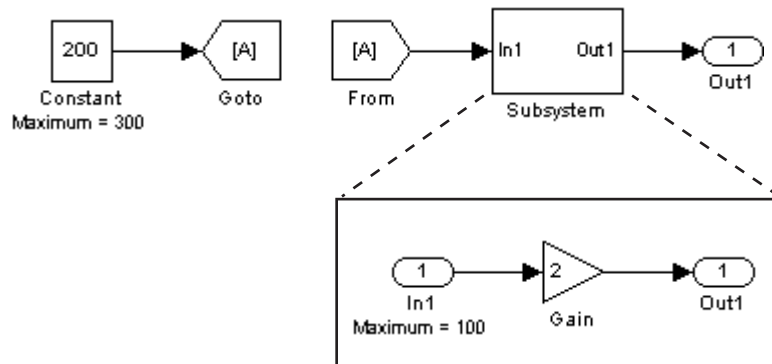
```
Warning: Inconsistent numeric values for port 1
of 'example/Gain': Output value (21) at major
time step 4.2 is greater than maximum (20) from
'example/Gain'.
```

Each message identifies the block whose output signal exceeds its specified range, and the time step at which this violation occurs.

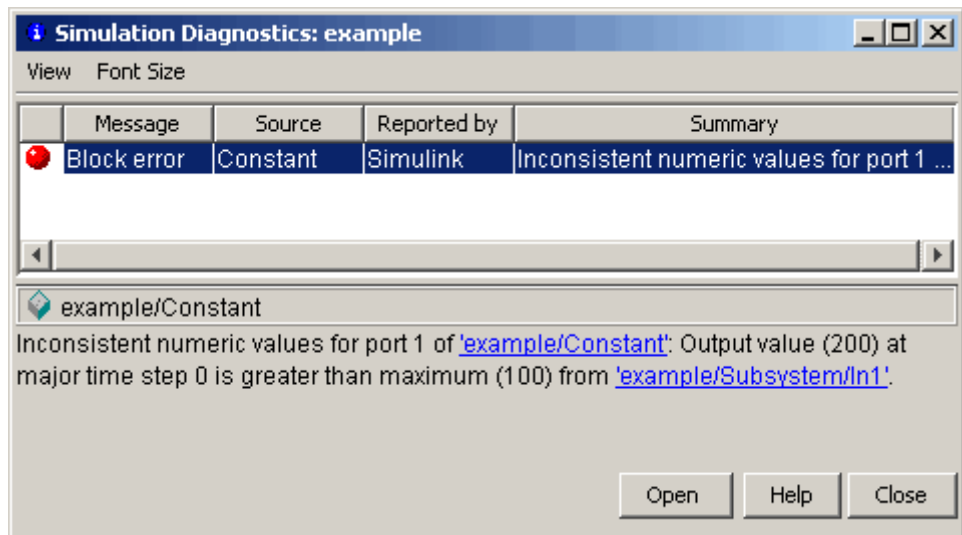
Signal Range Propagation for Virtual Blocks

Some virtual blocks (see “Virtual Blocks” on page 7-2) 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:



Introducing the Signal and Scope Manager

In this section...

“What is the Signal & Scope Manager?” on page 10-37

“Displaying the Signal and Scope Manager User Interface” on page 10-38

“Understanding the Signal and Scope Manager User Interface” on page 10-38

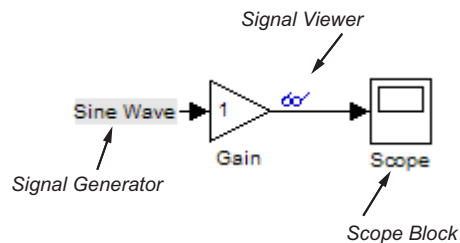
What is the Signal & Scope Manager?

The Signal & Scope Manager is a user interface to the Signal Viewers and Generator objects. From the Signal and Scope Manager you manage all signal generators and viewers from a central place.

Note The Signal and Scope Manager requires that you have Java™ enabled when you start MATLAB. This is the default.

What are Viewer and Generator Objects?

The small icons identifying a viewer or generator are called Viewer and Generator Objects. These objects are not the same as scope or signal blocks. 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.

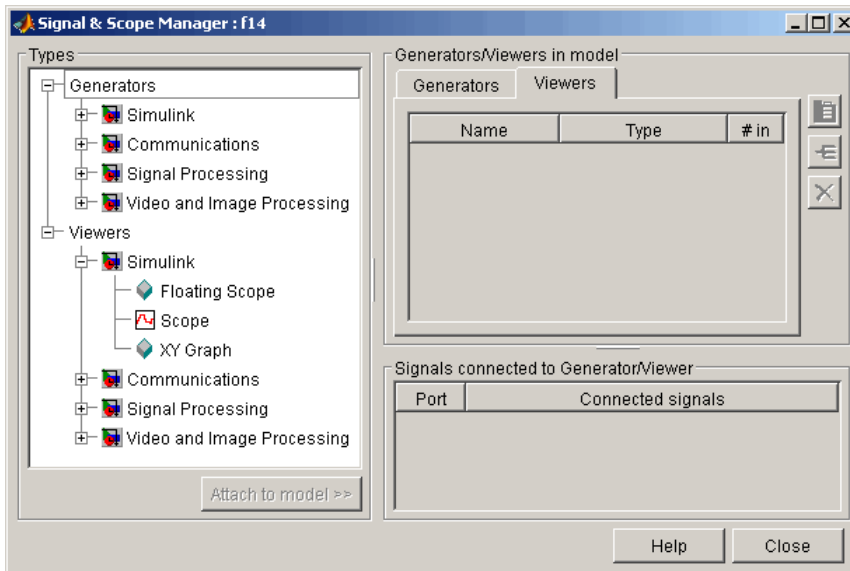


Displaying the Signal and Scope Manager User Interface

Access the Signal and Scope Manager from the model editor's **Tools** menu.

Alternatively, right click within your model and select **Signal & Scope Manager** from the context menu.

Understanding the Signal and Scope Manager User Interface



The Signal and Scope manager user interface is comprised of three panes:

- **Types.** Selects the viewer or generator to attach to your model. For more information, see “Types Pane” on page 10-39.
- **Generators/Viewers in model.** Selects signal sources and viewers for your model.

For more information on sources, see “Generators Tab” on page 10-39.

For more information on viewers, see “Viewers Tab” on page 10-40.

- **Signals connected to Generator/Viewer.** Manages the connections to the generators and viewers present in your model. For more information, see “Signals connected to Generator/Viewer Pane” on page 10-42.

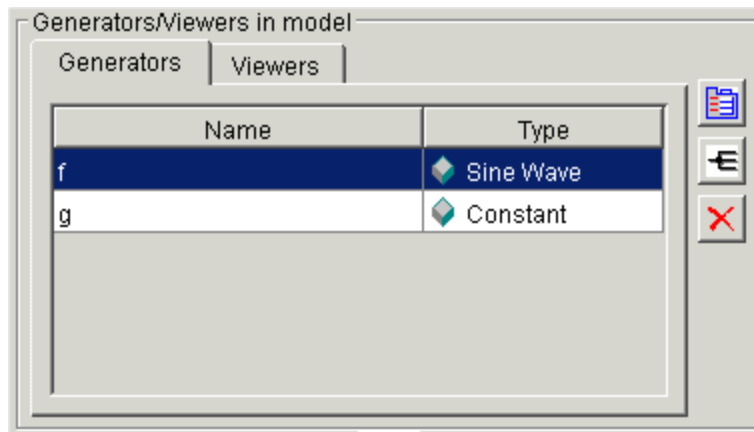
Types Pane

The **Types** pane shows the generators and viewers associated with the products installed on your system. Expand a products node list to show all the generators and viewers 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 “How Scope Blocks and Signal Viewers Differ” on page 24-3.

Generators Tab

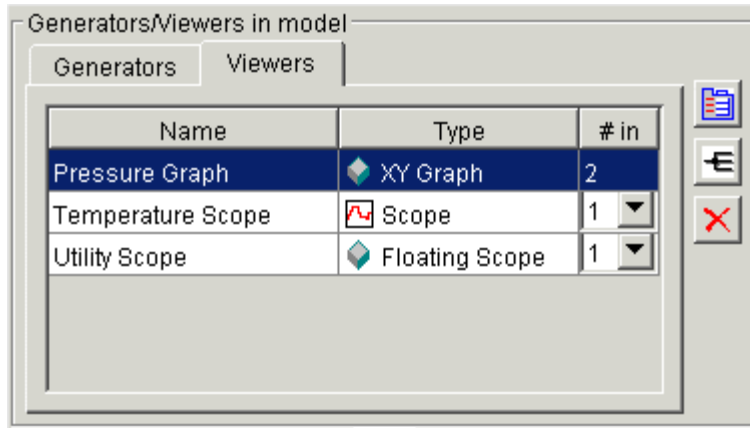
The **Generators** tab displays a table listing the generators associated with your model.



Each row corresponds to a generator. The columns specify each generator's name and type.

Viewers Tab




The **Viewers** tab displays a table listing the viewers present in your model.



Each row corresponds to a viewer. The columns specify each viewer's 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, pull down the list and select the desired value.

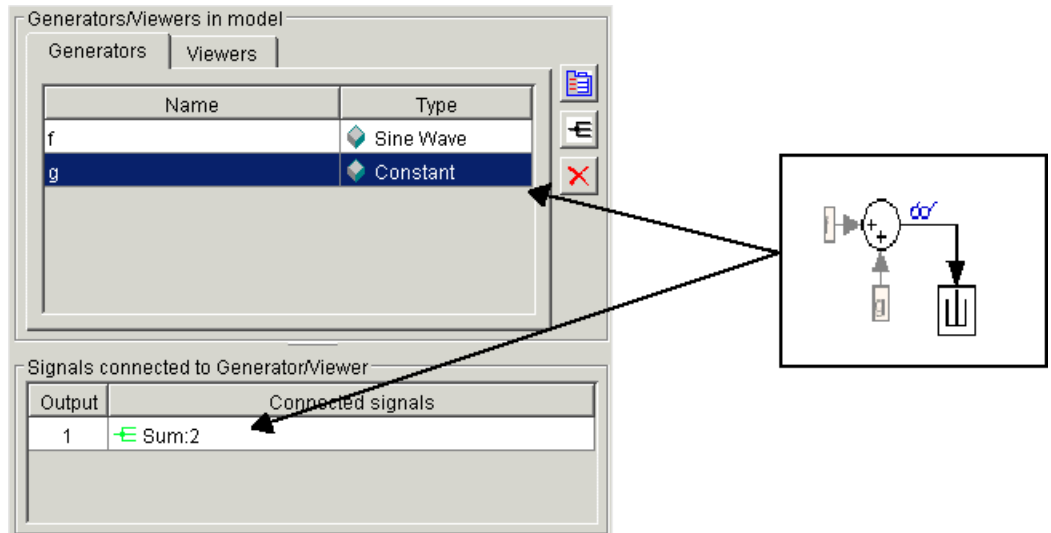
Edit Buttons

Use these buttons to manage generators and viewers after you have selected them in the Generators or Viewers table:

Button	Description
	<p>Opens the parameter dialog box for the selected generator or viewer.</p> <p>From the parameter dialog you view and change object parameters.</p> <p>See “Scope Viewer Parameters Dialog Box” on page 24-13 for more information.</p>
	<p>Opens the Signal Selector for the selected generator or viewer.</p> <p>You use the Signal Selector to connect and disconnect generators and viewers.</p> <p>See “The Signal Selector” on page 10-48 for information on the signal selector.</p>
	<p>Deletes the selected generator or viewer.</p>

Signals connected to Generator/Viewer Pane

This table lists the signals connected to the generator or viewer selected in the Generators/Viewers control group of the Signal and Scope Manager.



This graphic illustrates the table display when two generators are connected to a sum block. The Viewers display works in the same way.

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.

Connection Menu

Selecting a connection in the **Signals connected to Generator/Viewer** table and pressing the right button on your mouse displays a context menu. From this context menu you can:

- Open the Properties dialog
- Highlight the connections in your block diagram
- Open the Signal Selector

Using the Signal and Scope Manager

In this section...

“Introduction” on page 10-43

“Attaching a New Viewer or Generator” on page 10-43

“Creating a Multiple Axes Viewer” on page 10-44

“Adding Additional Signals to an Existing Viewer” on page 10-45

“Viewing Test Point Data” on page 10-45

“Adding Custom Viewers and Generators” on page 10-46

Introduction

This section shows you how to use the Signal and Scope Manager to perform some basic Viewer and Generator object tasks.

If you are not familiar with the Signal and Scope manager, or Viewer or Generator objects, or if you do not know how to display the Signal and Scope manager, see “Introducing the Signal and Scope Manager” on page 10-37.

To learn how to use and adjust the viewers you have created, see “The Scope Viewer Toolbar” on page 24-12.

Attaching a New Viewer or Generator

To connect a new viewer or generator to a signal in the currently selected model:

- 1 Display the Signal and Scope manager.
- 2 Select a viewer or generator from the **Types** pane.
- 3 Click **Attach to Model**.
- 4 Click the **Signal Selector** button to display the Signal Selector dialog.

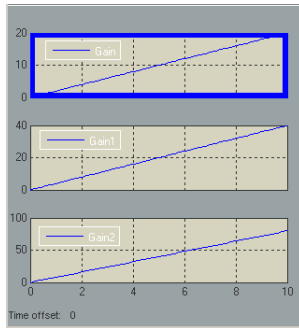
For more information, see “The Signal Selector” on page 10-48.

- 5 Select the signals to be displayed by this viewer, and close the dialog.

Tip To display a viewer that has been attached, double click on the viewer of interest in the Viewers pane.

Creating a Multiple Axes Viewer

To create a viewer with more than one axes:



- 1 Display the Signal and Scope manager.
- 2 Select a viewer from the **Types** pane.
- 3 Click **Attach to Model**.
- 4 Click the #in pulldown, and select the total number of axes for the graph.
- 5 Navigate to the **Signals connected to Generator/Viewer** pane and select **Axes 1**.

Signals connected to Generator/Viewer	
Axes	Connected signals
1	no selection
2	no selection
3	no selection

6 Click the **Signal Selector** button to display the Signal Selector dialog.

For more information, see “The Signal Selector” on page 10-48.

7 Select the signals to add to this axis, and close the dialog.

8 Select the next axes, and repeat steps 6 and 7. Continue in this way until signals have been added to all axes.

Tip

- Click on the Scope Viewer icon to display the scope.
 - Run the simulation after adding the new signals to make them visible.
-

Adding Additional Signals to an Existing Viewer

To add signals to a viewer you have already created:

1 Display the Signal and Scope manager.

2 Navigate to the **Viewers** pane, and select the scope to which you will add signals.

3 Click the **Signal Selector** button to display the Signal Selector dialog.

For more information, see “The Signal Selector” on page 10-48.

4 Select the signals to add to this viewer, and close the dialog.

Tip Run the simulation after adding the new signals to make them visible.

Viewing Test Point Data

You can use 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.

For more information, see “Working with Test Points” on page 10-61 and Chapter 6, “Referencing a Model”.

Adding Custom Viewers and Generators

You can add custom signal viewers or generators so that they appear in the Signal and Scope Manager. The following procedure assumes that you have a custom viewer named `newviewer` that you want to add.

Note If the viewer is a compound viewer, such as a subsystem with multiple blocks, make the top-level subsystem atomic.

- 1** Open a new Simulink library.

For example, open the Simulink browser and select **File > New > Library**.

- 2** Save the library.

For example, save it as `newlib`.

- 3** In the MATLAB Command Window, set the library type.

For example, use this command to set the library type of `newviewer` to viewer,

```
set_param('newlib','LibraryType','SSMgrViewerLibrary')
```

To set library type for generators, use the type `'SSMgrGenLibrary'` as in this example:

```
set_param('newlib','LibraryType','SSMgrGenLibrary')
```

- 4** Set the display name of the library, as in this example:.

```
set_param('newlib','SSMgrDisplayString','My Custom Library')
```

- 5** Add the viewer or generator to the library.

- 6** Set the `iotype` of the viewer, as in this example:

```
set_param('newlib/newviewer','iotype','viewer')
```

- 7 Save the library newlib.

Select **File > Save**.

- 8 Using the MATLAB editor, create a file named `sl_customization.m`. In this file, enter a directive to incorporate the new library as a viewer library.

For example, to save newlib as a viewer library, add the following lines:

```
function sl_customization(cm)
cm.addSigScopeMgrViewerLibrary('newlib')
%end function
```

To add a library as a generator library, add a line like the following:

```
cm.addSigScopeMgrGeneratorLibrary('newlib')
```

- 9 Add a corresponding `cm.addSigScope` line for each viewer or generator library you want to add.
- 10 Save the `sl_customization.m` file on your MATLAB path. Edit this file to add new viewer or generator libraries.
- 11 To see the new custom libraries, restart MATLAB and start the Signal and Scope Manager.

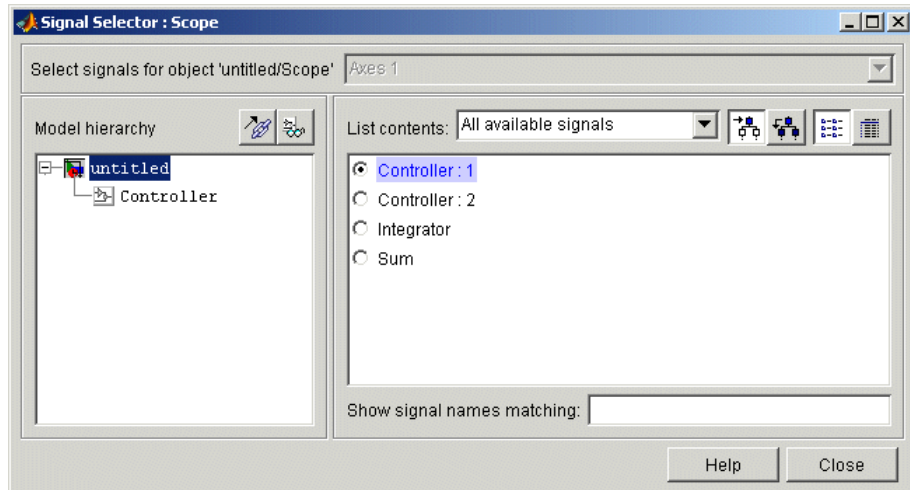
The Signal Selector

In this section...
“About the Signal Selector” on page 10-48
“Port/Axis Selector” on page 10-49
“Model Hierarchy” on page 10-50
“Inputs/Signals List” on page 10-50

About the Signal Selector

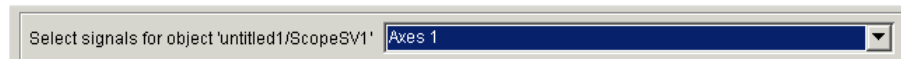
The Signal Selector allows you to connect a generator or viewer object (see “Introducing the Signal and Scope Manager” on page 10-37) or the Floating Scope to block inputs and outputs. It appears when you click the **Signal selection** button for a generator or viewer object in the Signal & Scope Manager or on the toolbar of the Floating Scope’s 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.



Port/Axis Selector

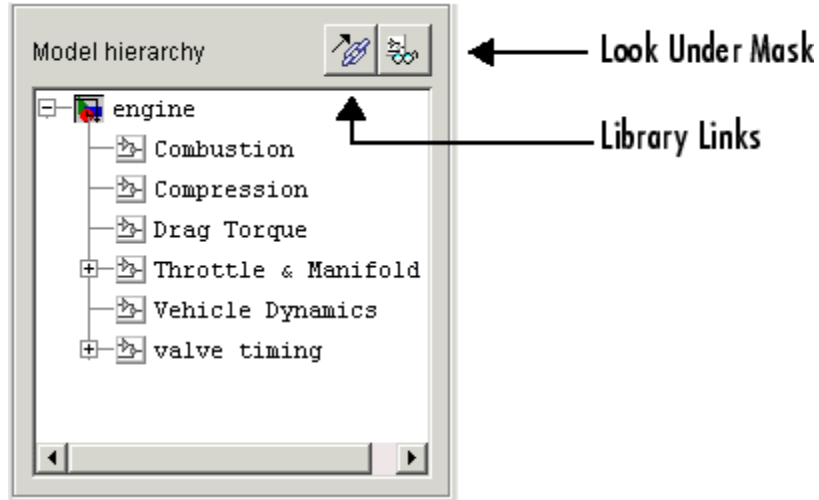
This list box allows you to select the owner output port (in the case of signal generators) or display axis (in the case of signal viewers) to which you want to connect blocks in your model.



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.

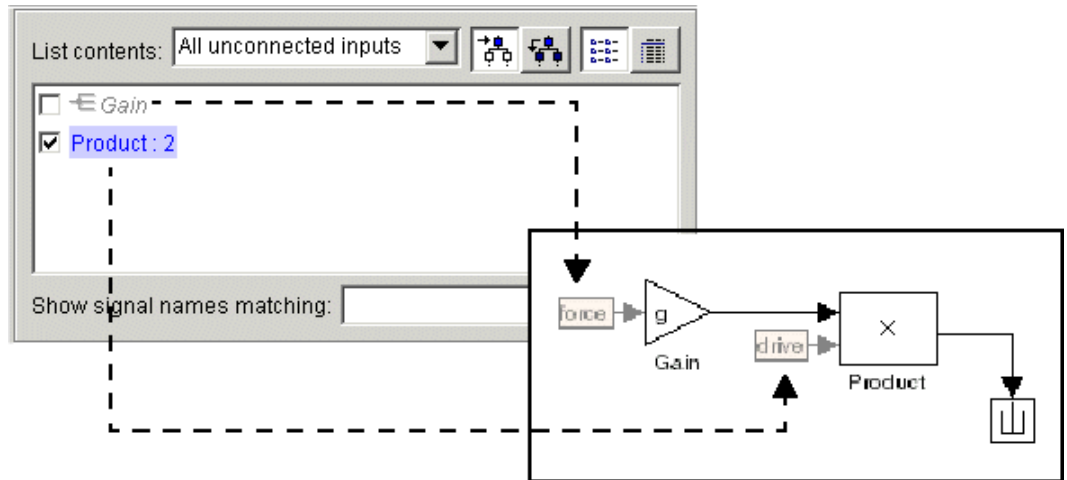


Selecting a subsystem causes the adjacent port list to display the ports available for connection in the selected subsystem. To display subsystems included as library links in your model, click the **Library Links** button at the top of the **Model hierarchy** control. To display subsystems contained 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 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.

Initializing Signals and Discrete States

In this section...

“About Initialization” on page 10-53

“Using Block Parameters to Initialize Signals and Discrete States” on page 10-54

“Using Signal Objects to Initialize Signals and Discrete States” on page 10-54

“Using Signal Objects to Tune Initial Values” on page 10-55

“Example: Using a Signal Object to Initialize a Subsystem Output” on page 10-57

“Initialization Behavior Summary for Signal Objects” on page 10-58

About Initialization

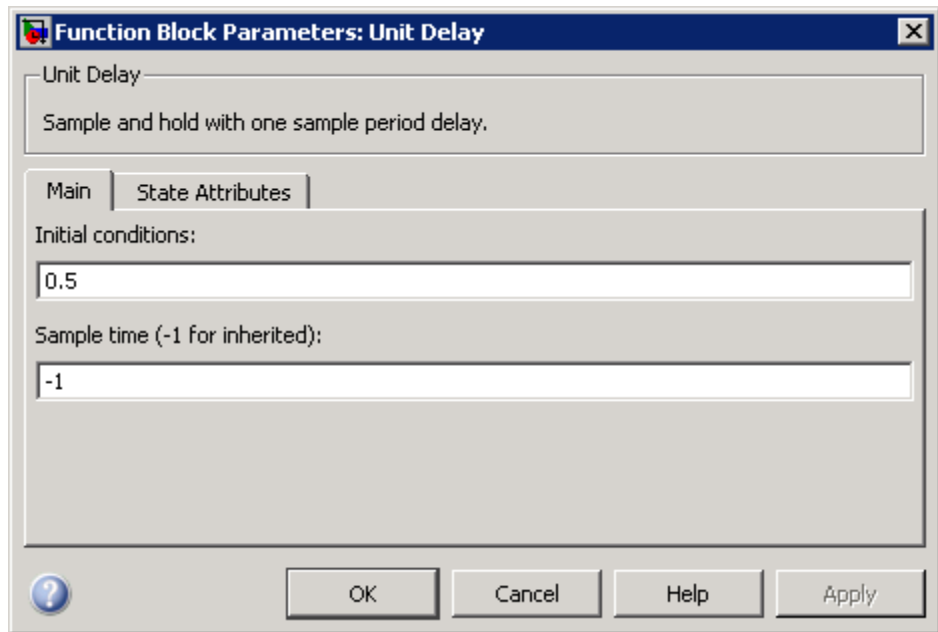
Simulink allows you to specify the initial values of signals and discrete states, i.e., the values of the signals and discrete states at the **Start time** of the simulation. You can use signal objects to specify the initial values of any signal or discrete state in a model. In addition, for some blocks, e.g., Outputport, 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 “Resolving Symbols” on page 4-75.

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 0.5.



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 “Working with Data Objects” on page 13-26.

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 `Simulink.Parameter` class in the Simulink online reference) 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.

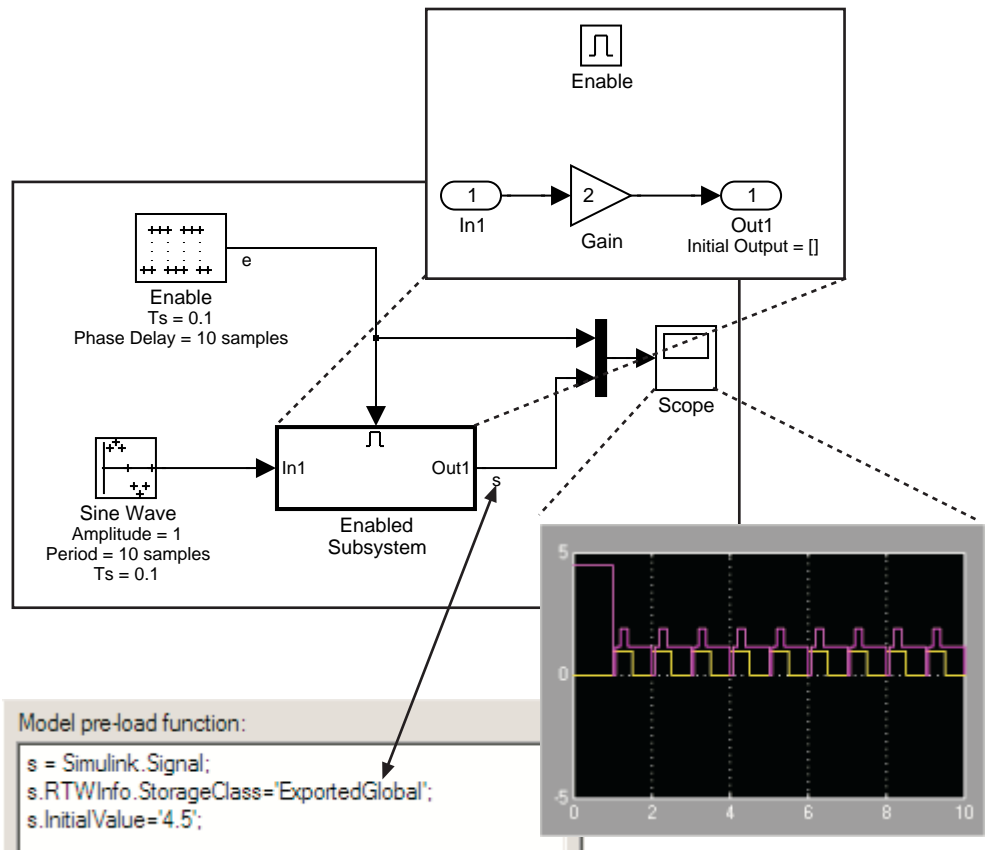
For example, suppose you want to tune the initial value of a Memory block state named `M1`. To do this, you might create a signal object named `M1`, set its storage class to 'ExportedGlobal', set its initial value to `K` (`M1.InitialValue='K'`), where `K` is a workspace variable in the MATLAB workspace, 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 2-9 and "Changing the Values of Block Parameters During Simulation" on page 7-23.

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.

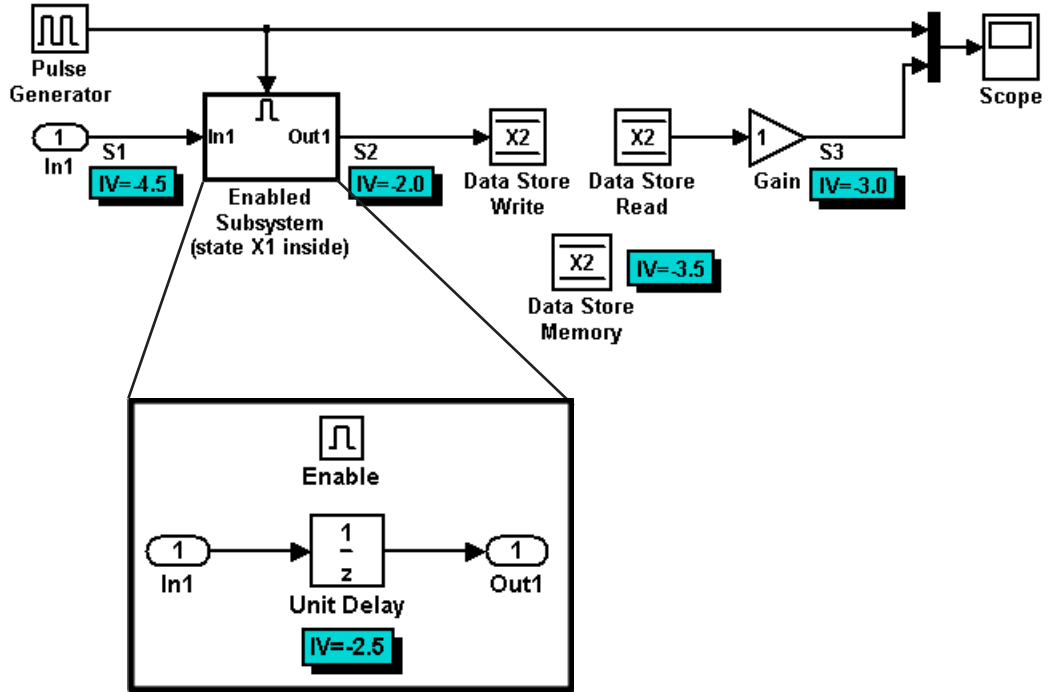


Signal s is initialized to 4.5. To avoid a consistency error, the initial value of the enabled subsystem's Output block must be $[]$ or 4.5.

If you need a signal object and its initial value setting to persist across Simulink sessions, see “Creating Persistent Data Objects” on page 13-36.

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 S1.InitialValue. • If you use the Data Import/Export pane of the Configuration Parameters dialog to specify values for the root inputs, the initial value is overwritten and may differ at each time step. Otherwise, the value remains constant.

Signal or Discrete State	Description	Behavior
X1	Unit Delay block — Block with a discrete state that has an initial condition	<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. • 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 “Using Work Vectors” in Writing S-Functions. • 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.

Signal or Discrete State	Description	Behavior
		<ul style="list-style-type: none">• Affects code generation only. For simulation, setting the initial value for S3 is irrelevant because the values are overwritten at the model's simulation start time.

Working with Test Points

In this section...

“About Test Points” on page 10-61

“Designating a Signal as a Test Point” on page 10-61

“Displaying Test Point Indicators” on page 10-62

About Test Points

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 2-35).

Test points are primarily intended for use when generating code from a model with Real-Time Workshop. For more information about test points in the context of code generation, see “Signals with Test Points” in the Real-Time Workshop documentation.

Designating a Signal as a Test Point

To specify that a signal whose storage class is Auto is a test point, open the signal’s **Signal Properties** dialog box and select **Logging and accessibility** > **Test point**. The signal is now a test point. Selecting or clearing this option has no effect unless the signal’s storage class is Auto.

Any signal whose storage class is not Auto is automatically a test point, regardless of the setting of **Signal Properties** > **Logging and accessibility** > **Test point**. You can specify a non-Auto signal storage class in three ways:

- Set **Signal Properties > Real-time Workshop > Storage class** to anything other than Auto.
- Resolve the signal to a base workspace Simulink.Signal object that specifies a storage class other than Auto.
- Embed a signal object that specifies a storage class other than Auto on the port where the signal originates.

Using a base workspace signal object to specify that a signal is a test point can be convenient, because it allows you to control testpointing without having to change the model itself. Assigning storage class `SimulinkGlobal` has exactly the same effect as assigning storage class `Auto` and selecting **Signal Properties > Logging and accessibility > Test point**.

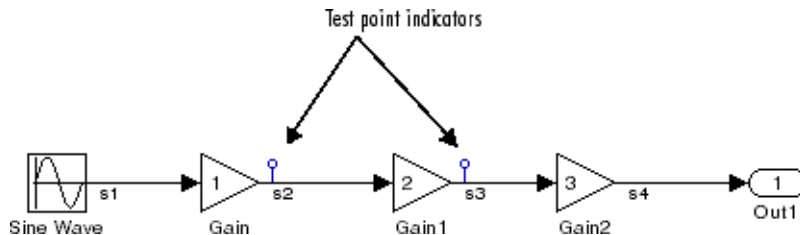
See “Signal Properties Dialog Box” for more information about specifying signal properties.

Test Points in Referenced Models that Use Library Blocks

If you set the test point property of a signal in a library block that is referenced by a model that is itself referenced by another model, you must update the referenced model containing the test pointed signal by opening and saving it. Otherwise, Simulink will be unable to log or display the referenced signal.

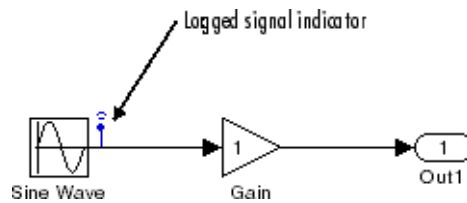
Displaying Test Point Indicators

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 Chapter 15, “Importing and Exporting Data” 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, select or clear **Port/Signal Displays > Testpoint/Logging Indicators** from the Simulink **Format** menu.

Displaying Signal Properties

In this section...

“Port/Signal Displays Menu” on page 10-64

“Port Data Types” on page 10-65

“Signal Dimensions” on page 10-65

“Signal Resolution Indicators” on page 10-66

“Wide Nonscalar Lines” on page 10-67

Port/Signal Displays Menu

The **Format >Port/Signal Displays** submenu of the model window offers the following options for displaying signal properties on the block diagram:

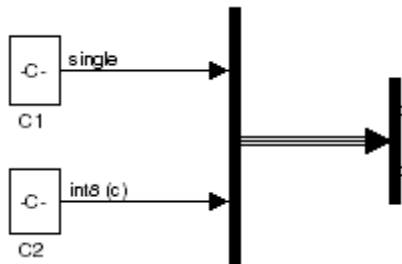
- Linearization Indicators
- Port Data Types (See “Port Data Types” on page 10-65)
- Signal Dimensions (See “Signal Dimensions” on page 10-65)
- Storage Class
- Testpoint/Logging Indicators
- Signal Resolution Indicators (See “Signal Resolution Indicators” on page 10-66)
- Viewer Indicators
- Wide Nonscalar Lines (See “Wide Nonscalar Lines” on page 10-67)

In addition, you can display sample time information. If you first select **Format > Sample Time Display**, 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

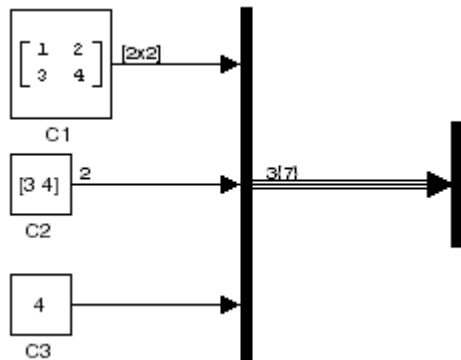
Displays the data type of a signal next to the output port that emits the signal.



The notation (c) following the data type of a signal indicates that the signal is complex.

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 $\{M\}$.

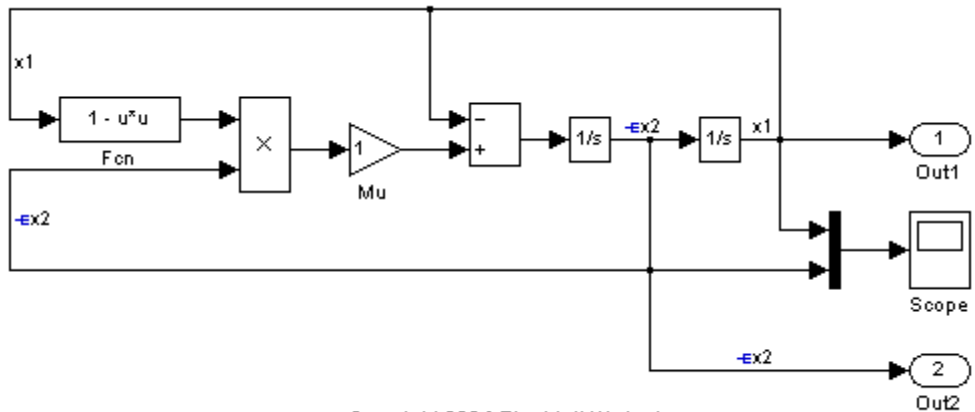
Signal Resolution Indicators

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 x_2 must resolve to a signal object, so a signal resolution icon appears to the left of the signal name in each label:



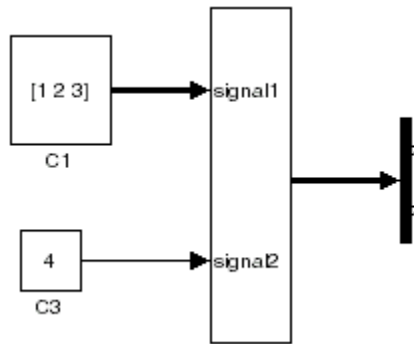
Copyright 2004 The MathWorks, Inc.

To suppress the display of signal resolution icons, in the model window deselect **Format > Port/Signal Displays > Signal Resolution Indicators**, which is selected by default. To restore signal resolution icons, reselect **Signal Resolution Indicators**. Individual signals cannot be set to show or hide signal resolution indicators independently of the setting for the whole model. For additional information, see:

- “Resolving Symbols” on page 4-75
- “Initializing Signals and Discrete States” on page 10-53
- Simulink.Signal

Wide Nonscalar Lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.



See Chapter 12, “Using Composite Signals” for more information about vector and matrix signals.

Working with Signal Groups

In this section...

“About the Signal Groups” on page 10-69

“Creating a Signal Group Set” on page 10-69

“Signal Builder Dialog Box” on page 10-71

“Editing Signal Groups” on page 10-73

“Editing Signals” on page 10-73

“Editing Waveforms” on page 10-76

“Signal Builder Time Range” on page 10-82

“Exporting Signal Group Data” on page 10-83

“Printing, Exporting, and Copying Waveforms” on page 10-83

“Simulating with Signal Groups” on page 10-84

“Simulation Options Dialog Box” on page 10-85

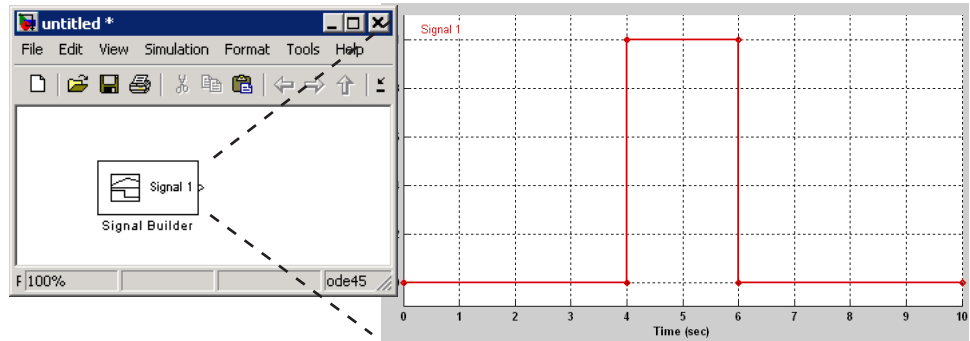
About the Signal Groups

The Signal Builder block allows you to create 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 used in conjunction with Simulink’s Assertion blocks and Simulink Verification and Validation’s Model Coverage Tool. For a description of the Model Coverage Tool, see the “Simulink Verification and Validation User’s Guide” on The MathWorks Web site (www.mathworks.com).

Creating a Signal Group Set

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's signal editor (see “Signal Builder Dialog Box” on page 10-71) 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.

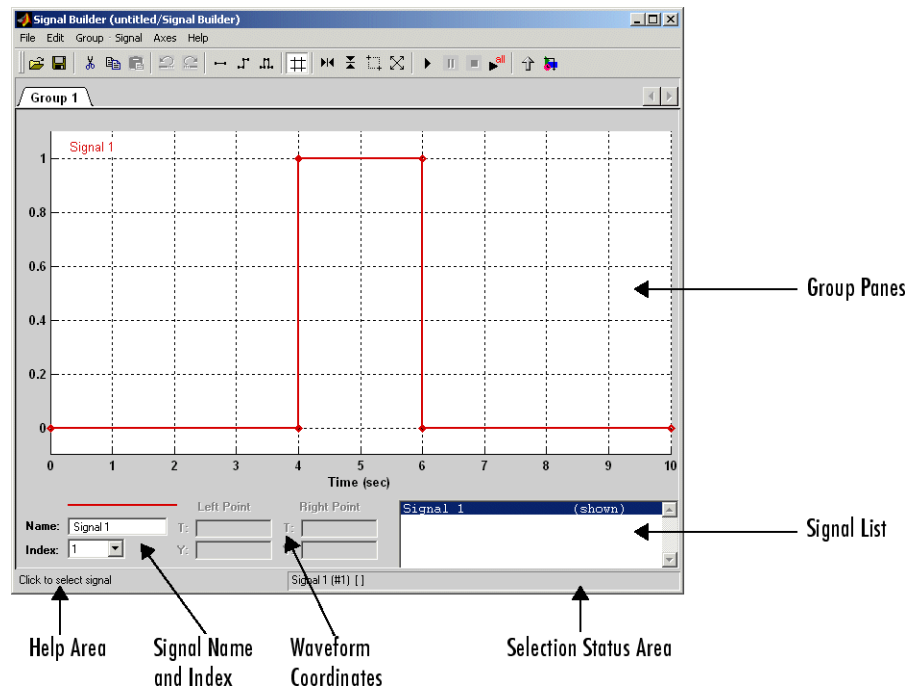
You can create as many Signal Builder blocks as you like in a model, each representing a distinct set of interchangeable groups of signal sources. When there are multiple signals in a group, signals might have different end times. However, Signal Builder block requires the end times of signals within 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 10-84 for information on using signal groups in a model.

Signal Builder Dialog Box

The Signal Builder block's dialog box 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 dialog box, double-click the block. The **Signal Builder** dialog box appears.



The **Signal Builder** dialog box allows you to create and modify signal groups represented by a Signal Builder block. The **Signal Builder** dialog box includes the following controls.

Group Panes

Displays the set of interchangeable signal source groups represented by the block. The pane for each group displays an editable representation of each waveform that the group contains. The name of the group appears on the

pane's tab. Only one pane is visible at a time. To display a group that is invisible, select the tab that contains its name. The block outputs the group of signals whose pane is currently visible.

Signal Axes

The signals appear on separate axes that share a common time range (see “Signal Builder Time Range” on page 10-82). This allows you to easily 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's **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 10-73) of the signals that belong to the currently selected signal group. Clicking an entry in the list selects the signal. Double-clicking a signal's 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 10-76).

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 10-75).

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’s Index” on page 10-75).

Help Area

Displays context-sensitive tips on using **Signal Builder** dialog box features.

Editing Signal Groups

The Signal Builder dialog box allows you to create, rename, move, and delete signal groups from the set of groups represented by a Signal Builder block.

Creating and Deleting Signal Groups

To create a signal group, you must copy an existing signal group and then modify it to suit your needs. To copy an existing signal group, select its tab and then select **Copy** from the Signal Builder’s **Group** menu. To delete a group, select its tab and then select **Delete** from the **Group** menu.

Renaming Signal Groups

To rename a signal group, select the group’s tab and then select **Rename** from the Signal Builder’s **Group** menu. A dialog box appears. 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, select the pane and then select **Move Right** from the Signal Builder’s **Group** menu to move the group lower in the stack or **Move Left** to move the pane higher in the stack.

Editing Signals

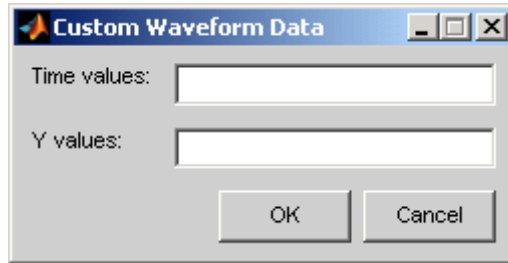
The **Signal Builder** dialog box allows you to create, cut and paste, hide, and delete signals from signal groups.

Creating Signals

To create a signal in the currently selected signal group, select **New** from the Signal Builder’s **Signal** menu. A menu of waveforms appears. The menu includes a set of standard waveforms (**Constant**, **Step**, etc.) and a **Custom**

waveform option. Select one of the waveforms. If you select a standard waveform, the Signal Builder adds a signal having that waveform to the currently selected group.

If you select **Custom**, a custom waveform dialog box appears.



The dialog box allows you to specify a custom piecewise linear waveform to be added to the groups defined by the Signal Builder block. Enter the custom waveform's time coordinates in the **Time values** field and the corresponding signal amplitudes in the **Y values** field. The entries in either field can be any MATLAB expression that evaluates to a vector. The resulting vectors must be of equal length. Click **OK**. The Signal Builder adds a signal having the specified shape to the currently selected group.

Copying and Pasting Signals

To copy a signal from one group and paste it into another group as a new signal:

- 1 Select the signal you want to copy.
- 2 Select **Copy** from the Signal Builder's **Edit** menu or click the corresponding button from the toolbar.
- 3 Select the group into which you want to paste the signal.
- 4 Select **Paste** from the Signal Builder's **Edit** menu or click the corresponding button on the toolbar.

To copy a signal from one axes and paste it into another axes to replace its signal:

- 1 Select the signal you want to copy.
- 2 Select **Copy** from the Signal Builder's **Edit** menu or click the corresponding button from the toolbar.
- 3 Select the signal on the axes that you want to replace.
- 4 Select **Paste** from the Signal Builder's **Edit** menu or click the corresponding button on the toolbar.

Deleting Signals

To delete a signal, select the signal and choose **Delete** or **Cut** from the Signal Builder's **Edit** menu. As a result, Simulink deletes the signal from the current group. Since each signal group must contain the same number of signals, Simulink also deletes all signals sharing the same index in the other groups.

Renaming a Signal

To rename a signal, select the signal and choose **Rename** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field that displays the signal's current name. Edit or replace the current name with a new name. Click **OK**. Or edit the signal's name in the **Name** field in the lower-left corner of the **Signal Builder** dialog box.

Changing a Signal's Index

To change a signal's index, select the signal and choose **Change Index** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field containing the signal's existing index. Edit the field 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** dialog box displays the group's waveforms in the group's tabbed pane. To hide a waveform, select the waveform and then select **Hide** from the Signal Builder's **Signal** menu. To redisplay a hidden waveform, select the signal's **Group** pane, then select **Show** from the Signal Builder's **Signal** menu to display a menu of hidden signals. Select the signal from the menu. Alternatively, you can hide and redisplay a hidden waveform

by double-clicking its name in the Signal Builder's signal list (see "Signal List" on page 10-72).

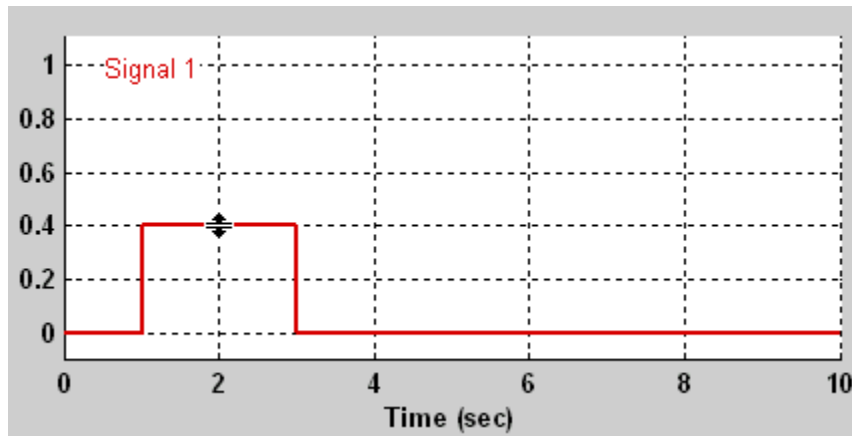
Editing Waveforms

The **Signal Builder** dialog box allows you to change the shape, color, and line style and thickness of the waveforms output by a group.

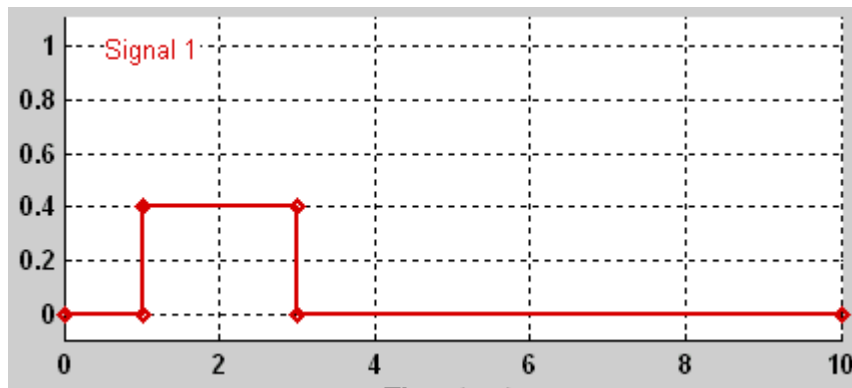
Reshaping a Waveform

The **Signal Builder** dialog box 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.

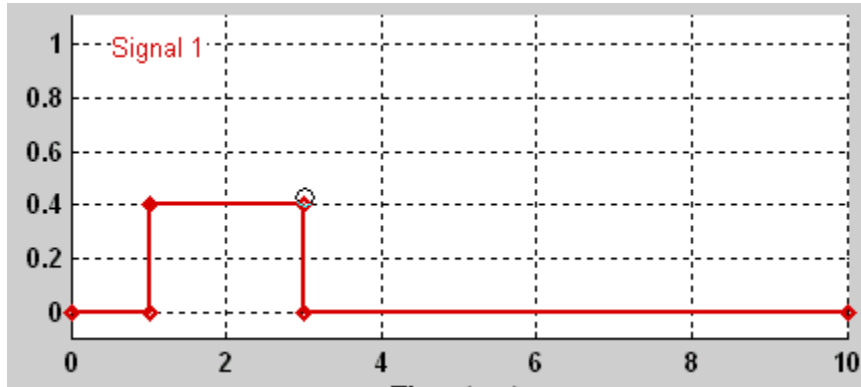


The Signal Builder displays the waveform's points to indicate that the waveform is selected.

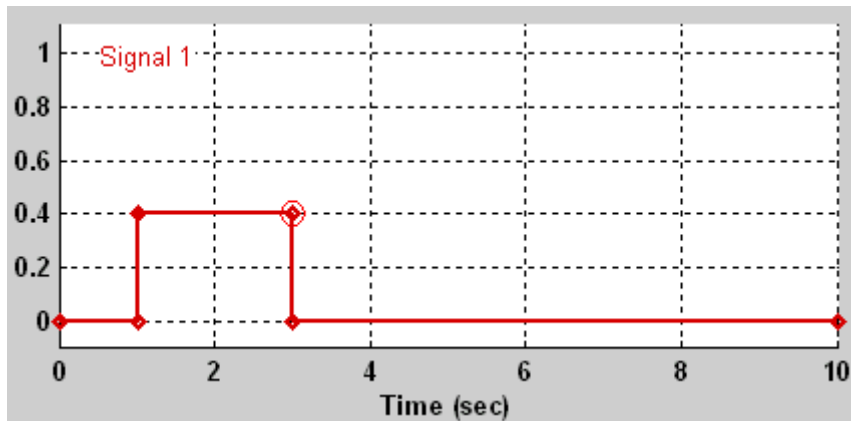


To deselect a waveform, left-click any point on the waveform's axis that is not on the waveform itself or press the **Esc** key.

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.

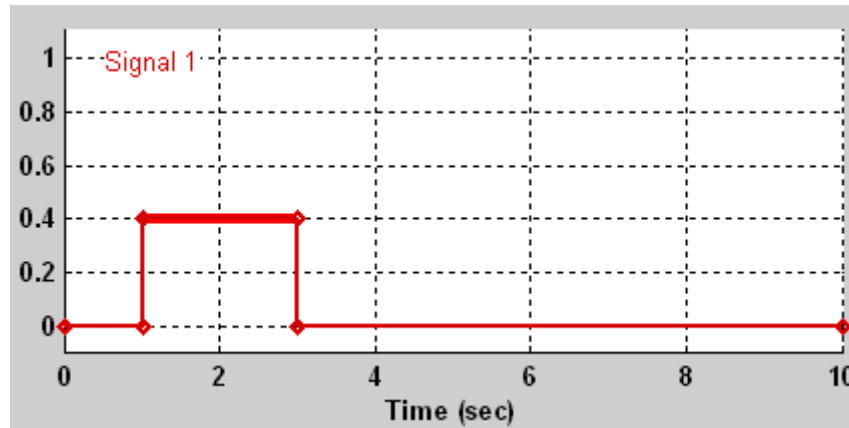


Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate that it is selected.



To deselect the point, press the **Esc** key.

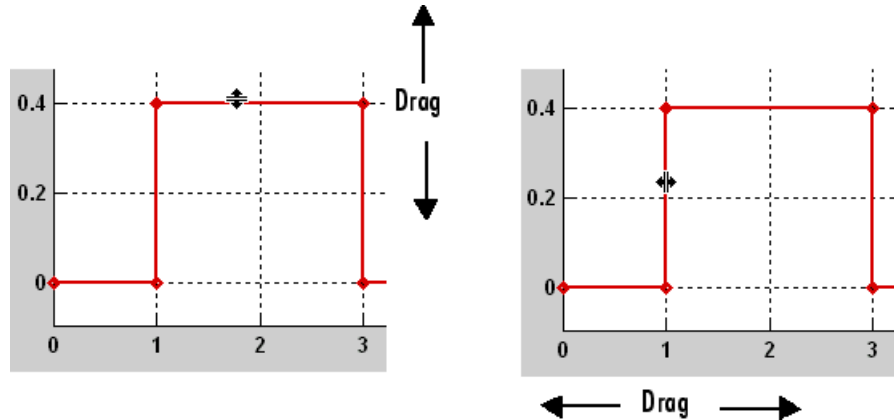
Selecting Segments. To select a line segment, first select the waveform that contains it. Then left-click the segment. The Signal Builder thickens the segment to indicate that it is selected.



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 10-80) 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's points to the nearest point or points on the grid, respectively. The Signal Builder's **Axes** menu allows you to specify the grid's 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** dialog box. 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** dialog box. 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's **Signal** menu. The Signal Builder displays the MATLAB color chooser. Choose a new color for the waveform. Click **OK**.

Changing a Waveform's 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's **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's current thickness. Edit the thickness value and click **OK**.

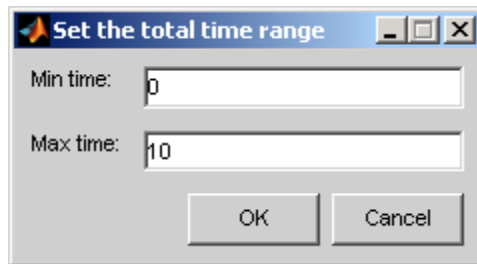
Signal Builder Time Range

The Signal Builder's 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's time range (see "Changing a Signal Builder's Time Range" on page 10-82).

If the simulation starts before the start time of a block's time range, the block extrapolates its initial output from its first two defined outputs. If the simulation runs beyond the block's time range, the block by default outputs values extrapolated from the last defined signal values for the remainder of the simulation. The Signal Builder's **Simulation Options** dialog box allows you to specify other final output options (see "Signal values after final time" on page 10-85 for more information).

Changing a Signal Builder's Time Range

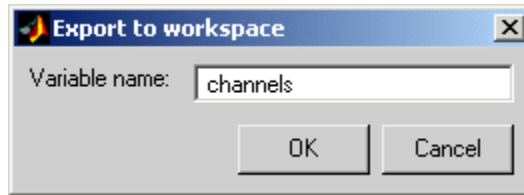
To change the time range, select **Change Time Range** from the Signal Builder's **Axes** menu. A dialog box appears.



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's signal groups to the MATLAB workspace, select **Export to Workspace** from the block's **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's 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's `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's `allXData` and `allYData` fields.

Printing, Exporting, and Copying Waveforms

The **Signal Builder** dialog box 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's **File** menu.

You can also export the waveforms to other destinations by using the **Export** option from the block's **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's **Edit** menu.

Simulating with Signal Groups

You can use standard simulation commands to run models containing Signal Builder blocks or you can use the Signal Builder's **Run** or **Run all** command (see "Running All Signal Groups" on page 10-84).

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** dialog box for that block, if the dialog box is open, otherwise the group that was selected when the dialog box was last closed. To activate a group, open the group's **Signal Builder** dialog box and select the group.

Running Different Signal Groups in Succession

The Signal Builder's 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's dialog box.

Running All Signal Groups

To run all the signal groups defined by a Signal Builder block, open the block's dialog box and click the **Run all** button



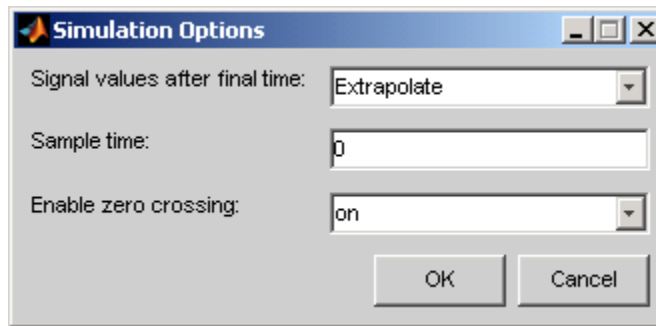
from the Signal Builder's toolbar. The **Run all** command runs a series of simulations, one for each signal group defined by the block. If you installed Simulink Verification and Validation on your system and are using the Model Coverage Tool, the **Run all** command configures the tool to collect and 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 Signal Builder's **File** menu. The dialog box appears.



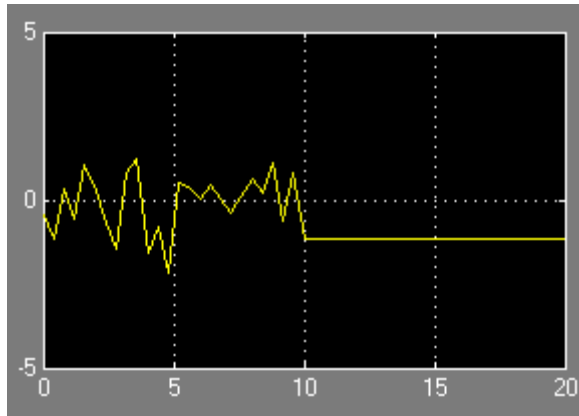
The dialog box allows you to specify the following options.

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

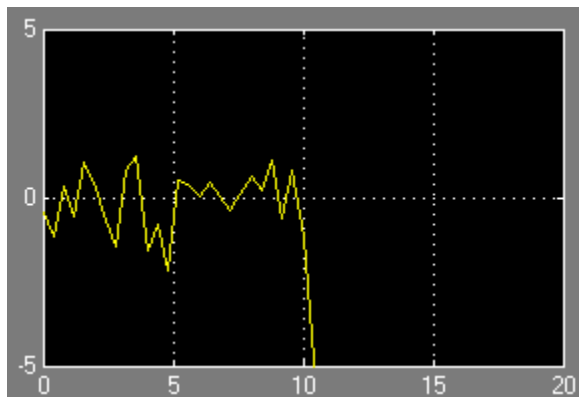
- Hold final value

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.



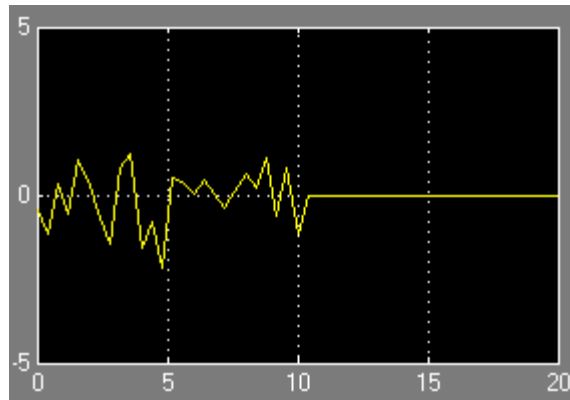
- **Extrapolate**

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.



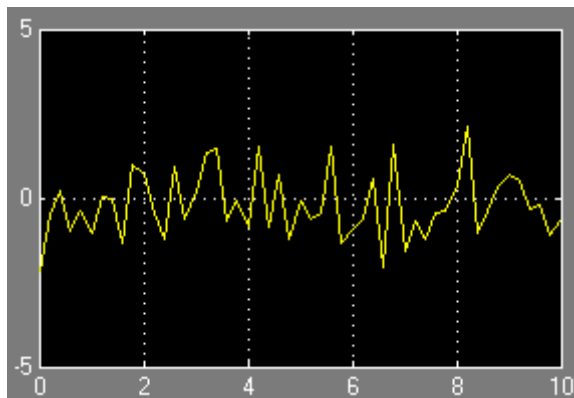
- **Set to zero**

Selecting this option causes the Signal Builder block to output zero for the remainder of the simulation.

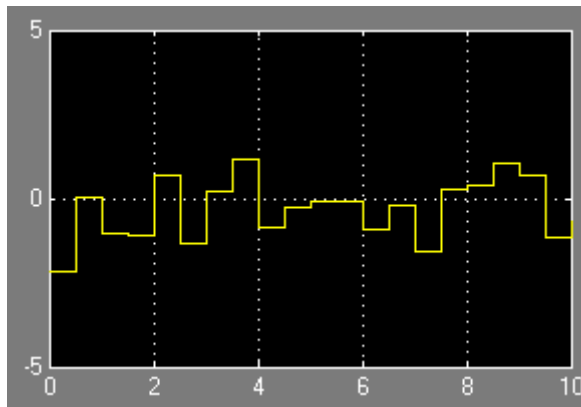


Sample time

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 2-24.

Working with Variable-Size Signals

- “Variable-Size Signal Basics” on page 11-2
- “Simulink Models Using Variable-Size Signals” on page 11-6
- “S-Functions Using Variable-Size Signals” on page 11-19
- “Simulink Block Support for Variable-Size Signals” on page 11-22
- “Variable-Size Signal Limitations” on page 11-29

Variable-Size Signal Basics

In this section...
“About Variable-Size Signals” on page 11-2
“Creating Variable-Size Signals” on page 11-2
“How Variable-Size Signals Propagate” on page 11-3
“Empty Signals” on page 11-4
“Subsystem Initialization of Variable-Size Signals” on page 11-4

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 10-2 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:

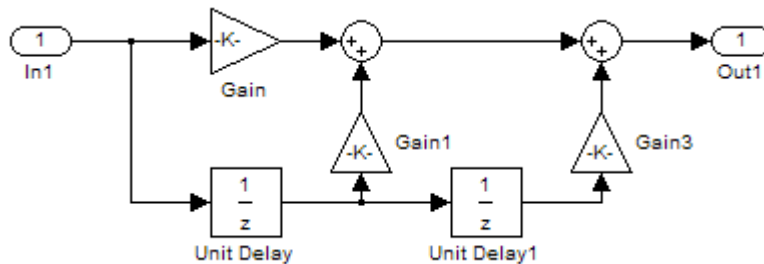
- **At every step of model execution.**

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.

“Demo of Mode-Dependent Variable-Size Signals” on page 11-13 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.

Simulink Models Using Variable-Size Signals

In this section...
“Demo of Variable-Size Signal Generation and Operations” on page 11-6
“Demo of Variable-Size Signal Length Adaptation” on page 11-10
“Demo of Mode-Dependent Variable-Size Signals” on page 11-13

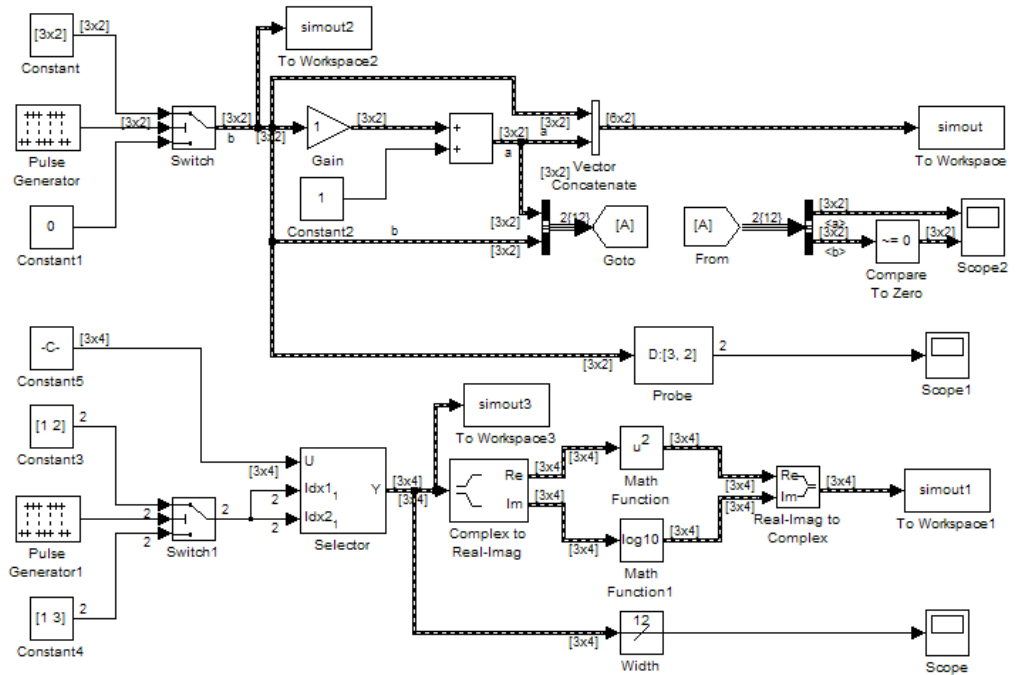
Demo of Variable-Size Signal Generation and Operations

This demo 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 11-22.

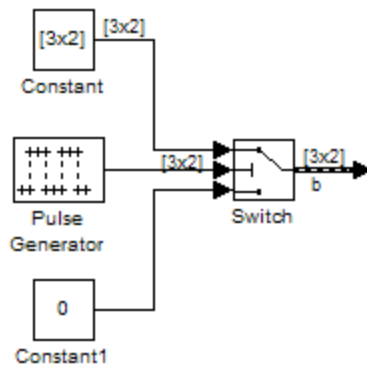
To open the demo model, in the MATLAB Command Window, type:

```
sldemo_varsize_basic
```



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 3x2. When you select the **Allow different data input sizes** parameter on the Switch block, Simulink does not expand the scalar value from the Constant1 block.

Saving Variable-Size Signal Data

The model saves output from the Switch block using the ToWorkspace2 block with 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 =
```

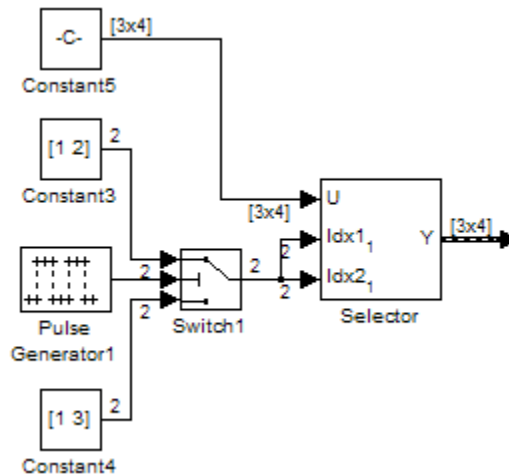
```

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. To view the variable-size signal values, enter:

```
simout3.signals.values
```

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 demonstrates 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.

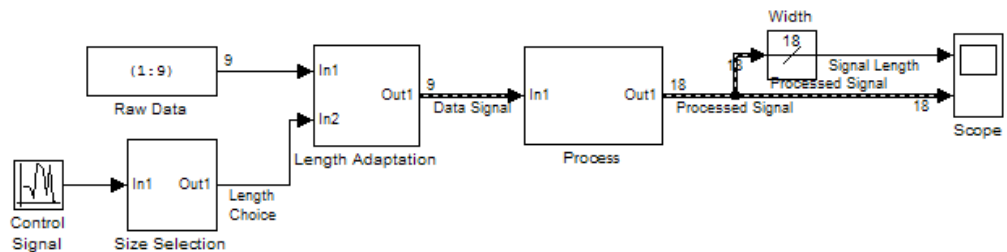
Demo of Variable-Size Signal Length Adaptation

This demo 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 is connects to a processing block, where blocks that support variable-size signals operate on it. An Embedded 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 11-22.

To open the demo model, in the MATLAB® Command Window, type:

```
sldemo_varsize_datalelengthadapt
```



Creating a Variable-Size Signal by Adapting the Length of a Data Signal

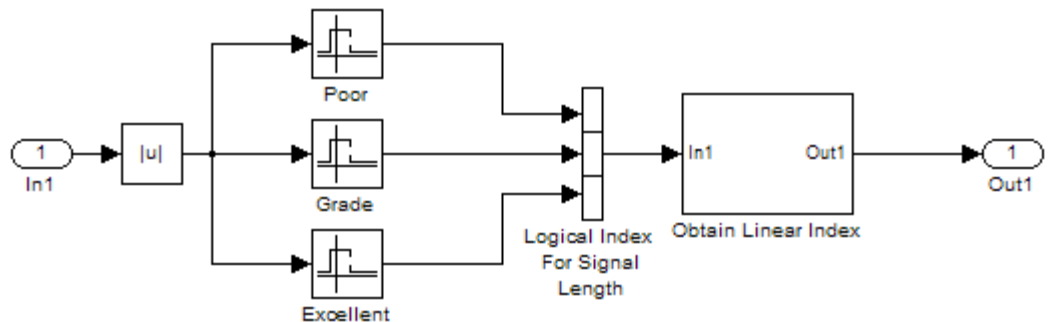
This model generates a data signal and converts the signal to a variable-size signal. The size of the signal depends on the value of a control signal. The raw data signal is a column vector with values from 1 to 9.

```
[1:9].'
```

```
ans =
```

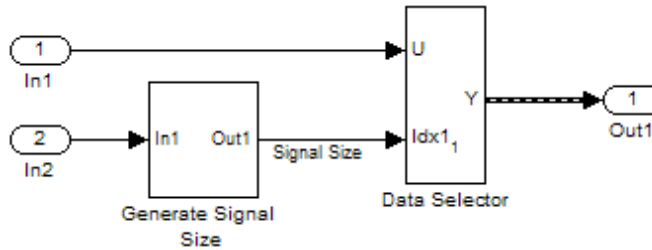
```
1
2
3
4
5
6
7
8
9
```

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.



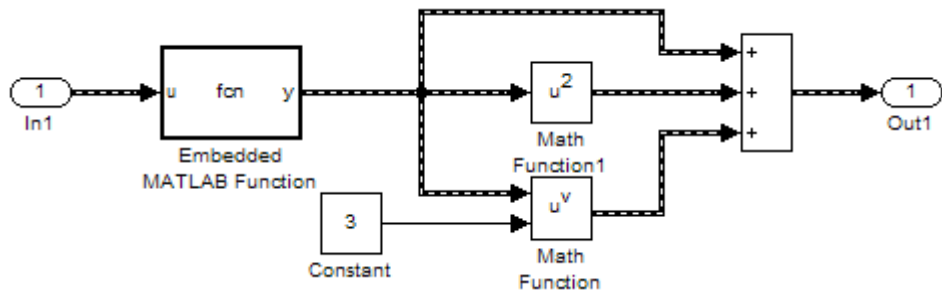
In the Length Adaptation subsystem, the Signal Size subsystem generates an index based on the quality value from the Size Selection subsystem (In2). The

Data Selector block uses the starting and ending indices to adapt the length of the data signal (In1) and output a variable-size signal.



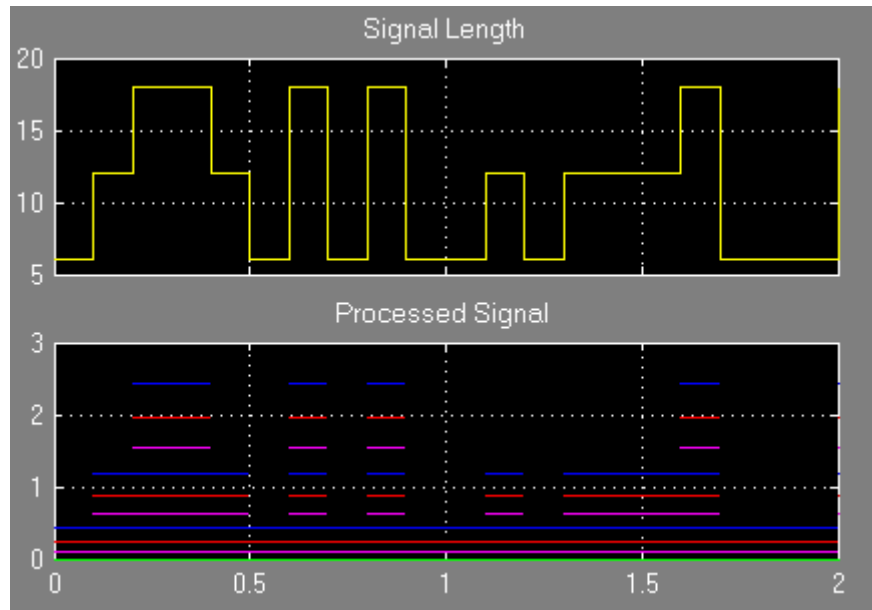
Processing a Variable-Size Signal

The center section of the model processes the variable-size signal. The Embedded 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 demonstrate 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.



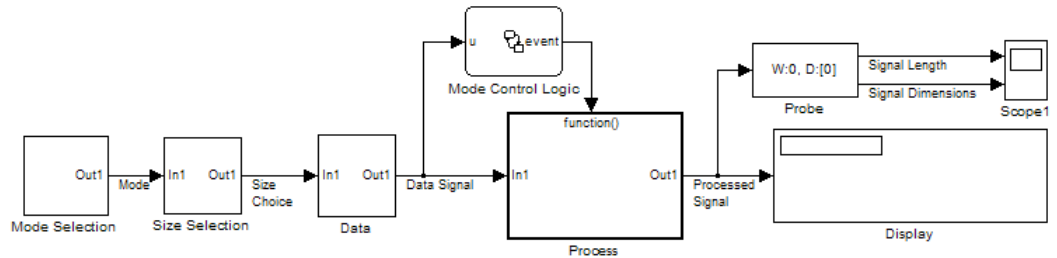
Demo of Mode-Dependent Variable-Size Signals

This demo 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.

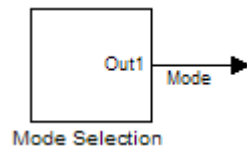
To open the model, In the MATLAB Command Window, type:

```
sldemo_varsize_multimode
```

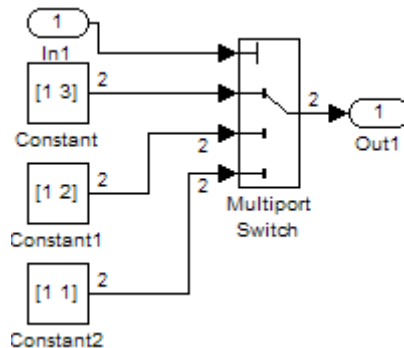


Creating a Variable-Size Signal Based on Mode

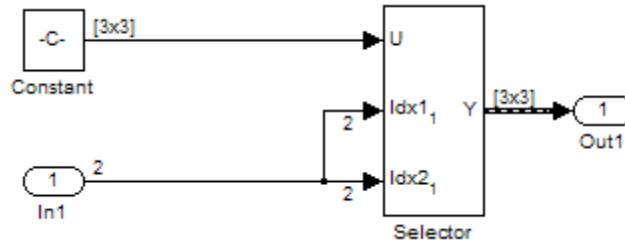
The Mode Selection subsystem determines the mode for processing a data signal and outputs a mode value (1, 2, or 3). This value helps to select the length of the data signal using the Size Selection and Data subsystems.



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) =
```

```

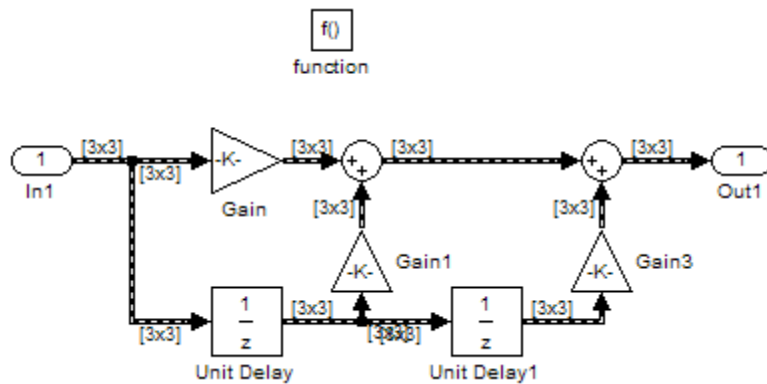
1     4     NaN
2     5     NaN
NaN     NaN     NaN
```

```
ans(:,:,3) =
    1     4     7
    2     5     8
    3     6     9
```

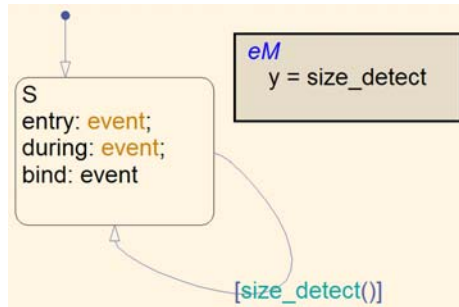
Processing a Variable-Size Signal with a Conditionally Executed Subsystem

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 11-3.

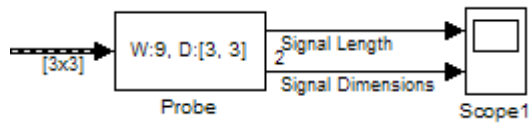


The Stateflow chart determines if there is a change in the size of the signal. The function `size_detect` calculates the width of the variable-size signal at each time step, and compares the current width to the previous width. If there is a change in signal size, the chart outputs a function-call output event that resets and repropagates the signal sizes within the Process subsystem.

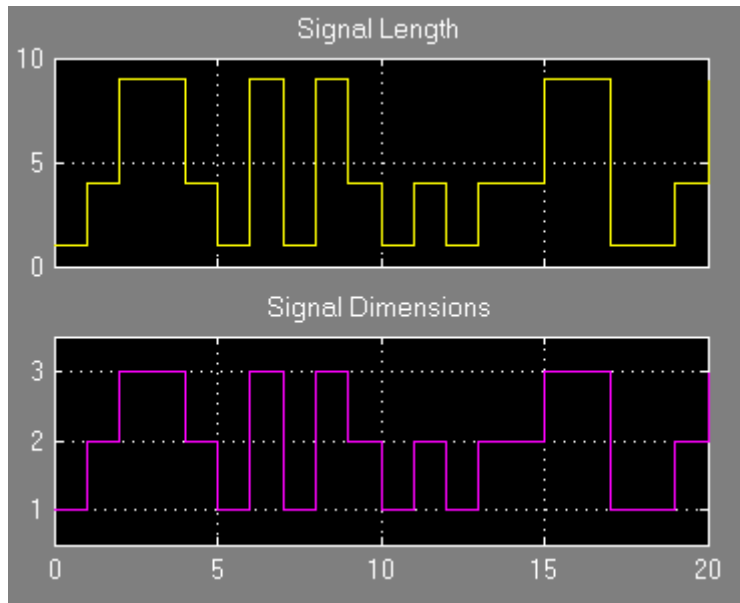


Visualizing Data

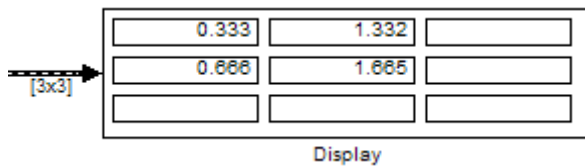
Use the Probe block to visualize signal size and signal dimension.



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.



S-Functions Using Variable-Size Signals

In this section...

“Demo of Level-2 M-File S-Function with Variable-Size Signals” on page 11-19

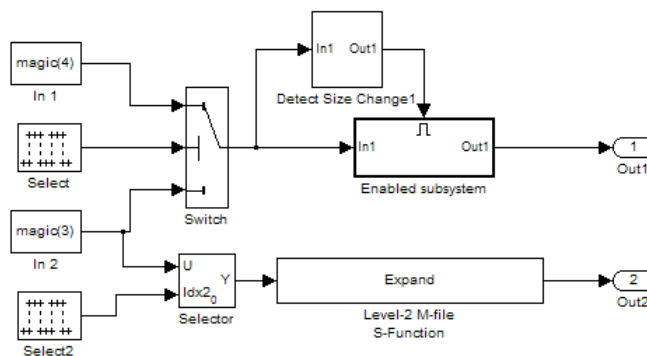
“Demo of C-File S-Function with Variable-Size Signals” on page 11-20

Demo of Level-2 M-File S-Function with Variable-Size Signals

Both Level-2 M-File S-Functions and C-File 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 demo model, in the MATLAB Command Window, type:

```
msfcdemo_varsize
```



```
matlabroot\toolbox\simulink\simdemos\simfeatures\msfcn_varsize_expand.m
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_dmsfcn_varsize_expand.tlc
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\msfcn_varsize_holdStatesUntilReset.m
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_dmsfcn_varsize_holdStatesUntilReset.tlc
```

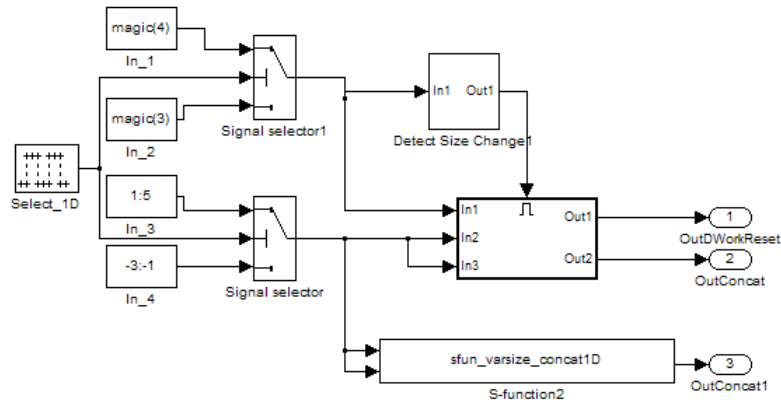
The Enabled subsystem includes a Level-2 M-File S-Function which demonstrates how to implement a block that holds its states until reset. Because this block contains states and delays the input signal, the input size can change only when a reset occurs.

The Expand block is a Level-2 M-File 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.

Demo of C-File S-Function with Variable-Size Signals

To open this demo model, in the MATLAB Command Window, type:

```
sfcndemo_varsize
```



```
matlabroot\toolbox\simulink\simdemos\simfeatures\src\sfun_varsize_concat1D.c
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_tlc_sfun_varsize_concat1D.tlc
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\src\sfun_varsize_holdStatesUntilReset.c
```

```
matlabroot\toolbox\simulink\simdemos\simfeatures\tlc_tlc_sfun_varsize_holdStatesUntilReset.tlc
```

The enabled subsystems have two S-Functions:

- `sfun_varsize_holdStatesUntilReset` is a C-File 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-File S-function that implements the concatenation of two unoriented vectors. You can use this function within an enabled subsystem by itself.

Simulink Block Support for Variable-Size Signals

In this section...
“Discrete” on page 11-22
“Logic and Bit Operations” on page 11-22
“Math Operations” on page 11-22
“Subsystem Blocks” on page 11-23
“Conditionally Executed Subsystem Blocks” on page 11-24
“Signal Attributes” on page 11-25
“Signal Routing” on page 11-25
“Switching Blocks” on page 11-25
“Source Blocks” on page 11-26
“Sink Blocks” on page 11-26
“User-Defined Function Blocks” on page 11-27
“Signal Processing Blockset” on page 11-27
“Video and Image Processing Blockset” on page 11-28

Discrete

- Unit Delay

Logic and Bit Operations

- Logical Operator
- Relational Operator – isInf, isNaN, isFinite

Math Operations

- Abs
- Add
- Assignment

- Complex to Magnitude-Angle
- Complex to Real-Imag
- Divide
- Dot Product
- Gain
- Magnitude-Angle to Complex
- Math Function
- Matrix Concatenate
- MinMax
- Permute Dimensions
- Product
- Product of Elements
- Real-Imag to Complex
- Reshape
- Subtract
- Sum
- Sum of Elements
- Trigonometric Function
- Vector Concatenate

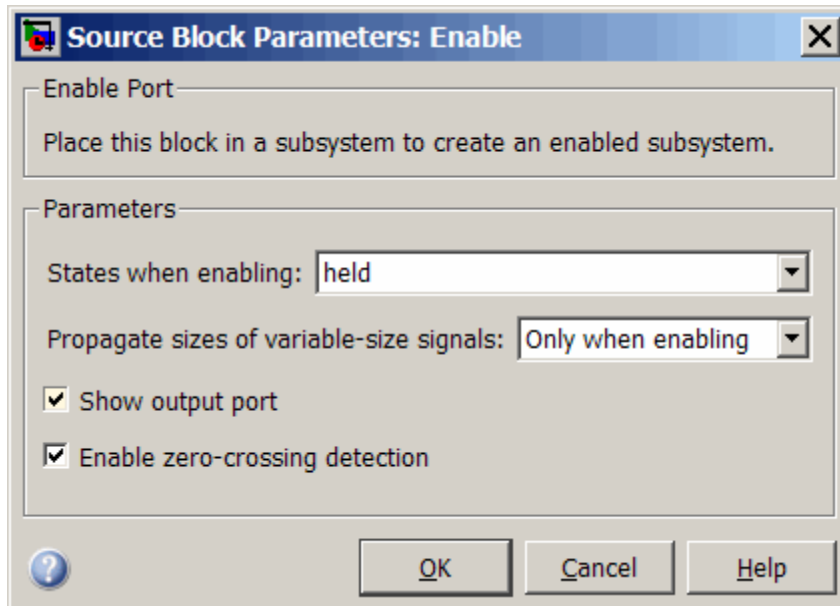
Subsystem Blocks

- Atomic Subsystem
- CodeReuse Subsystem
- Enabled Subsystem
- For Iterator Subsystem
- Function-Call Subsystem
- If

- If Action Subsystem
- Model
- Subsystem
- Triggered Subsystem

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

Signal Attributes

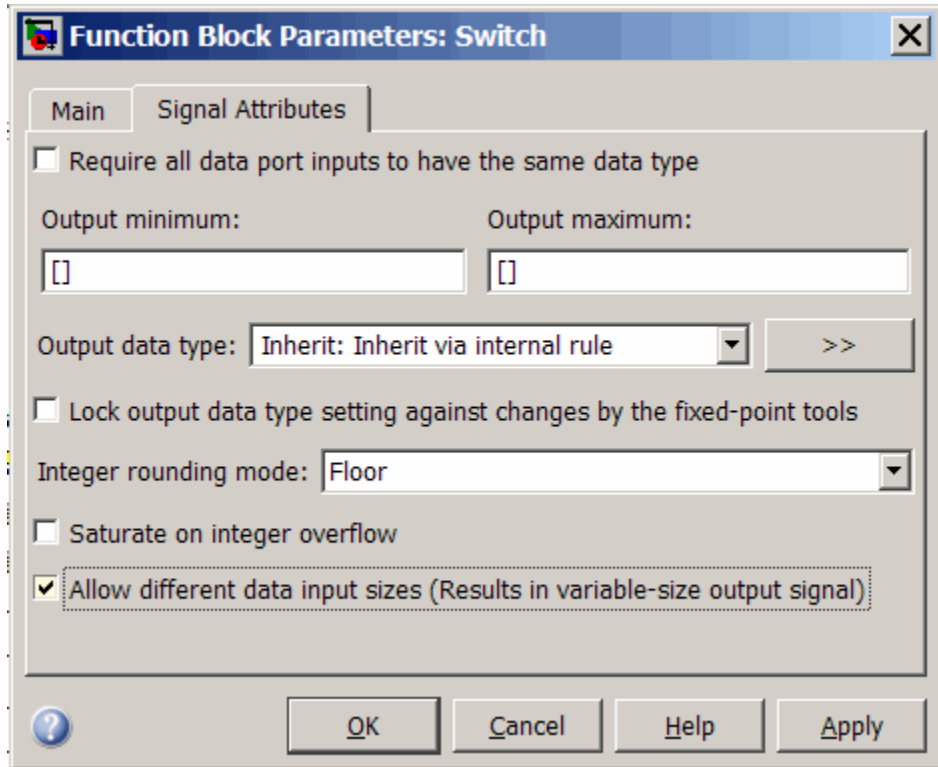
- Data Type Conversion
- Data Type Conversion Inherited
- Probe
- Rate Transition
- Signal Conversion
- Signal Specification — **Variable-size signal** values set to Inherit, No, Yes
- Width

Signal Routing

- Bus Assignment
- Bus Creator
- Bus Selector
- From
- Goto
- Selector

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

Source Blocks

You can use the following blocks to generate variable-size signals:

- From Workspace
- Inport (root inport) – **Variable-size signal** values set to Inherit, No, Yes

Sink Blocks

You can use the following blocks to accept variable-size signals:

- Display
- Outport (root outport) — **Variable-size signal** values set to Inherit, No, Yes
- To Workspace
- Scope

User-Defined Function Blocks

- Level-2 M-File S-Function
- C-MEX S-Function

Signal Processing Blockset

- Array-Vector Add
- Array-Vector Divide
- Array-Vector Multiply
- Array-Vector Subtract
- Difference
- FFT
- IFFT
- Inherit Complexity
- Matrix Product
- Matrix Sum
- Maximum
- Mean
- Minimum
- Normalization
- RMS
- Standard Deviation

- To Audio Device
- Variance
- Window Function

Video and Image Processing Blockset

Many blocks in the Video and Image Processing Blockset™ library that support variable-size signals.

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 a structure with time format for logging variable-size signals.
Right-click signal logging (time-series format) does not support variable-size signals.	Use a To Workspace block or a root Output block for logging variable-size signals.
External mode simulation does not support a variable-size signal feeding into a Scope or Display block.	Use the Assignment block to convert a variable-size signal into a fixed-size signal before feeding into Scope or Display blocks for external mode simulation.
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 Signal Processing Toolbox™ software.	Use the Frame Conversion block to convert a signal into sample-based signal.
Variable-size signals must have a discrete sample time.	—
A scalar signal (width equals 1) cannot be a variable-size signal because the maximum size is 1.	—
Real-Time Workshop 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.	—

Limitation	Workaround
Simulink does not support variable-size parameter or DWork vectors.	—
Root-level inport does not support Rapid Accelerator mode.	—

Using Composite Signals

- “About Composite Signals” on page 12-2
- “Creating and Accessing a Bus” on page 12-5
- “Nesting Buses” on page 12-7
- “Bus-Capable Blocks” on page 12-9
- “Using Bus Objects” on page 12-10
- “Using the Bus Editor” on page 12-13
- “Filtering Displayed Bus Objects” on page 12-34
- “Customizing Bus Object Import and Export” on page 12-40
- “Using the Bus Object API” on page 12-47
- “Virtual and Nonvirtual Buses” on page 12-48
- “Connecting Buses to Inports and Outports” on page 12-51
- “Buses and Libraries” on page 12-56
- “Avoiding Mux/Bus Mixtures” on page 12-57
- “Buses in Generated Code” on page 12-64
- “Composite Signal Limitations” on page 12-65

About Composite Signals

In this section...
“Composite Signal Terminology” on page 12-2
“Types of Simulink Buses” on page 12-3
“Buses and Muxes” on page 12-3
“Bus Objects” on page 12-4
“Bus Code” on page 12-4

Composite Signal Terminology

A *composite signal* is a signal that is composed of other signals. The constituent signals originate separately and join to form the composite signal. They can then be extracted from the composite signal downstream and used as if they had never been joined. Composite signals can reduce visual complexity in models by grouping signals that run in parallel over some or all of their courses, and can serve various other purposes.

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 12-9, “Connecting Buses to Inports and Outports” on page 12-51, and “Composite Signal Limitations” on page 12-65 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 10-12 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 12-48 for more information.

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. See “Mux Signals” on page 10-15 for more information. In some cases, muxes and virtual buses can be treated interchangeably; implicit type conversion occurs when needed. However, The MathWorks discourages treating muxes and buses interchangeably. The practice may become unsupported in the future, and should not be used in new applications. Simulink software provides diagnostics that report cases where muxes and virtual buses are used interchangeably, and includes capabilities that you can use to upgrade a model to eliminate such mixtures. See “Avoiding Mux/Bus Mixtures” on page 12-57 for details.

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 “Using Bus Objects” on page 12-10 for more information. You can create bus objects programmatically or by using the Simulink Bus Editor, which facilitates bus object creation and management. See “Using the Bus Editor” on page 12-13 for more information.

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, see “Optimizing Buses for Code Generation” for information about the best techniques to use.

Creating and Accessing 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

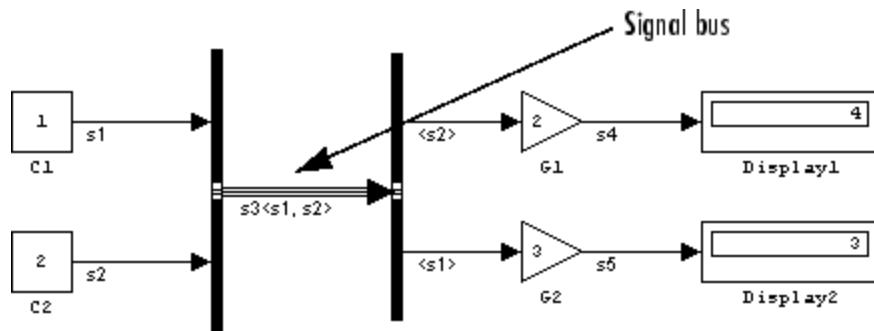
Select elements from a bus

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.

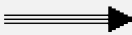

To create and access a bus signal that has default properties:

- 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 that are input to a Bus Creator block, transmitted as a bus signal to a Bus Selector block, and output as separate signals.



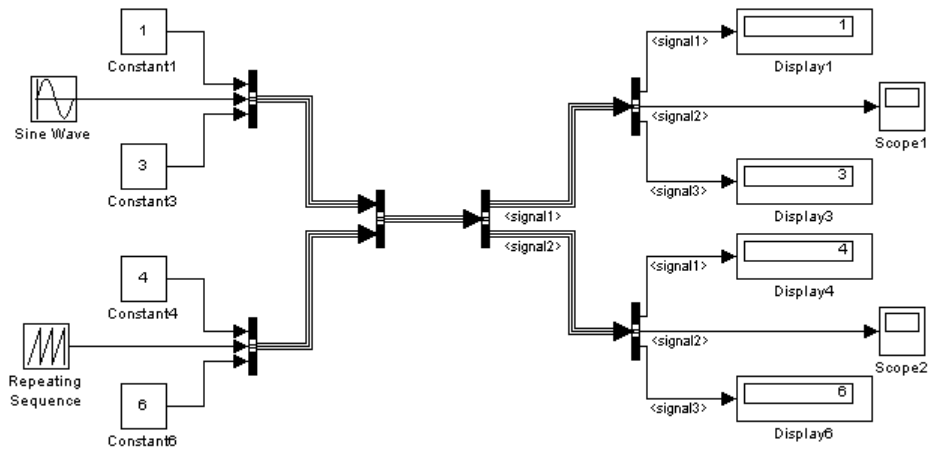
The Bus Creator and Bus Selector blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the bus signal, is tripled because the model has been built, and the middle line is solid because the bus is virtual. The line would be dashed if the bus were nonvirtual:

Virtual Bus	
Nonvirtual Bus	

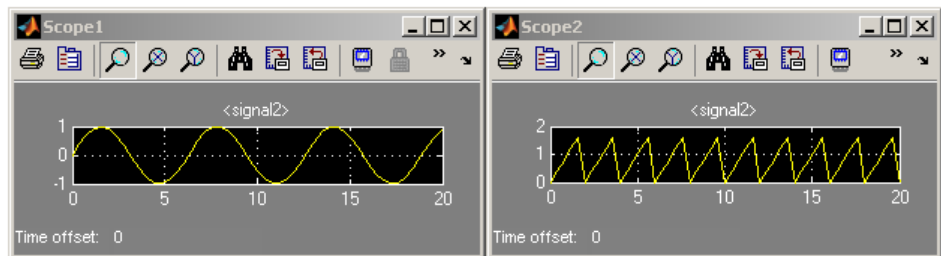
See “Signal Line Styles” on page 10-6 for more about the graphical appearance of signals. You can also display other signal characteristics graphically, as described under “Displaying Signal Properties” on page 10-64. For more information about creating and accessing buses, see the reference documentation for the Bus Creator, Bus Selector, and Bus Assignment blocks.

Nesting Buses

Buses can be nested to any depth. The Simulink software automatically handles most of the complexities involved. For example, the next figure shows six signals nested into two buses, which are nested into one, followed by separation into two buses and then into six separate signals:



The six signals retain their separate identities just as if no bus creation and selection occurred, as shown by the Display and Scope blocks:



Specifying nonvirtual buses, like those in the previous figure, requires only cloning blocks, setting parameters, and connecting signals. Bus Creator and Bus Selector blocks have two ports by default. See the Bus Creator and Bus Selector block documentation for information about how to specify buses of different widths.

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 Source** or **Highlight to Destination** from the signal's Context menu. (See "Displaying Signal Sources and Destinations" on page 10-21 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

Buses are not intended to support computation performed directly on the bus. Therefore, only a small subset of blocks, called *bus-capable blocks*, can process buses directly. All virtual blocks are bus-capable. The following nonvirtual blocks are also bus-capable:

- Memory
- Merge
- Multiport Switch
- Rate Transition
- Switch
- 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 the specific block in *Blocks-Alphabetical List* for more information.

You can sometimes connect a virtual bus to a block that is not bus-capable without generating an error, but such a connection intermixes buses and muxes, which The MathWorks discourages. See “Avoiding Mux/Bus Mixtures” on page 12-57 for details.

Using Bus Objects

In this section...
“About Bus Objects” on page 12-10
“Bus Object Capabilities” on page 12-11
“Associating Bus Objects with Simulink Blocks” on page 12-11

About Bus Objects

The properties that can be specified for a bus signal by its Bus Creator block parameters, and inherited by all downstream blocks that use the bus, are adequate for defining virtual buses and performing limited error checking. Defining a nonvirtual bus, or performing complete error checking on any bus, requires you to specify additional information by providing a *bus object*.

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.

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.

Referenced models, Stateflow charts, and Embedded MATLAB 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 any Bus Creator block, Inport block, or Outport block. 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. Thus 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 did not specify the property, but left 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 “Using the Bus Editor” on page 12-13, or you can use the Simulink API, as described in “Using the Bus Object API” on page 12-47. After you have created a bus object and specified its attributes, you can associate it with any block that needs to use the bus definition that the object provides, as described in “Associating Bus Objects with Simulink Blocks” on page 12-11.

Associating Bus Objects with Simulink Blocks

You can associate a bus object with any Bus Creator block, Inport block, or Outport block. The techniques vary slightly depending on the type of the block.

Associating Bus Objects with Bus Creator Blocks

To associate a bus object with a Bus Creator block:

- 1 Open the Block Parameters dialog.
- 2 Select **Specify properties via bus object**.
- 3 Enter the bus object name in the **Bus object** field.
- 4 Click **OK** or **Apply**.

See the Bus Creator block documentation for more information.

Associating Bus Objects with Inport and Outport Blocks

Inport and Outport blocks use bus objects to provide the specifications necessary for a bus signal to cross the boundary between nonvirtual subsystems.

To associate a bus object with an Inport or Outport block:

- 1** Open the Block Parameters dialog.
- 2** Select the **Signal Attributes** tab.
- 3** Select **Specify properties via bus object**.
- 4** Enter the bus object name in the **Bus object for validating input bus field**.
- 5** Click **OK** or **Apply**.

See the Inport block and Outport block documentation for more information.

Using the Bus Editor

In this section...

“Introduction” on page 12-13
“Opening the Bus Editor” on page 12-14
“Displaying Bus Objects” on page 12-15
“Creating Bus Objects” on page 12-17
“Creating Bus Elements” on page 12-20
“Nesting Bus Definitions” on page 12-23
“Changing Bus Entities” on page 12-25
“Exporting Bus Objects” on page 12-30
“Importing Bus Objects” on page 12-32
“Closing the Bus Editor” on page 12-32

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 an M-file or MAT-file
- Export bus objects to an M-file or MAT-file

For a description of bus objects and their use, see “Using Bus Objects” on page 12-10.

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

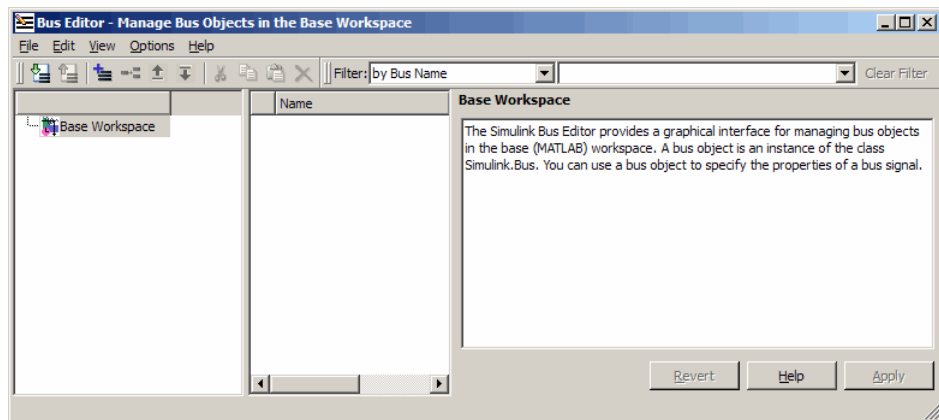
do not directly affect bus object definitions in saved M-files or MAT-files. To save changed bus object definitions, export them from the base workspace into M-files or MAT-files, as described in “Exporting Bus Objects” on page 12-30.

Opening the Bus Editor







You can open the Bus Editor in any of these ways:

- Select **Bus Editor** from the model editor’s **Tools** menu.
- Click the **Launch Bus Editor** button on a bus object’s dialog box in the Model Explorer.
- Enter `buseditor` at the command line of the MATLAB software.

After you have performed any of these actions, the Bus Editor appears. If no bus objects exist, the Bus Editor looks like this:



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 GUI 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 an M-file or MAT-file into the base workspace.
Export		Export all bus objects and elements to an M-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 a bus object's elements.

For brevity, this section refers only to menu commands. You can use toolbar icons and keyboard shortcuts instead wherever that is convenient.

Displaying Bus Objects

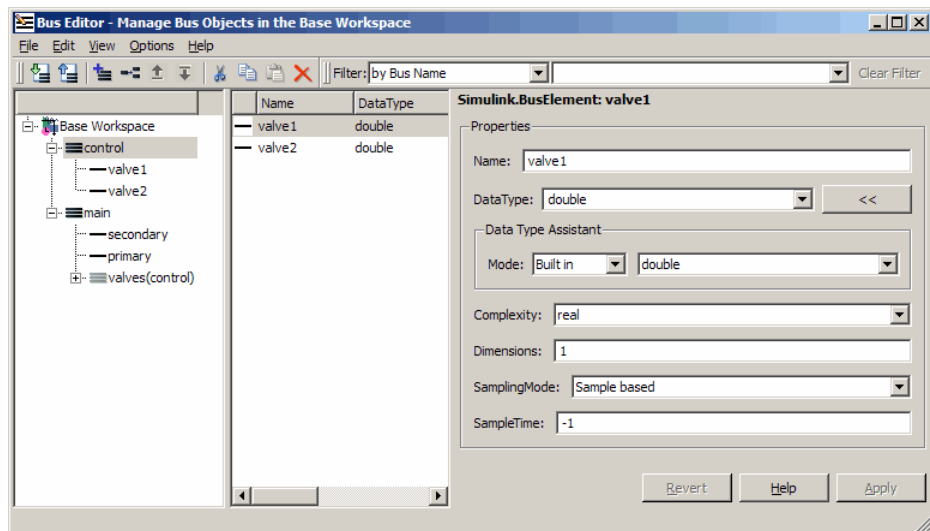
The Bus Editor is similar to the Model Explorer (which can display bus objects but cannot edit them) and uses the same three panes to display bus objects:

- **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 `control1` 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 `control1`, `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 12-27.

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 12-28.

Filter Boxes

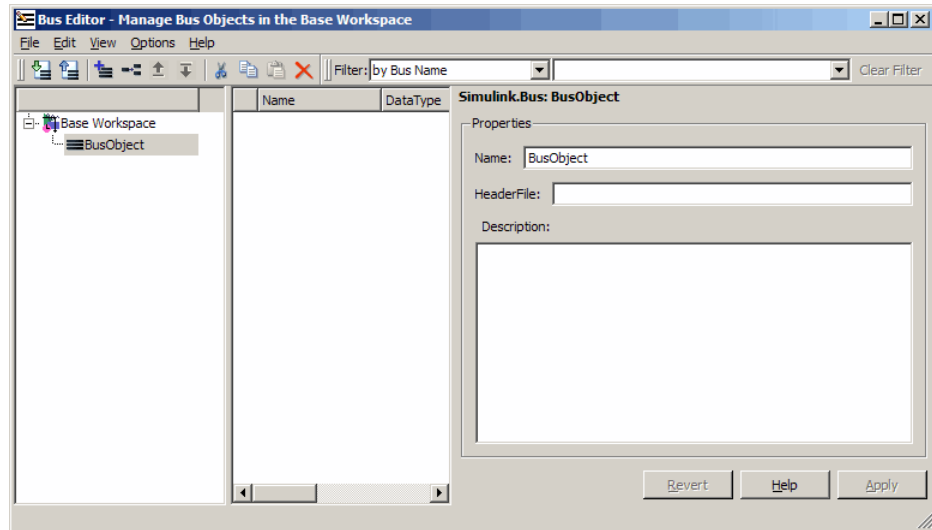
By default, the Bus Editor displays all bus objects that exist in the base workspace. Where a model contains large numbers of bus objects, seeing them all at the same time can be inconvenient. To facilitate working efficiently with large collections of bus objects, you can use the **Filter** boxes, to the right of the iconic tools in the toolbar, to show a selected subset of bus objects. See “Filtering Displayed Bus Objects” on page 12-34 for details.

Creating Bus Objects

To use the Bus Editor to create a new bus object in the base workspace:

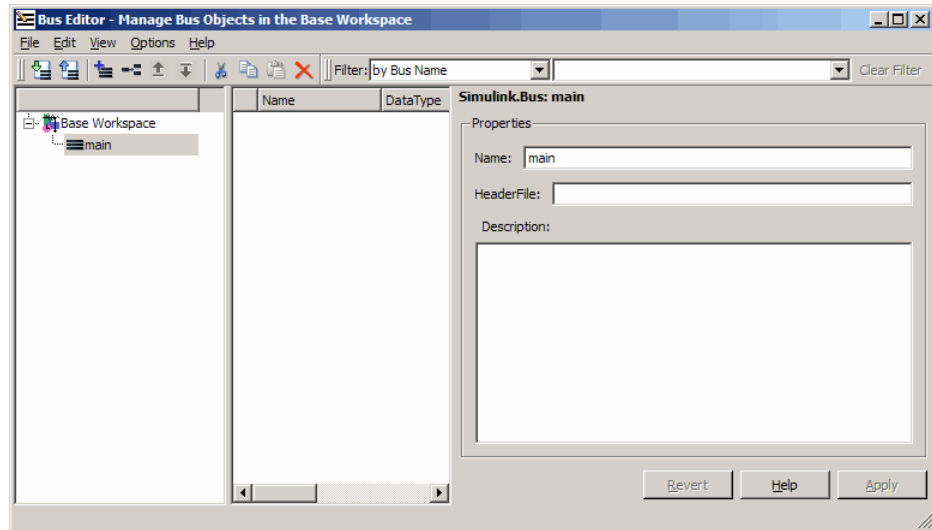
1 Choose **File > Add Bus**.

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:

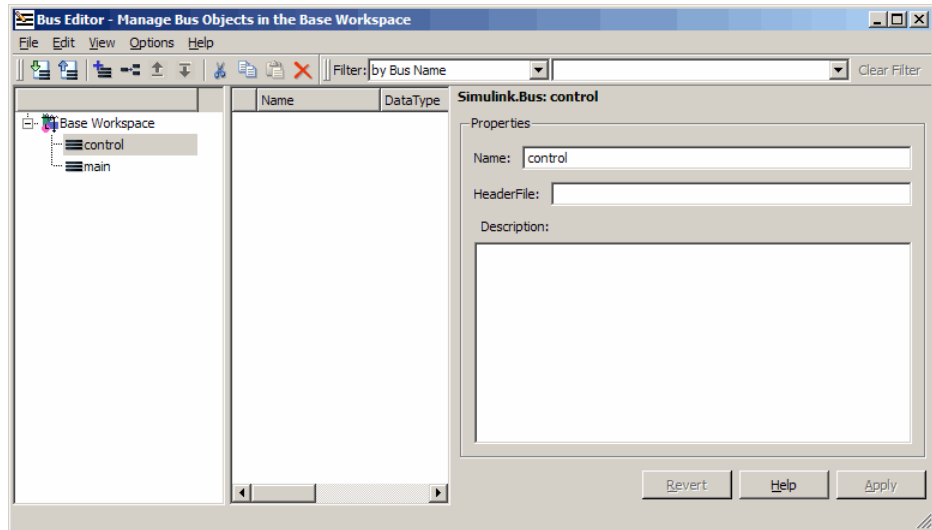


- 2 To specify the bus object name and other properties, in the Dialog pane:
 - a Specify the **Name** of the of the new bus object (or you can retain the default name). The name must be unique in the base workspace.
 - b Optionally, specify a **C Header file** that defines a user-defined type corresponding to this bus object. This header file has no effect on Simulink simulation; it is used only by Real-Time Workshop 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.

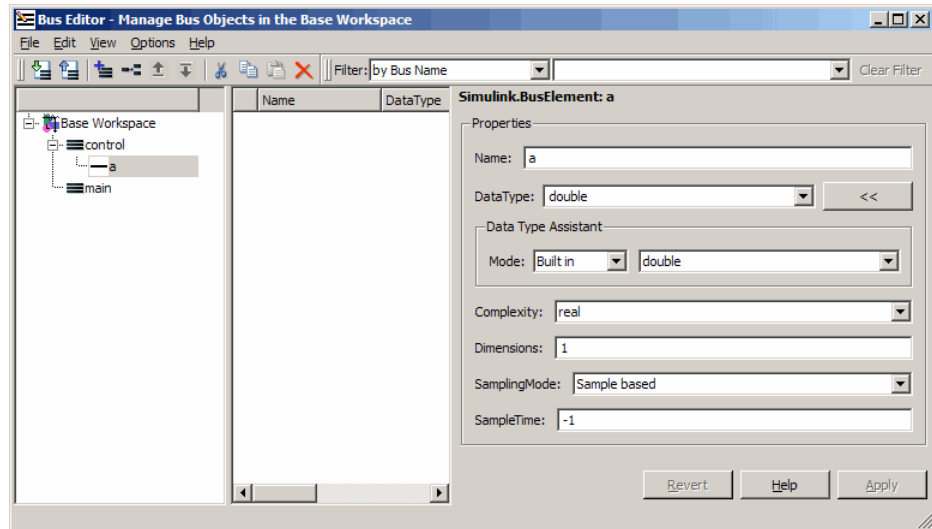
Creating Bus Elements

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.

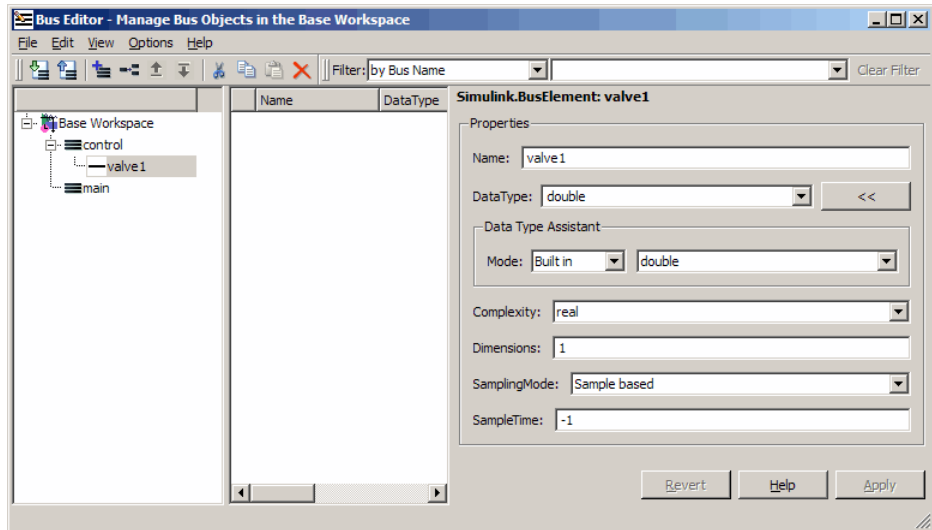
- 2 Choose **File > Add/Insert BusElement**.

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:

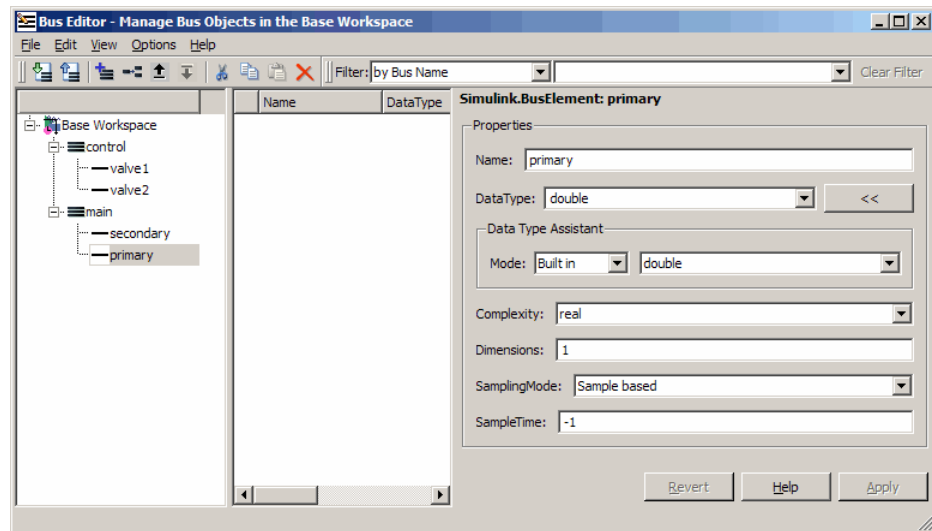


- 3 To specify the bus element name and other properties, in the Dialog pane:
 - a Specify the **Name** of the of the new bus element (or you can retain the default name). The name must be unique among the elements of the bus object.
 - b Specify the other properties of the element. These must match exactly the properties of the corresponding signal within the bus, 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 `control1`, 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.

Nesting Bus Definitions

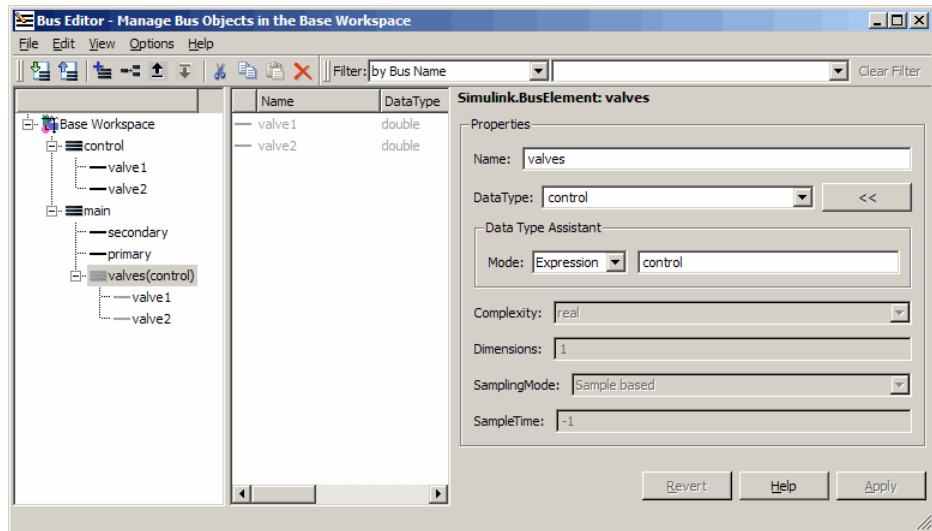
As described in “Nesting Buses” on page 12-7, any signal in a bus can be another bus, which can in turn contain subordinate buses, and so on to any depth. Describing nested buses with bus objects requires nesting the bus definitions that the objects provide.

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.

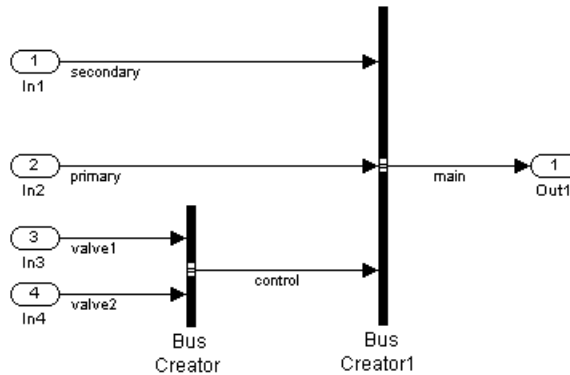
A data type defined by a bus object is called a *bus type*. Nesting buses by assigning bus types to elements, rather than by subordinating the bus objects that define the types, allows the same bus definition to be used conveniently in multiple contexts without unwanted interactions. To specify that 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.

Changing 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 “Filtering Displayed Bus Objects” on page 12-34 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 “Nesting Buses” on page 12-7.

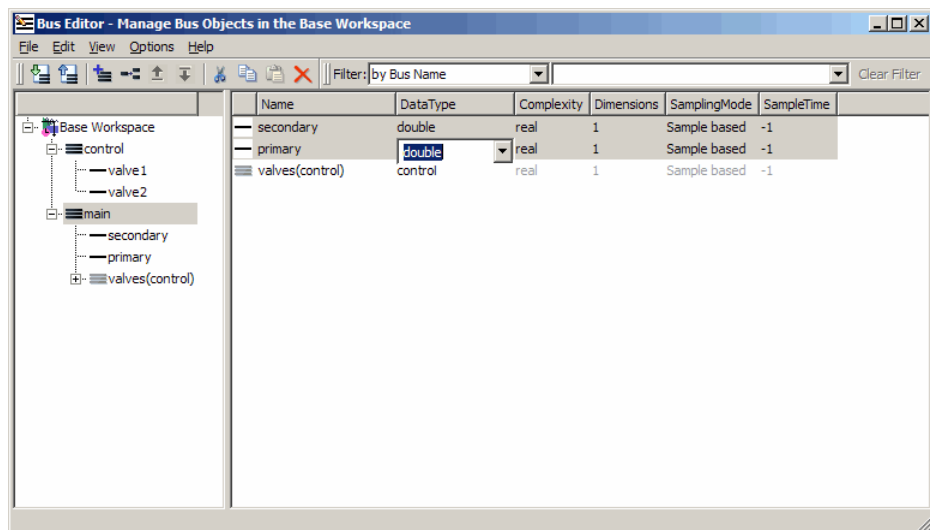
Editing in the Contents Pane

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.

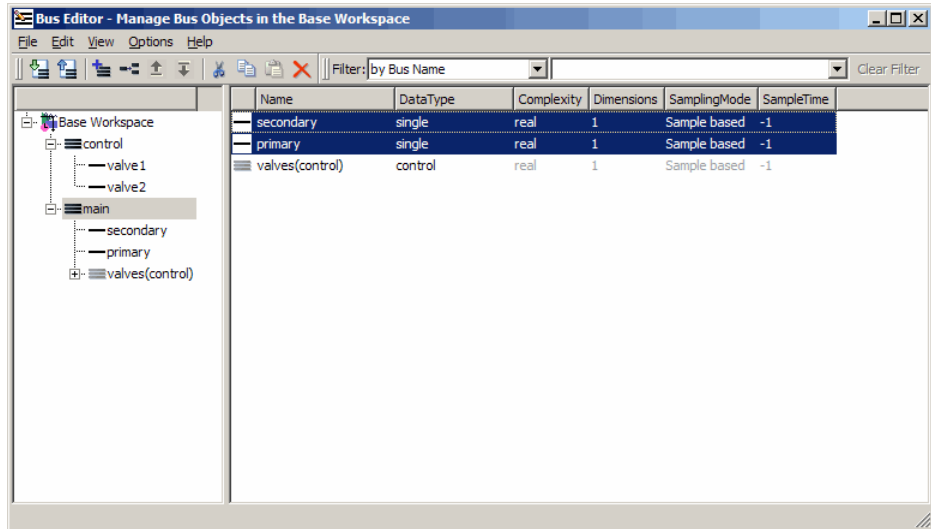
Choose **View > Dialog View** to hide the Dialog pane to provide more room to display properties in the Contents pane. Choose the command again to redisplay the Dialog pane.

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:

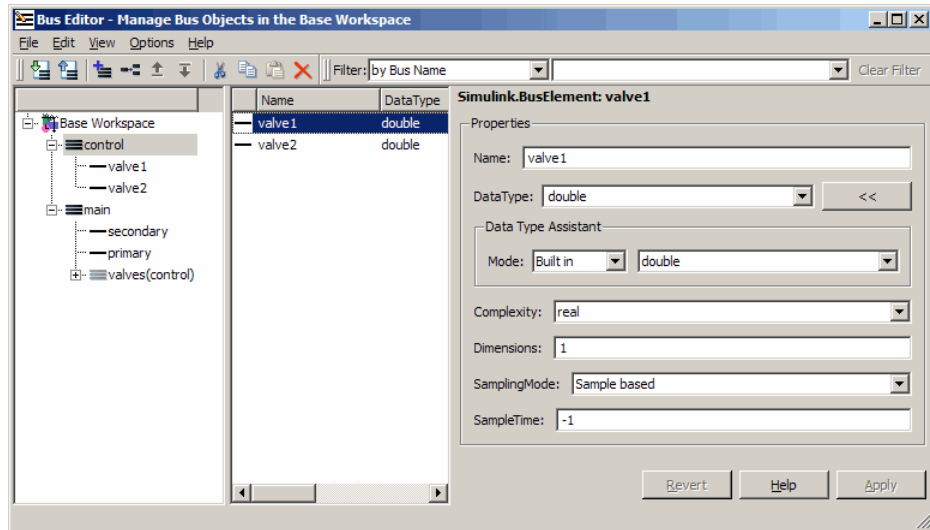


If you change the value of **DataType** to `single` and press Return, the value changes for both elements. The effect is the same no matter which element you edit in a multiple selection:

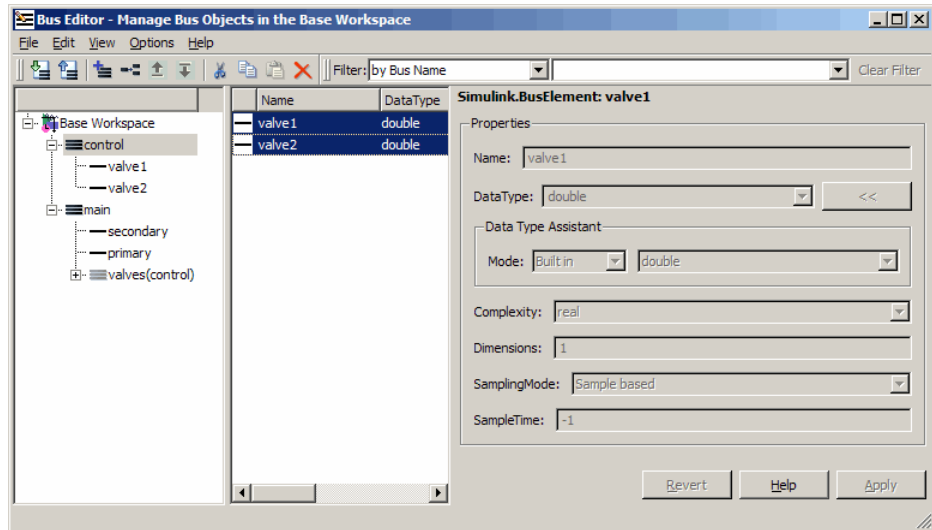


Editing in the Dialog Pane

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**.

Exporting Bus Objects

Like all base workspace objects, bus objects are not saved with a model that uses them, but exist separately in an M-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 an M-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 M-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** 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, an M-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 “Customizing Bus Object Import and Export” on page 12-40 for details.

Importing Bus Objects

You can use the Bus Editor to import the definitions in an M-file or MAT-file to the base workspace. Importing an M-file or MAT-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, an M-file or MAT-file stored in the file system. For example, the bus objects could be retrieved from records in a corporate database. See “Customizing Bus Object Import and Export” on page 12-40 for details.

Closing 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 “Exporting Bus Objects” on page 12-30. 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**.

Filtering Displayed Bus Objects

In this section...

“Filtering by Name” on page 12-35

“Filtering by Relationship” on page 12-36

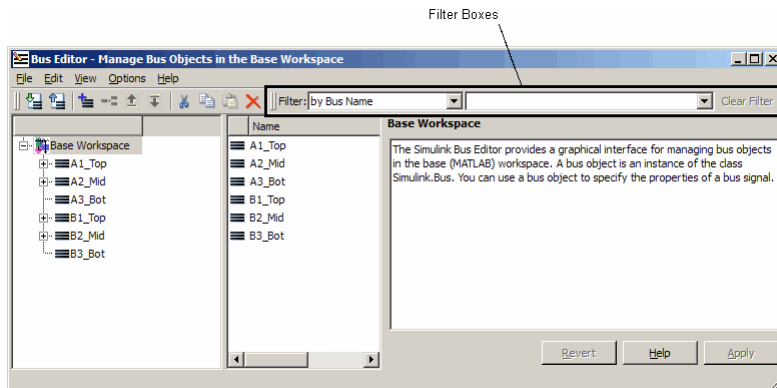
“Changing Filtered Objects” on page 12-38

“Clearing the Filter” on page 12-39

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

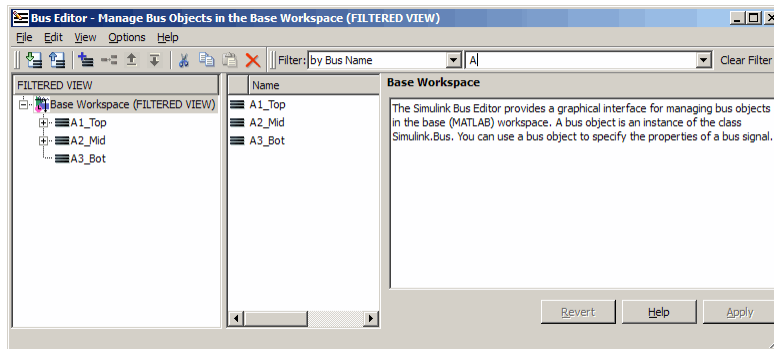
To set a filter, you specify values in the **Filter** boxes to the right of the tools in the toolbar. The left **Filter** box specifies the type of filtering. This box always appears, and is called the **Filter Type** box. Depending on the specified type of filtering, one or two boxes appear to the right of the **Filter Type** box. The next figure indicates the **Filter** boxes and shows that six bus objects exist in the Base Workspace:



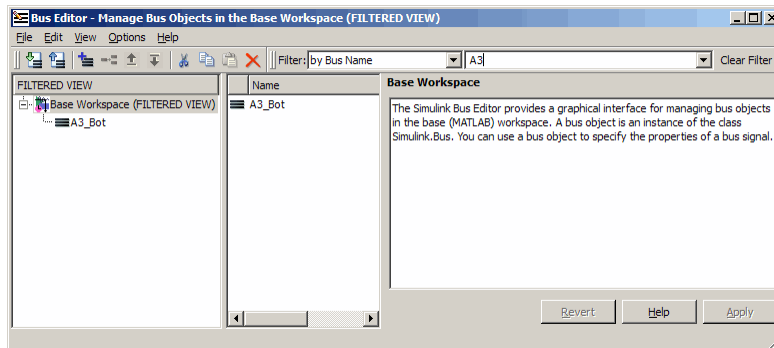
The bus objects shown form two disjoint hierarchies. A1_Top is the parent of A2_Mid, which is the parent of A3_Bot. Similarly, B1_Top > B2_Mid > B3_Bot. See “Nesting Buses” on page 12-7 for information about bus object hierarchies.

Filtering by Name

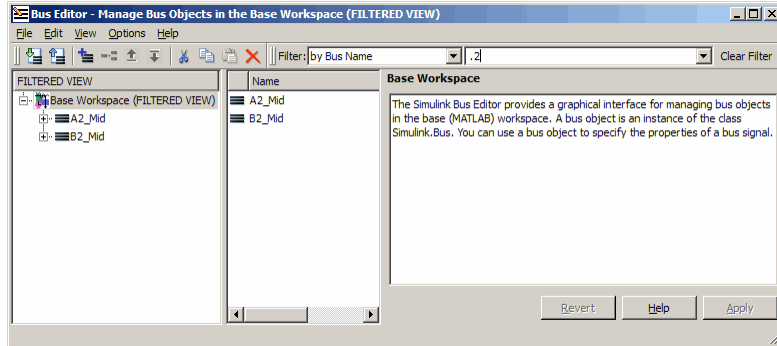
To filter bus objects by name, set the **Filter Type** box to by **Bus Name** (which is the default). The right **Filter** box is the **Object Name** box. Type any MATLAB regular expression (which can just be a string) into the **Object Name** box. As you type, the Bus Editor updates dynamically to show 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:



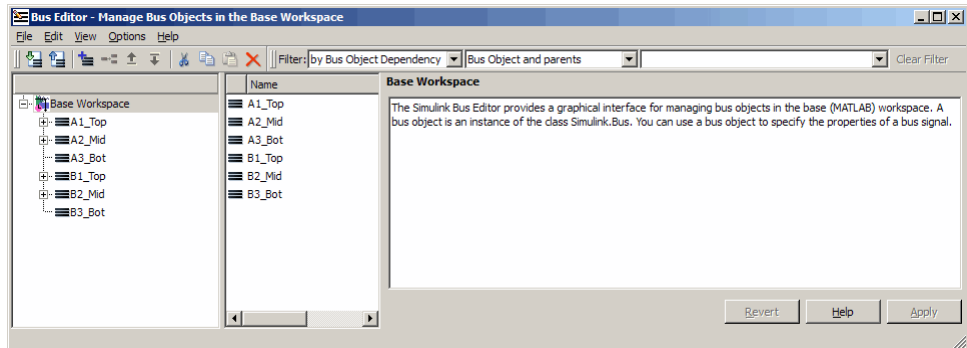
In a MATLAB regular expression, the metacharacter dot (.) matches any character, so entering .2 displays:



See “Regular Expressions” for complete information about MATLAB regular expression syntax.

Filtering 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:

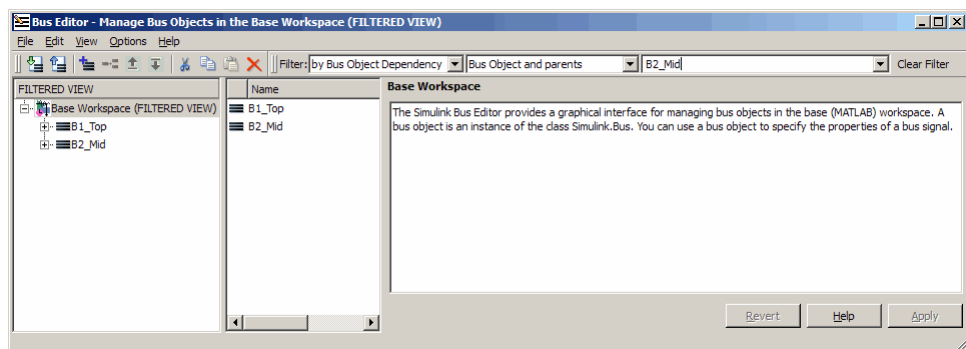
- **Bus Object and Parents** — Show a specified bus object and all superior bus objects in the hierarchy (default)
- **Bus Object and Dependents** — Show a specified bus object and all subordinate bus objects in the hierarchy
- **Bus Object and Related Objects** — Show a specified bus object and all superior and subordinate bus objects

In the **Object Name** box, specify a bus object by name. You can use the dropdown to select any existing bus object name, or you can type a name. As you type, the editor:

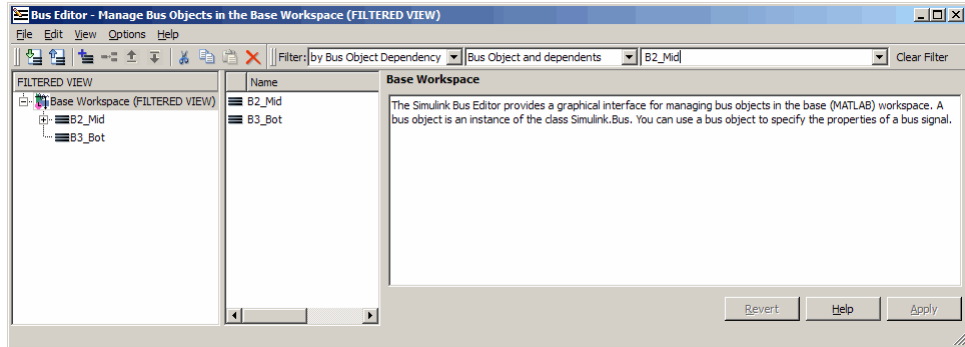
- Dynamically completes the field to indicate the first bus object that alphabetically matches what you have typed.
- Updates the display panes to show only that object and any objects that have the specified relationship to it.

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:

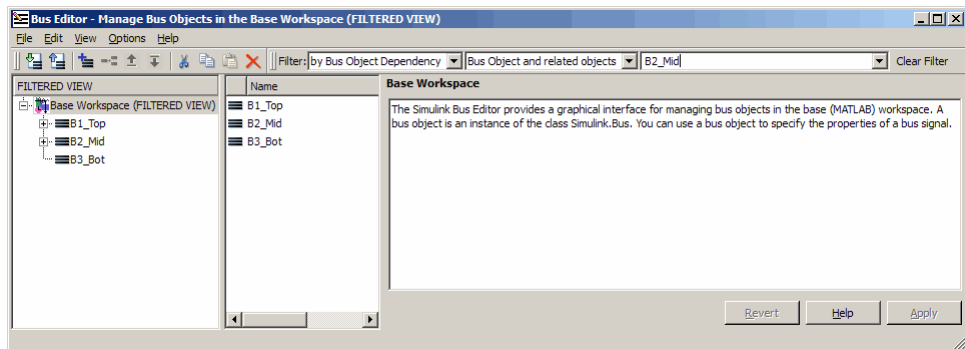
Bus Object and Parents



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.

Changing 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.

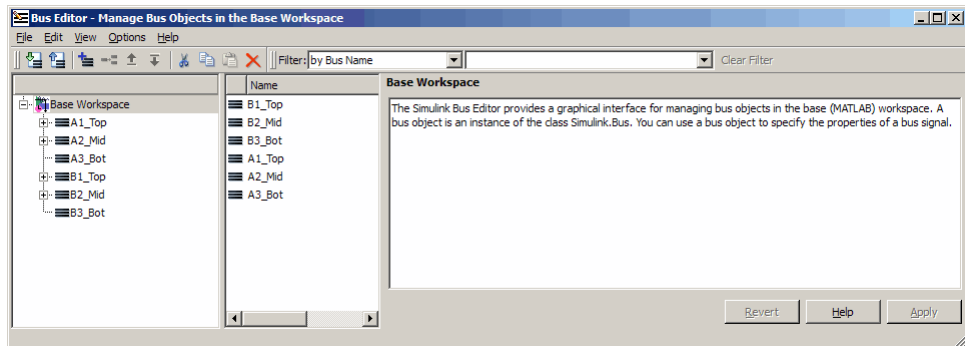
A new bus object created within the Bus Editor with a filter in effect may or may not appear, depending on the filter. If you create a new bus object but do

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.

Clearing 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.

Customizing Bus Object Import and Export

In this section...

“Prerequisites for Customization” on page 12-41

“Writing a Bus Object Import Function” on page 12-41

“Writing a Bus Object Export Function” on page 12-42

“Viewing the Customization Manager” on page 12-43

“Registering Customizations” on page 12-44

“Changing Customizations” on page 12-45

You can use the Bus Editor to import bus objects to the base workspace, as described in “Importing Bus Objects” on page 12-32, and to export bus objects from the base workspace, as described in “Exporting Bus Objects” on page 12-30. By default, the Bus Editor can import bus objects only from an M-file or MAT-file, and can export bus objects only to an M-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 M-functions that provide the desired capabilities, and registering these M-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 M-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's **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 M language and M programming techniques that you will need. See *MATLAB Programming Fundamentals* and related MATLAB documentation for information about M programming.
- Simulink bus object syntax. See “About Bus Objects” on page 12-10, `Simulink.Bus`, `Simulink.BusElement`, and “Nesting Bus Definitions” on page 12-23.
- 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 Chapter 28, “Customizing the Simulink User Interface”.

Writing a Bus Object Import Function

A callback function that customizes bus import can use any M 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 M 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 M 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 M 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.

Viewing the Customization Manager

The Simulink Customization Manager allows you to customize many Simulink capabilities. Complete information about it appears in Chapter 28, “Customizing the Simulink User Interface”. To understand how the Simulink Customization Manager provides bus object import and export customization, you must know something about its structure. To obtain the manager for inspection, execute:

```
cm=Simulink.CustomizationManager
```

where `cm` could be any MATLAB variable name. The call looks like a class instantiation, but actually it retrieves a pre-existing object; a second call would not create a second instance. To see the elements of the Customization Manager, execute:

```
cm.get
```

The result is a list of elements:

```
showWidgetIdAsToolTip: 0
registerTargetInfo: @registerTargetInfo
ExtModeTransports: [1x1 Simulink.ExtModeTransports]
BusEditorCustomizer: [1x1 BusEditor.customizer]
slDataObjectCustomizer: [1x1 mpt.SLDataObjectCustomizer]
RTWBuildCustomizer: [1x1 mpt.RTWBuildCustomizer]
miscCustomizer: [1x1 mpt.MiscCustomizer]
```

The relevant element is the `BusEditorCustomizer`. To see its elements, execute:

```
cm.BusEditorCustomizer.get
```

The elements in their initial state are:

```
importCallbackFcn: []  
exportCallbackFcn: []
```

Registering bus editor import and export callbacks requires setting the values of these elements to be handles to your custom import and export functions. This section shows examples of M function handles, but does not describe their syntax. See “Function Handles” in the MATLAB documentation for information about M function handle syntax.

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 M 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 `Simulink.CustomizationManager`, as shown in “Viewing the Customization Manager” on page 12-43.

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.

Using the Bus Object API

The Simulink software provides all Bus Editor capabilities programmatically. Many of these capabilities, like importing and exporting M-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 an M-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 an M-file in a specified format

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 an M-file.

Virtual and Nonvirtual Buses

In this section...
“Introduction” on page 12-48
“Creating Nonvirtual Buses” on page 12-48
“Nonvirtual Bus Sample Times” on page 12-49
“Automatic Bus Conversion” on page 12-50

Introduction

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. 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. Simulation results and generated code are exactly the same as if the bus did not exist, which functionally it does not.

By contrast, 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. Such a bus is represented by a structure in generated code, which can be helpful when tracing the correspondence between the model and the code.

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. Not all blocks can accept buses. See “Bus-Capable Blocks” on page 12-9 for more about which blocks can handle which types of buses. Virtual buses are the default except where nonvirtual buses are explicitly required. See “Connecting Buses to Inports and Outports” on page 12-51 for more information.

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
- Outport

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 12-11.
- 2** Open the **Block Parameters** dialog of the Bus Creator, Inport, or Outport block.
- 3** Do one of the following, depending on the type of the block:
 - **Bus Creator:** Select **Output as nonvirtual bus**.
 - **Inport:** Select **Signal Attributes > Output as nonvirtual bus**.
 - **Outport:** Select **Signal Attributes > Output as nonvirtual bus in parent model**.
- 4** 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. You can eliminate the need for the automatic conversion by specifying a nonvirtual bus in the block where the bus originates, or by inserting explicit Signal Conversion blocks. The latter is generally unnecessary, but can be useful to clarify the model.

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 “Using Bus Objects” on page 12-10. Conversion to a nonvirtual bus fails if no bus object is specified, and the Simulink software posts an error message describing the problem.

Connecting Buses to Inports and Outports

In this section...

“Connecting Buses to Root Level Inports” on page 12-51

“Connecting Buses to Root Level Outports” on page 12-51

“Connecting Buses to Nonvirtual Inports” on page 12-52

“Connecting Buses to Model, Stateflow, and Embedded MATLAB Blocks” on page 12-54

“Connecting Multi-Rate Buses to Referenced Models” on page 12-54

Connecting Buses to Root Level Inports

If you want a root level Inport of a model to produce a bus signal, you must select the Inport’s **Specify properties via bus object** parameter and set the Inport’s **Bus object for validating input bus** parameter to the name of a bus object that defines the bus that the Inport produces. If the bus contains mixed data types, the Inport’s data type must be auto. See “Using Bus Objects” on page 12-10 for more information.

Connecting Buses to Root Level Outports

A root level Outport of a model can accept a virtual bus only if all elements of the bus have the same data type. The Outport block automatically unifies the bus to a vector having the same number of elements as the bus, and outputs that vector.

If you want a root level Outport of a model to accept a bus signal that contains mixed types, you must select the Outport’s **Specify properties via bus object** parameter and set the Outport’s **Bus object for validating input bus** parameter to the name of a bus object that defines the type of bus that the Outport produces. If the bus signal is virtual, it will be converted to nonvirtual, as described in “Automatic Bus Conversion” on page 12-50. See “Using Bus Objects” on page 12-10 more information.

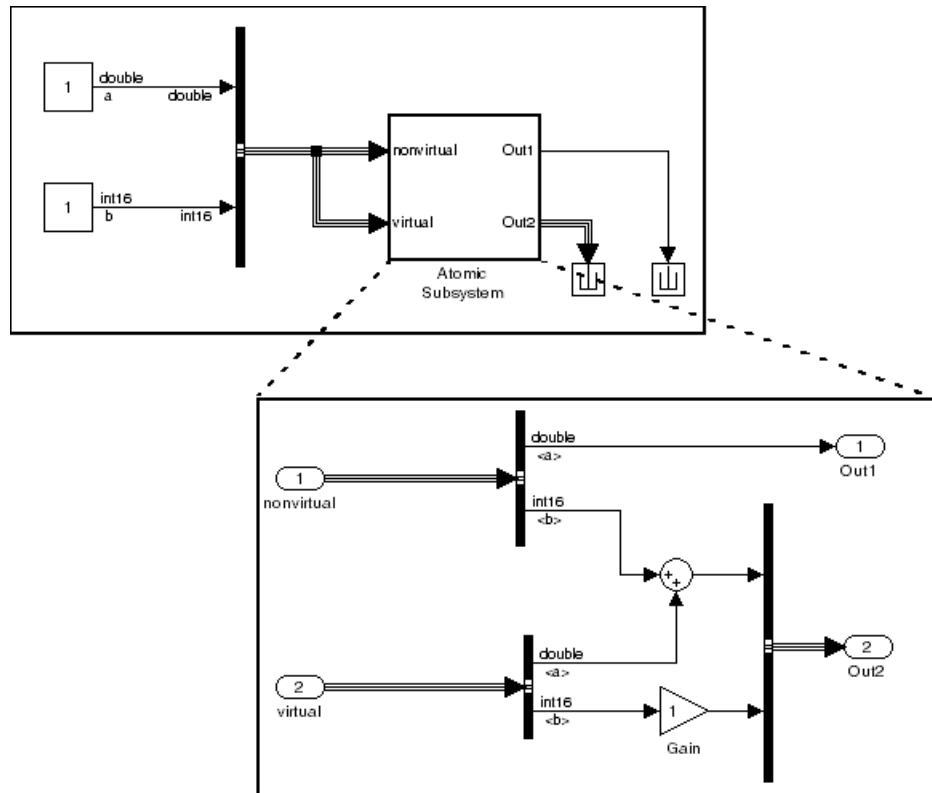
An Outport in a conditionally executed subsystem that is connected to a bus that contains mixed data types cannot be configured to reset and have initial values specified.

Connecting 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.

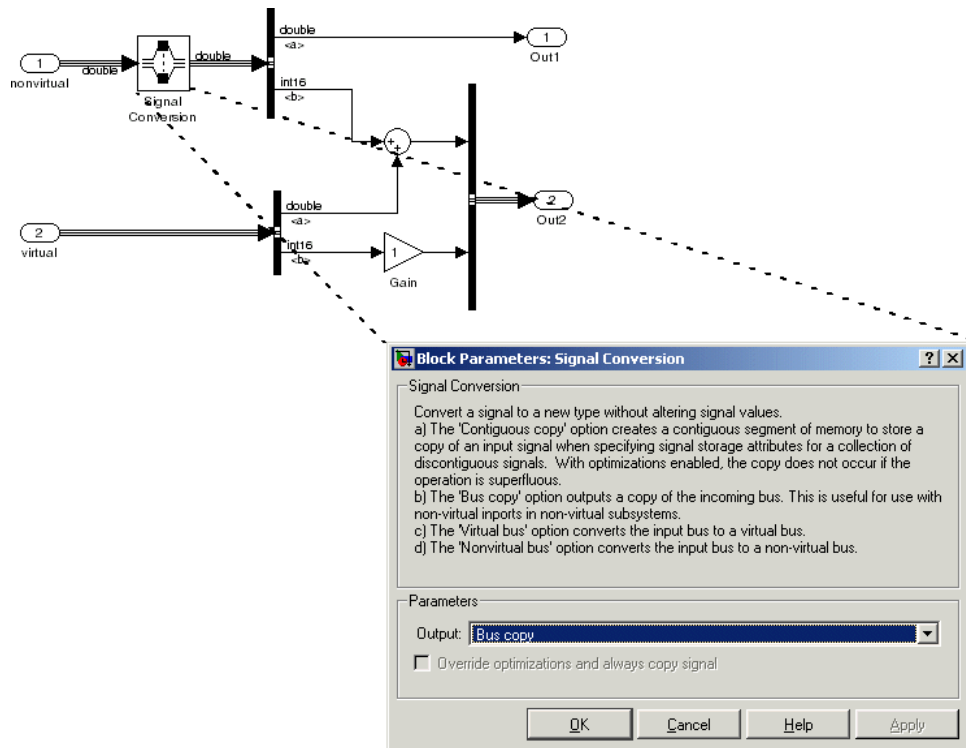
If the components are of differing data types, attempting to simulate the model causes the Simulink software to halt the simulation and display an error message. You can avoid this problem, without changing the semantics of your model, by inserting a Signal Conversion block between the Inport block and the Outport block to which it was originally connected.

For example, consider the following model:



In this model, the Inport 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 connected to the subsystem's inputs has components of differing data types. As a result, this model cannot be simulated.

Inserting a Signal Conversion block with the `bus copyoption` selected breaks the direct connection to the subsystem's output and thereby enables the Simulink software to simulate the model.



Connecting Buses to Model, Stateflow, and Embedded MATLAB Blocks

Referenced models, Stateflow charts, and Embedded MATLAB 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 12-50 for details.

Connecting 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.

Buses and Libraries

When you define a library block, the block can input, process, and output buses just as an ordinary subsystem can. The MathWorks recommends not using Bus Selector blocks in library blocks, because such use complicates changing the library blocks and increases the likelihood of errors. When a Bus Selector block appears in a library block, the following considerations and recommendations apply:

- You cannot change a Bus Selector block directly within a library. To change the Bus Selector block, copy the library block that uses it to a model, edit the Bus Selector block within the context of the model, then copy the changed library block back to the library.
- The Inport that feeds the Bus Selector block should have an associated bus object, as described in “Using Bus Objects” on page 12-10. This bus object must be stored in an M-file or MAT-file, and imported into the base workspace of any model that subsequently uses the library block.
- Any model that uses the library block containing the Bus Selector should set **Configuration Parameters > Connectivity > Element name mismatch** to error. This setting minimizes the possibility of consistency errors at the interface to the library block.

See Chapter 8, “Working with Block Libraries” for information about creating block libraries and copying library blocks to and from them.

Avoiding Mux/Bus Mixtures

In this section...

“Introduction” on page 12-57

“Using Diagnostics for Mux/Bus Mixtures” on page 12-58

“Using the Model Advisor for Mux/Bus Mixtures” on page 12-60

“Correcting Buses Used as Muxes” on page 12-62

“Bus to Vector Block Backward Compatibility” on page 12-63

“Avoiding Mux/Bus Mixtures When Developing Models” on page 12-63

Introduction

By default, muxes and virtual buses can be used interchangeably if all constituent signals have the same attributes and no nested buses exist. Such a signal could have been implemented as either a mux or a virtual bus, and the Simulink software by default automatically interconverts between the two formats as needed. A situation that treats muxes and virtual buses interchangeably is called a *mux/bus mixture*. All such mixtures occur in one of two ways:

- Mux blocks used to create virtual buses, such as a Mux block that outputs to a Bus Selector block
- Virtual bus signals treated as muxes, such as a bus signal that inputs directly to a Gain block

Neither of these mixtures is compatible with strong type checking, so both increase the likelihood of run-time errors. You should not create mux/bus mixtures in new applications, and you should upgrade existing applications to avoid such mixtures. See “Mux Signals” on page 10-15 for information about muxes.

The Simulink software provides Configuration Parameters diagnostics, Model Advisor checks, and other tools for detecting and correcting mux/bus mixture, as described in this section.

Note Do not confuse muxes used as bus elements, which is legal and causes no problems, with mux/bus mixtures. A mux/bus mixture occurs only when some blocks treat a signal as a mux, while other blocks treat that same signal as a bus.

Using Diagnostics for Mux/Bus Mixtures

Two controls are provided on the **Diagnostics > Connectivity** pane to detect mux/bus mixtures:

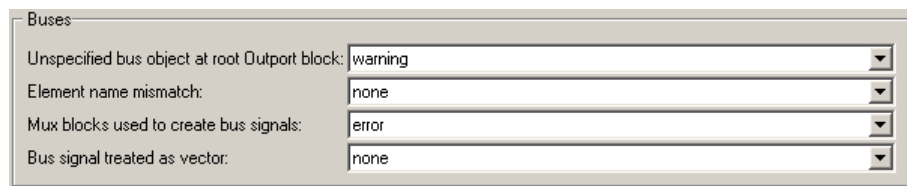
- **Mux blocks used to create bus signals**
- **Bus signal treated as vector**

You can use these diagnostics as described in this section, or you can use the Model Advisor to perform the same checks and also obtain advice about corrections, as described in “Consulting the Model Advisor” on page 4-80. For complete information about the **Connectivity** pane, see “Connectivity Diagnostics Overview”.

Mux blocks used to create bus signals

To detect and correct muxes that are used as buses:

- 1 Set Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to warning or error.
- 2 Set Configuration Parameters > Diagnostics > Connectivity > Buses > Bus signal treated as vector** to none.



The screenshot shows a configuration pane titled "Buses" with four rows of settings, each with a dropdown menu:

Setting	Value
Unspecified bus object at root Output block:	warning
Element name mismatch:	none
Mux blocks used to create bus signals:	error
Bus signal treated as vector:	none

- 3 Click OK or Apply.**
- 4 Build the model.**

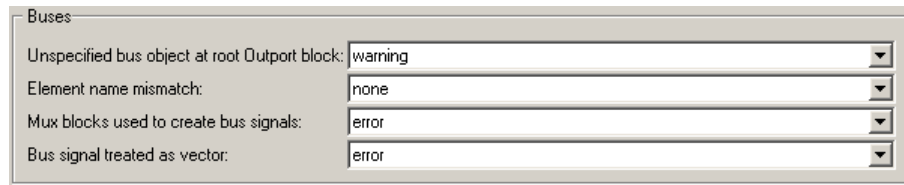
- 5 Replace blocks as needed to correct any cases of Mux blocks used to create buses. You can use the `slreplace_mux` function to replace all such Mux blocks in a single operation.

For complete information about this option, see the reference documentation for “Mux blocks used to create bus signals”.

Bus signal treated as vector

To detect and correct buses that are used as if they were muxes (vectors):

- 1 Correct any cases of Mux blocks used to create buses as described above.
- 2 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to error.
- 3 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Bus signal treated as vector** to warning or error.



- 4 Click **OK** or **Apply**.
- 5 Build the model.
- 6 Correct the model where needed as described under “Correcting Buses Used as Muxes” on page 12-62.

For complete information about this option, see the reference documentation for “Mux blocks used to create bus signals”.

Note **Bus signal treated as vector** is disabled unless **Mux blocks used to create bus signals** is set to **error**. Setting **Bus signal treated as vector** to **error** has no effect unless you have previously corrected all cases of Mux blocks used to create buses.

Equivalent Parameter Values

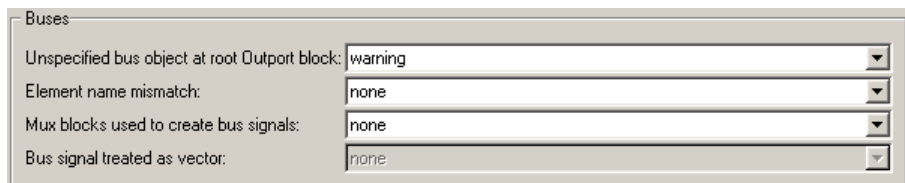
Due to the requirement that **Mux blocks used to create bus signals** be **error** before **Bus signal treated as vector** is enabled, one parameter, `StrictBusMsg`, can specify all permutations of the two controls. The parameter can have one of five values. The following table shows these values and the equivalent GUI control settings:

Value of <code>StrictBusMsg</code> (API)	Mux blocks used to create bus signals (GUI)	Bus signal treated as vector (GUI)
None	none	none
Warning	warning	none
ErrorLevel1	error	none
WarnOnBusTreatedAsVector	error	warning
ErrorOnBusTreatedAsVector	error	error

Using the Model Advisor for Mux/Bus Mixtures

The Model Advisor provides a convenient way to both run the diagnostics for mux/bus mixtures and obtain advice about corrections. To use the Model Advisor to detect and correct mux/bus mixtures:

- 1 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to **none**.



2 Click **OK** or **Apply**.

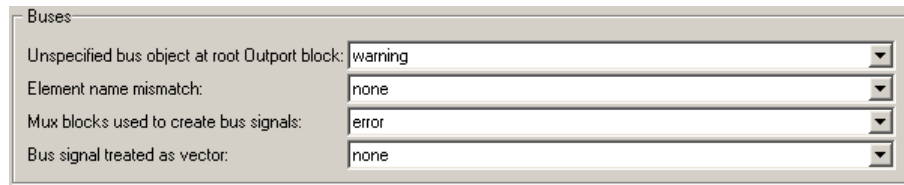
3 Select and run the Model Advisor check **Simulink > Check for proper bus usage**.

The Model Advisor reports any cases of Mux blocks used to create bus signals.

4 Follow the Model Advisor's suggestions to correct any errors reported by the check. You can use the `s1replace_mux` function to replace all such errors in a single operation.

5 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to error.

6 Set **Configuration Parameters > Diagnostics > Connectivity > Bus signal treated as vector** to none.



Buses	
Unspecified bus object at root Output block:	warning
Element name mismatch:	none
Mux blocks used to create bus signals:	error
Bus signal treated as vector:	none

7 Click **OK** or **Apply**.

8 Again run the Model Advisor check **Simulink > Check for proper bus usage**.

The Model Advisor reports any cases of bus signals treated as muxes (vectors).

9 Follow the Model Advisor's suggestions and the information in "Correcting Buses Used as Muxes" on page 12-62 to correct any errors discovered by the check.

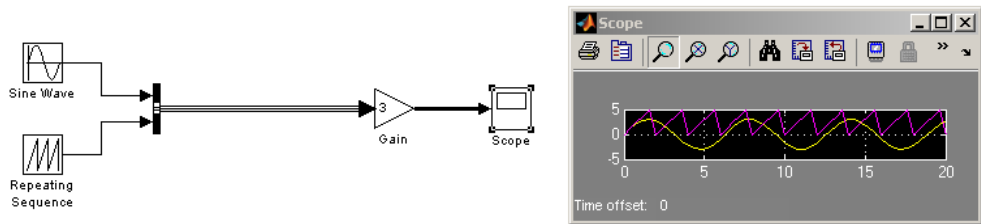
Instructions for using the Model Advisor appear in "Consulting the Model Advisor" on page 4-80.

Correcting Buses Used as Muxes

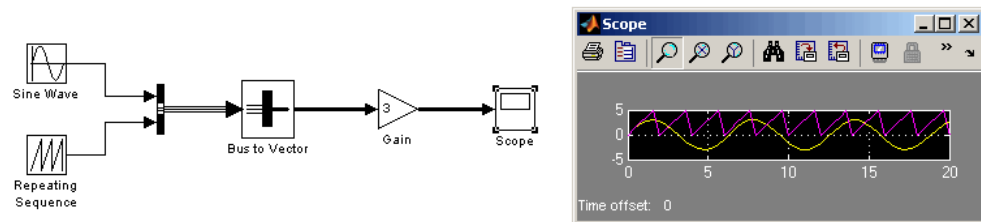
When you discover a bus signal used as a mux, one answer is to reorganize the model by replacing blocks so that the mixture no longer occurs. Where that is undesirable or unfeasible, the Simulink software provides two capabilities to address the problem:

- The Bus to Vector block (Signal Attributes library), which you can insert into any bus used implicitly as a mux to explicitly convert the bus to a mux (vector).
- The `Simulink.BlockDiagram.addBusToVector` function, which automatically inserts Bus to Vector blocks wherever needed.

For example, this figure shows a model that uses a bus as a mux by inputting the bus to a Gain block.



This figure shows the same model, rebuilt after inserting a Bus to Vector block into the bus.



Note that the results of simulation are the same in either case. The Bus to Vector block is virtual, and never affects simulation results, code generation, or performance. For more information, see the reference documentation for the Bus to Vector block and the `Simulink.BlockDiagram.addBusToVector` function.

Bus to Vector Block Backward Compatibility

If you use **Save As** to save a model in a version of the Simulink product before R2007a (V6.6), the following is done:

- Set the `StrictBusMsg` parameter to error if its value is `WarnOnBusTreatedAsVector` or `ErrorOnBusTreatedAsVector`.
- Replace 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.

Avoiding Mux/Bus Mixtures When Developing Models

The MathWorks discourages the use of mux/bus mixtures, and may cease to support them at some future time. The MathWorks therefore recommends upgrading existing models to eliminate any mux/bus mixtures, and permanently setting **Mux blocks used to create bus signals** and **Bus signal treated as vector** to error in all new models and all existing models that may undergo further development.

Note The Bus to Vector block is intended only for use in existing models to facilitate the elimination of implicit conversion of buses into muxes. New models and new parts of existing models should avoid mux/bus mixtures, and should not use Bus to Vector blocks for any purpose.

Buses in Generated 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. For example, a nonvirtual bus appears as a structure in generated code, and only one copy exists of any algorithm that uses the bus. 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 “Optimizing Buses for Code Generation” for information about the best techniques to use.

Composite Signal Limitations

- Simulink and Real-Time Workshop both support using arrays as elements of a bus. However, neither product supports using buses as elements of an array. The capabilities that of an array of buses can often be emulated by using a bus whose elements are arrays.
- Buses that contain signals of enumerated data types cannot pass through a block that requires an initial value, like a Unit Delay block. See “Enumerated Types and Bus Initialization” on page 14-25 for instructions on how to work around this limitation.
- 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 “Logging Signals” on page 15-3.

Working with Data

- “Working with Data Types” on page 13-2
- “Working with Data Objects” on page 13-26
- “Subclassing Simulink Data Classes” on page 13-40
- “Associating User Data with Blocks” on page 13-57

Working with Data Types

In this section...

- “About Data Types” on page 13-2
- “Data Types Supported by Simulink” on page 13-3
- “Fixed-Point Data” on page 13-4
- “Enumerated Data” on page 13-6
- “Block Support for Data and Numeric Signal Types” on page 13-6
- “Creating Signals of a Specific Data Type” on page 13-6
- “Specifying Block Output Data Types” on page 13-6
- “Using the Data Type Assistant” on page 13-14
- “Displaying Port Data Types” on page 13-23
- “Data Type Propagation” on page 13-24
- “Data Typing Rules” on page 13-24
- “Typecasting Signals” on page 13-25

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 Real-Time Workshop 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 13-24). 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 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
<code>uint16</code>	Unsigned 16-bit integer
<code>int32</code>	Signed 32-bit integer
<code>uint32</code>	Unsigned 32-bit integer

Besides the built-in types, Simulink defines a `boolean` (1 or 0) type, instances of which are represented internally by `uint8` values.

Many Simulink blocks also support fixed-point data types. See “Blocks — Alphabetical List” in the online documentation for information on the data types supported by specific blocks for parameter and input and output values. 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 Simulink® Fixed Point™ 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 Simulink Fixed Point 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 13-5 for details.

If you do not have the Simulink Fixed Point 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 Simulink Fixed Point 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 13-18 .

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Simulink Fixed Point product. However, even if you do not have the Simulink Fixed Point product, you can use the tool’s data type override mode to simulate a model that specifies fixed-point data types.

Data type override mode allows you to share fixed-point models with people in your company who do not have the Simulink Fixed Point product. In data type override mode, the fixed-point values are replaced with floating-point values when executing the model. Such replacement does not affect Fixed-Point Toolbox™ `fi` objects used as fixed-point parameters in your model. However, you can prevent the checkout of a Fixed-Point Toolbox license by setting the `fipref DataTypeOverride` property to `TrueDoubles` (see the Fixed-Point Toolbox documentation).

To simulate a model without using Simulink Fixed Point:

- 1** From the Simulink **Tools** menu, select **Fixed-Point > Fixed-Point Tool**.

The Fixed-Point Tool appears.

- 2** Set the **Fixed-point instrumentation mode** parameter to `Force off`.
- 3** Set the **Data type override** parameter to either `True doubles` or `True singles`.

Enumerated Data

An *enumerated data type* is a user-defined type whose values are strings that belong to a predefined set of strings. When the data type of an entity is an enumerated type, the value of the entity must be one of the strings that comprise the type. For information about enumerated data types, see Chapter 14, “Using Enumerated Data”. Do not confuse enumerated data types with enumerated property types (see “Enumerated Property Types” on page 13-53).

Block Support for Data and Numeric 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. See Simulink Blocks in *Simulink Reference* for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

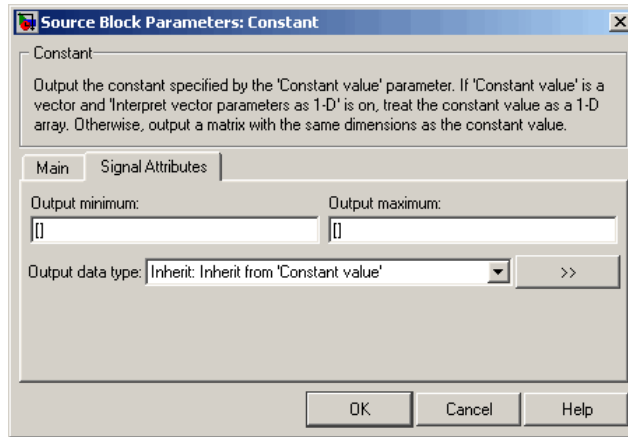
Creating Signals of a Specific Data Type

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.

Specifying 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:

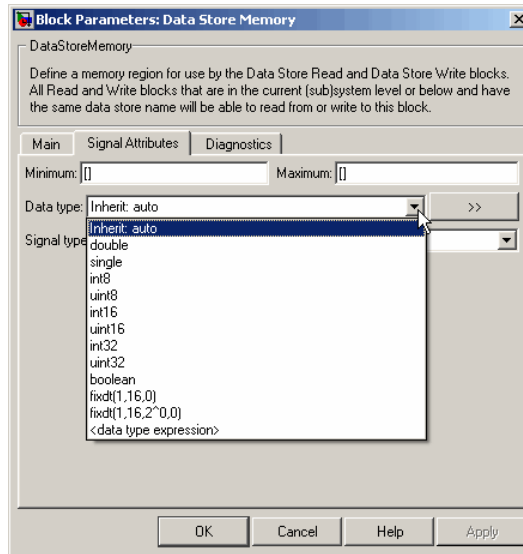
For Information About...	See...
Valid data type values that you can specify	“Entering Valid Data Type Values” on page 13-8
An assistant that helps you specify valid data type values	“Using the Data Type Assistant” on page 13-14
Specifying valid data type values for multiple blocks simultaneously	“Using the Model Explorer for Batch Editing” on page 13-11

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 13-9)
- The name of a built-in data type (see “Built-In Data Types” on page 13-10)
- An expression that evaluates to a data type (see “Data Type Expressions” on page 13-10)

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 Reference*.

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 13-24). 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 the accuracy and precision of the block output signal.

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 13-3 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 in the *Simulink Reference* 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

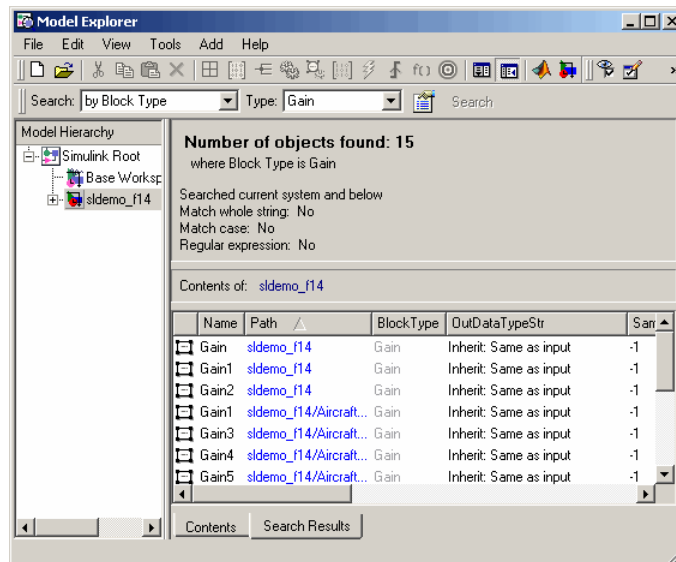
allow you to use custom aliases for data types. See “Working with Data Objects” on page 13-26 for more information about Simulink data objects.

Using the Model Explorer for Batch Editing

Using the Model Explorer (see “The Model Explorer” on page 19-2), you can assign the same output data type to multiple blocks simultaneously. For example, the `sldemo_f14` 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:

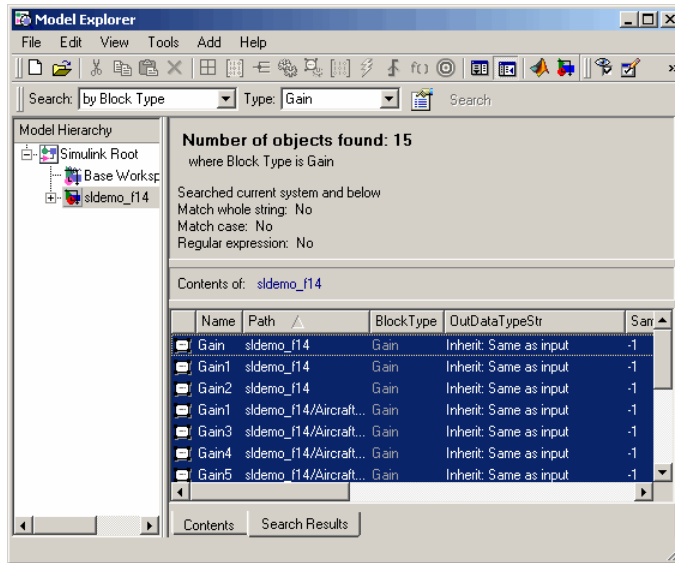
- 1 Use the Model Explorer search bar (see “Search Bar” on page 19-14) to identify all blocks in the `sldemo_f14` model of type Gain.

The Model Explorer **Contents** pane lists all Gain blocks in the model.



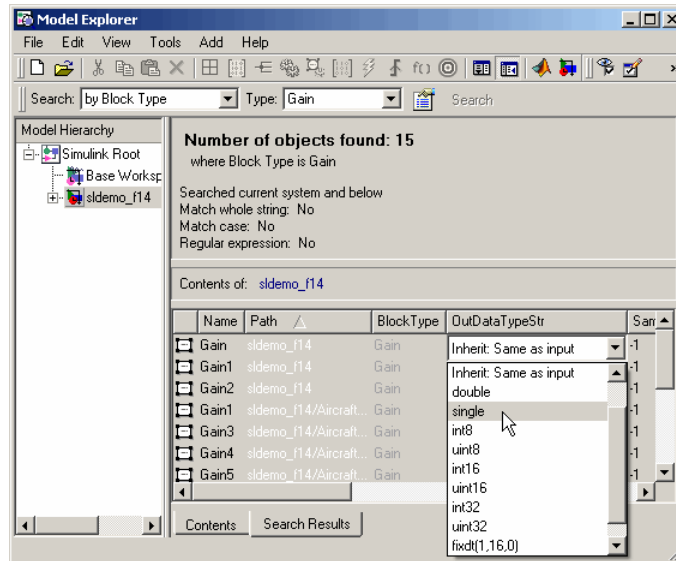
- 2 In the Model Explorer **Contents** pane, select all the Gain blocks whose **Output data type** parameter you want to specify.

Model Explorer highlights the rows corresponding to your selections.



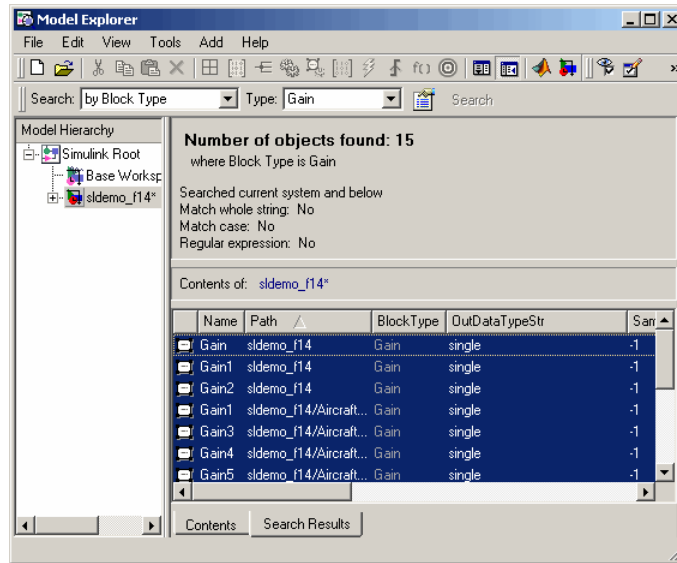
- 3 In the Model Explorer **Contents** pane, click the data type associated with one of the selected Gain blocks.

Model Explorer displays a pull-down menu with valid data type options.



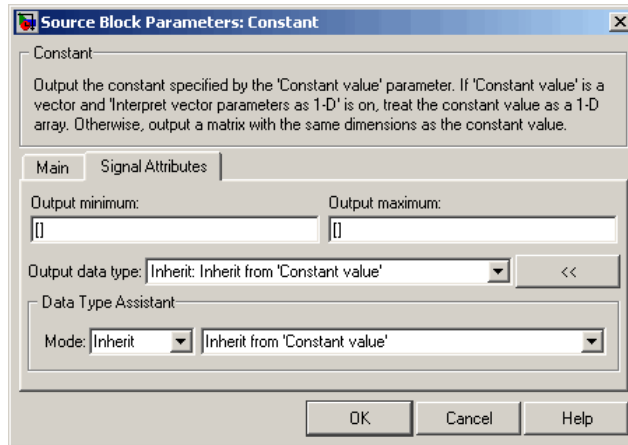
- 4 In the pull-down menu, enter or select the desired data type, for example, single.

Model Explorer specifies the data type of all selected items as single.

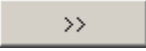
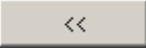


Using the Data Type Assistant

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:

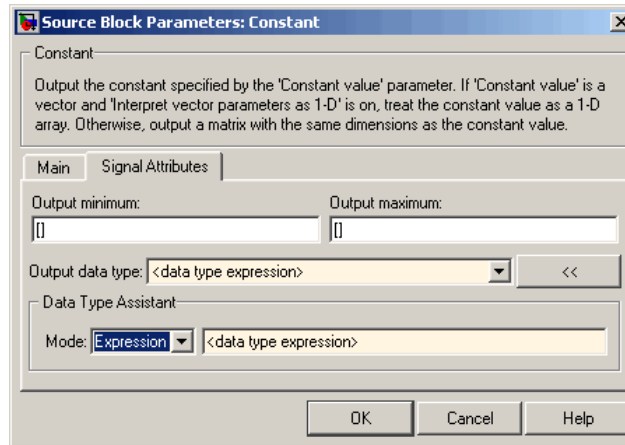
- Click the **Show data type assistant** button  to display the assistant.
- Click the **Hide data type assistant** button  to hide a visible assistant.

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
Expression	Expressions that evaluate to data types
Bus Object	Simulink.Bus object name, valid only for the Embedded MATLAB™ Function block and the Stateflow Chart block

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.

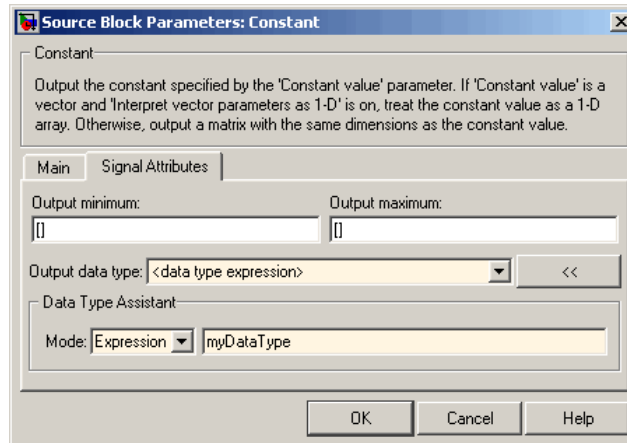


- 2 In the field that is to the right of the **Mode** field, select or enter a data type.

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:

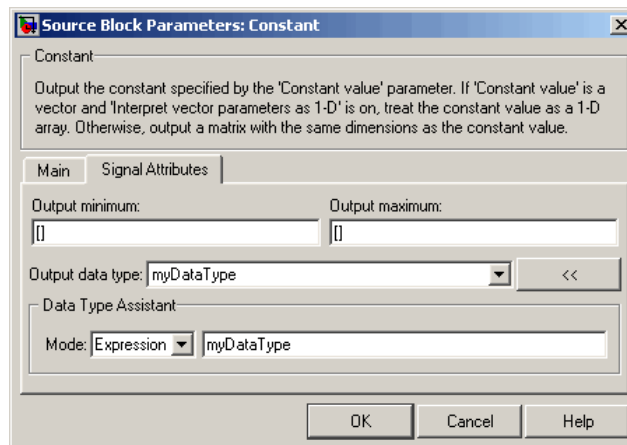
```
myDataType = Simulink.AliasType
myDataType.BaseType = 'single'
```

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.

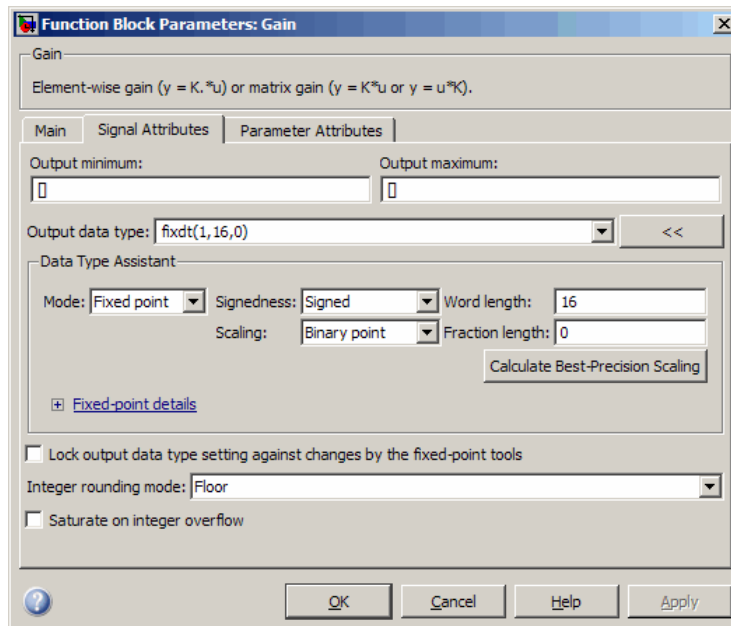


For more information about the data types that you can specify using the **Data Type Assistant**, see “Entering Valid Data Type Values” on page 13-8. See “Specifying Fixed-Point Data Types with the Data Type Assistant” in the

Simulink Fixed Point User's Guide for details about specifying fixed-point data types.

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 Concepts” in the *Simulink Fixed Point User's Guide*. 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:

Output data type: `fixdt(1,16,2^0,0)` <<

Data Type Assistant

Mode: Fixed point Signedness: Signed Word length: 16

Scaling: Slope and bias Slope: 2^0

Bias: 0

Calculate Best-Precision Scaling

[Fixed-point details](#)

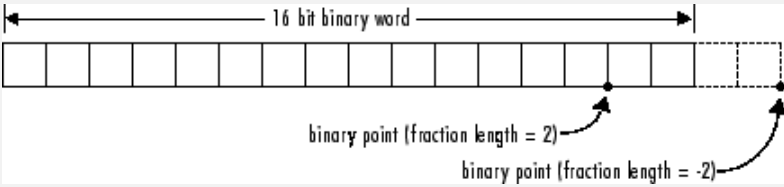
You can use the Data Type Assistant to set these fixed-point properties:

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>

Scaling Mode	Description
	 <p>The diagram shows a horizontal row of 16 small squares representing bits. Above the squares, a double-headed arrow spans the entire row and is labeled "16 bit binary word". Below the squares, two arrows point to specific bit positions. The first arrow points to the 3rd bit from the right and is labeled "binary point (fraction length = 2)". The second arrow points to the 15th bit from the right 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” in the *Simulink Fixed Point User’s Guide*.

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” in the *Simulink Fixed Point User’s Guide*.

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. See “Scaling” in the *Simulink Fixed Point User’s Guide* for instructions on autoscaling fixed-point data.

Showing Fixed-Point Details. When you specify a fixed-point data type, you can use the **Fixed-point details** subpane to see information about the fixed-point data type that is currently displayed in the Data Type Assistant. To see the subpane, click the expander next to **Fixed-point details** in the Data Type Assistant. The **Fixed-point details** subpane appears at the bottom of the Data Type Assistant:

The screenshot shows the Data Type Assistant interface. At the top, there are two input fields for "Output minimum:" and "Output maximum:", both containing empty boxes. Below these is a dropdown menu for "Output data type:" set to "fixdt(1,16,0)". The main section is titled "Data Type Assistant" and contains several settings: "Mode:" set to "Fixed point", "Signedness:" set to "Signed", "Word length:" set to "16", "Scaling:" set to "Binary point", and "Fraction length:" set to "0". There is a "Calculate Best-Precision Scaling" button. Below these settings is an expandable section for "Fixed-point details" which is currently expanded. It displays the following values: "Representable maximum:" 32767, "Output maximum:" (empty box), "Output minimum:" (empty box), "Representable minimum:" -32768, and "Precision:" 1. A "Refresh Details" button is located at the bottom right of this section.

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 “Checking Signal Ranges” on page 10-30 and “Checking Parameter Values” on page 7-19 for more information.

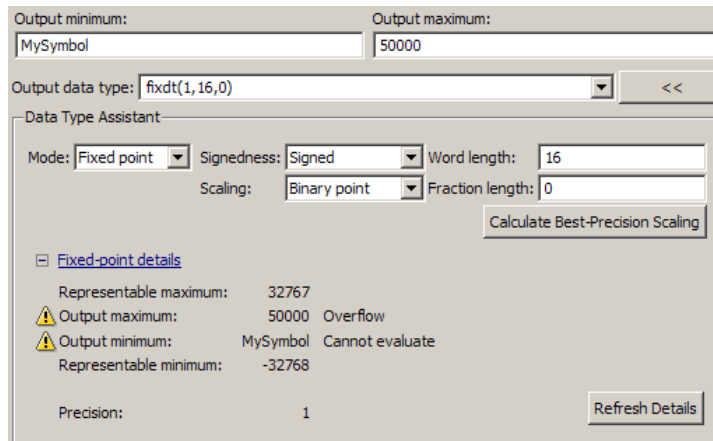
The rows labeled **Representable minimum**, **Representable maximum**, and **Precision** always appear. These rows show the minimum value, maximum value, and precision that can be represented by the fixed-point data type

currently displayed in the Data Type Assistant. See “Fixed-Point Concepts” in the *Simulink Fixed Point User’s Guide* for information about these three quantities.

The values displayed by the **Fixed-point details** subpane *do not* automatically update if you click **Calculate Best-Precision Scaling**, or change the range limits, the values that define the fixed-point data type, or anything elsewhere in the model. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**. The Data Type Assistant then updates or recalculates all values and displays the results.

Clicking **Refresh Details** does not change anything in the model, it only changes the display. Click **OK** or **Apply** to put the displayed values into effect. If the value of a field cannot be known without first compiling the model, the **Fixed-point details** subpane shows the value as Unknown.

If any errors occur when you click **Refresh Details**, the **Fixed-point details** subpane shows an error flag on the left of the applicable row, and a description of the error on the right. For example, the next figure shows two errors:

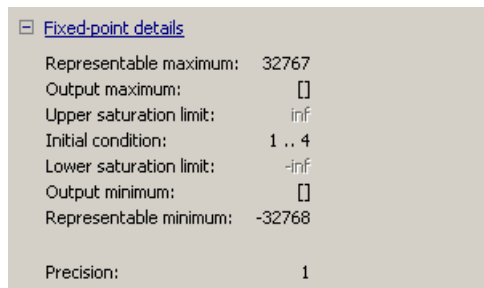


The row labeled **Output minimum** shows the error **Cannot evaluate** because evaluating the expression **MySymbol**, specified in the **Output minimum** field, did not return an appropriate numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane displays the unevaluated expression (truncating to 10 characters if necessary to save space) in place of the unavailable value.

To correct the error in this case, you would need to define `MySymbol` in an accessible workspace to provide an appropriate numeric value. After you clicked **Refresh Details**, the value of `MySymbol` would appear in place of its unevaluated text, and the error indicator and error description would disappear.

To correct the error shown for `Output maximum`, you would need to decrease **Output maximum**, increase **Word length**, or decrease **Fraction length** (or some combination of these changes) sufficiently to allow the fixed-point data type to represent the maximum value that it could have.

Other values relevant to a particular block can also appear in the **Fixed-point details** subpane. For example, on a Discrete-Time Integrator block's **Signal Attributes** tab, the subpane could look like this:



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.

Displaying Port Data Types

To display the data types of ports in your model, select **Format > Port/Signal Displays > 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 “Typecasting Signals” on page 13-25 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.

Typecasting 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.

Working with Data Objects

In this section...

“About Data Object Classes” on page 13-26

“About Data Object Methods” on page 13-27

“Using the Model Explorer to Create Data Objects” on page 13-29

“About Object Properties” on page 13-31

“Changing Object Properties” on page 13-31

“Handle Versus Value Classes” on page 13-33

“Comparing Data Objects” on page 13-35

“Saving and Loading Data Objects” on page 13-35

“Using Data Objects in Simulink Models” on page 13-36

“Creating Persistent Data Objects” on page 13-36

“Data Object Wizard” on page 13-36

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. For information on working with specific kinds of data objects, see “Simulink Classes”.

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 (see “Simulink Classes” for information on these built-in classes). Some of The MathWorks products based on Simulink, such as the Real-Time Workshop 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 “Subclassing Simulink Data Classes” on page 13-40).

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 13-33).

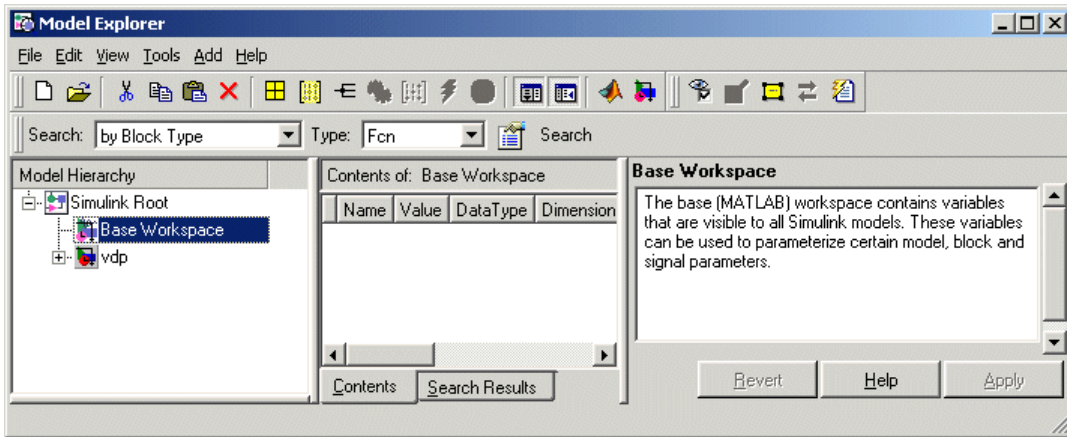
Using the Model Explorer to Create Data Objects

You can use the Model Explorer (see “The Model Explorer” on page 19-2) as well as MATLAB commands to create data objects. To use the Model Explorer,

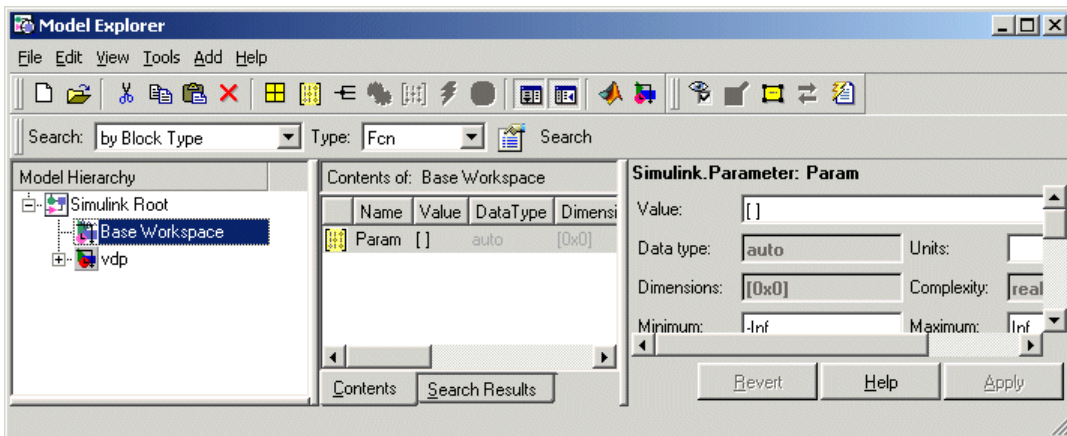
- 1 Select the workspace in which you want to create the object in the Model Explorer's **Model Hierarchy** pane.

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.

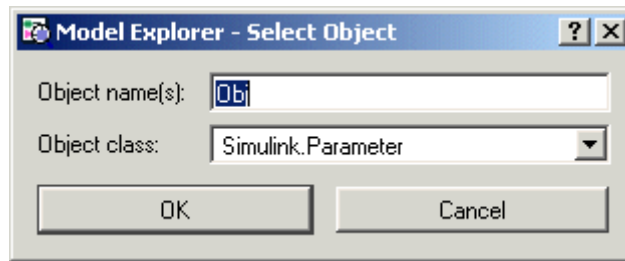
Note Subclasses of `Simulink.Parameter` and `Simulink.Signal` classes, such as `mpt.Parameter` and `mpt.Signal` objects (Real-Time Workshop Embedded Coder license required), can reside in a model workspace only if their storage class is set to `Auto`.



- 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.

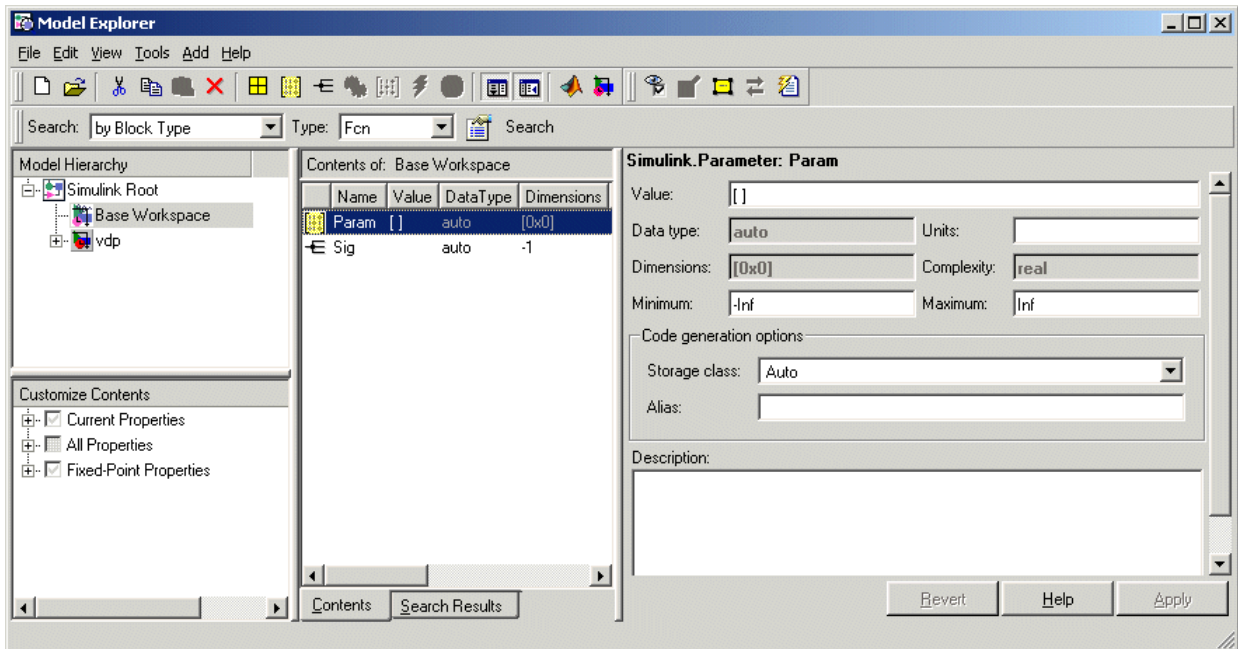
Changing Object Properties

You can use either the Model Explorer (see “Using the Model Explorer to Change an Object's Properties” on page 13-31) or MATLAB commands to change a data object's properties (see “Using MATLAB Commands to Change Workspace Data” on page 4-69).

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 object's properties in the **Contents** pane (see "Customizing the Contents Pane" on page 19-8). 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 13-33), `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.DataType = '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.RTWInfo.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 13-29) 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: []  
    RTWInfo: [1x1 Simulink.ParamRTWInfo]  
Description: ''  
    DataType: 'auto'  
        Min: -Inf  
        Max: Inf  
    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 `isContentEqual`, 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 `isContentEqual` method to verify that the object properties of A and B are equal.

```
>> result = A.isContentEqual(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 the objects or create the objects with the Simulink **Data Class Designer** (see “Subclassing Simulink Data Classes” on page 13-40) or at the command line and save them in a MAT-file (see “Saving and Loading Data Objects” on page 13-35). 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.

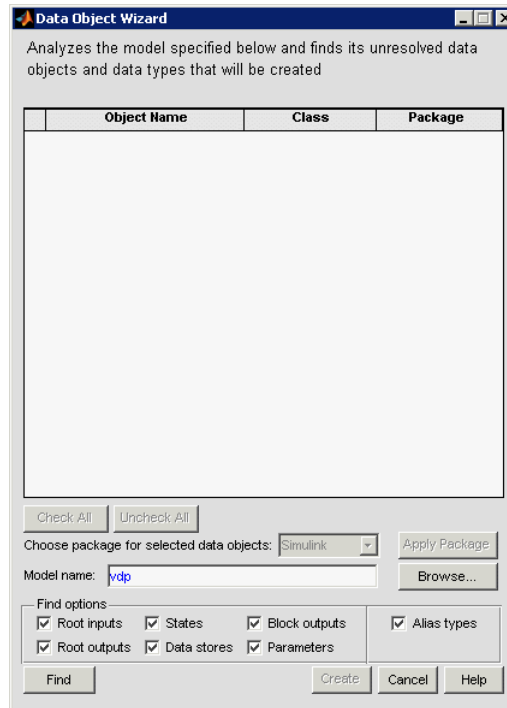
Data Object Wizard

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 Select **Tools > Data Object Wizard** from the Model Editor’s tool bar.

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.

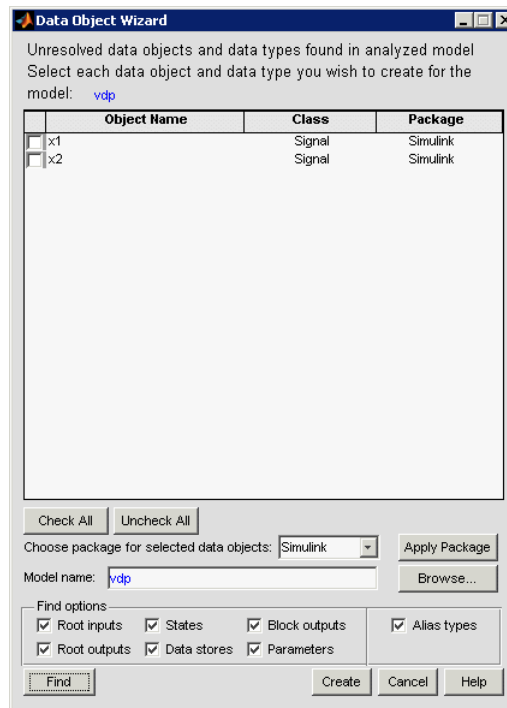
The search options include:

Option	Description
Root inputs	Named signals from root-level input ports

Option	Description
Root outputs	Named signals from root-level output ports
States	<p>States associated with any instances of the following discrete block types:</p> <ul style="list-style-type: none"> Discrete Filter Discrete State-Space Discrete-Time Integrator Discrete Transfer Fcn Discrete Zero-Pole Memory Unit Delay
Data stores	Data stores (see Chapter 16, “Working with Data Stores”)
Block outputs	Named signals emitted by non-root-level blocks.
Parameters	<ul style="list-style-type: none"> • Parameters of any instances of the following block types: <ul style="list-style-type: none"> Constant Gain Lookup Table Lookup Table (2-D) Relay • Stateflow data with a Scope of Parameter. <p>See “Sharing Simulink Parameters with Stateflow Charts” in the online Stateflow documentation for more information.</p>
Alias types	Data whose data type is a registered custom data type. This option applies only if you are generating code from the model. See “Creating Simulink Data Objects with Data Object Wizard” in the Real-Time Workshop Embedded Coder documentation for more information.

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.
- 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.

Subclassing Simulink Data Classes

In this section...

“About Packages and Data Classes” on page 13-40

“Working with Packages” on page 13-41

“Working with Classes” on page 13-45

“Enumerated Property Types” on page 13-53

“Enabling Custom Storage Classes” on page 13-56

About Packages and Data Classes

Simulink packages and data classes are introduced in “About Data Object Classes” on page 13-26 and “Defining Data Representation and Storage for Code Generation”. Simulink resources include several built-in packages and data classes. You can add user-defined packages and data classes with the Simulink **Data Class Designer**, which can:

- Create and delete user-defined packages that can hold user-defined subclasses.
- Create, change, and delete user-defined subclasses of some Simulink classes.
- Add and delete user-defined subclasses contained within user-defined packages.
- Define enumerated property types (not data types) for use by classes in a package.
- Enable defining custom storage classes for classes that have an appropriate parent class.

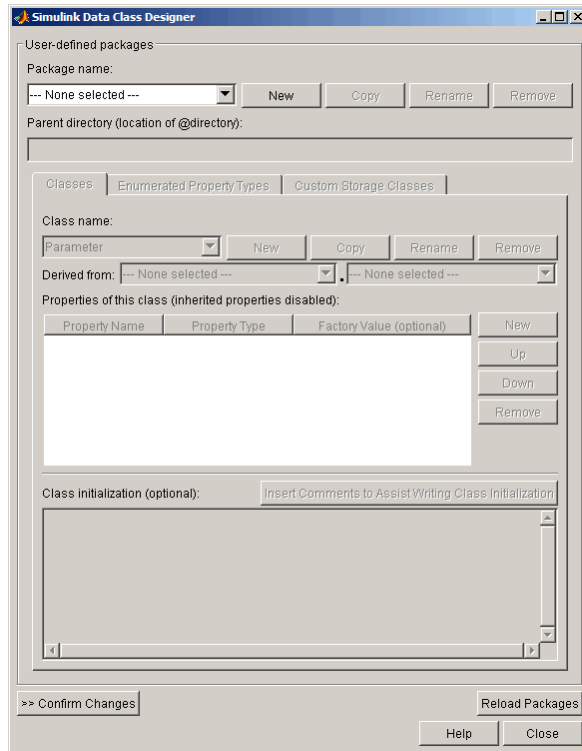
Simulink stores all package and data class definitions as P-files. You can view any package or data class in the Data Class Designer, but you cannot use the Designer to change a built-in package or data class. If you create or change user-defined packages and data classes, do so *only* in the Data Class Designer. Do not try to circumvent the Data Class Designer and directly modify any package or data class definition. An unrecoverable error could result.

Appropriately configured packages and data classes can define custom storage classes, which specify how data is declared, stored, and represented in generated code. Some built-in packages and data classes define built-in custom storage classes. You can use the Data Class Designer to enable a user-defined package and data class to have custom storage classes, as described in “Enabling Custom Storage Classes” on page 13-56.

Custom storage classes are defined and stored in different files than packages and data classes use, and different rules apply to changing them. See “Creating and Using Custom Storage Classes” in the Real-Time Workshop Embedded Coder documentation for complete information about custom storage classes.

Working with Packages

You can use the Simulink Data Class Designer to create and manage user-defined packages. To view the Data Class Designer, choose **Tools > Data Class Designer** from the Simulink menu. When opened, the designer first scans the MATLAB path and loads any packages that it finds. The Data Class Designer looks like this when no package is currently selected:



You can use the fields and buttons at the top and bottom of the Data Class Designer, and a few other designer capabilities, to select an existing package, create a new package, and copy, rename, or delete a package, as described in this section. The rest of the designer manages classes, as described in “Working with Classes” on page 13-45.

Selecting a Package

To select an existing user-defined package, select that package in the **Package Name** field of the Simulink **Data Class Designer**. The designer then displays the location of the package and the classes that it contains, plus class initialization and other information.

Creating a Package

To create a new package to contain your classes:

- 1 Click the **New** button next to the **Package name** field of the Data Class Designer.

Package name:

SimulinkDemos

Parent directory (location of @directory):

c:\matlab\toolbox\simulink\simdemos

A default package name is displayed in the **Package name** field.

Package name:

NewPackage1

Parent directory (location of @directory):

- 2 Edit the **Package name** field to contain the package name that you want.

Package name:

MyData

Parent directory (location of @directory):

- 3 Click **OK** to create the new package in memory.
- 4 In the package **Parent directory** field, enter the path of the directory where you want Simulink to create the new package.

Note Do not create class package directories under *matlabroot*. Packages in these directories are treated as built-in and will not be visible in the Data Class Designer.

Package name:

MyData

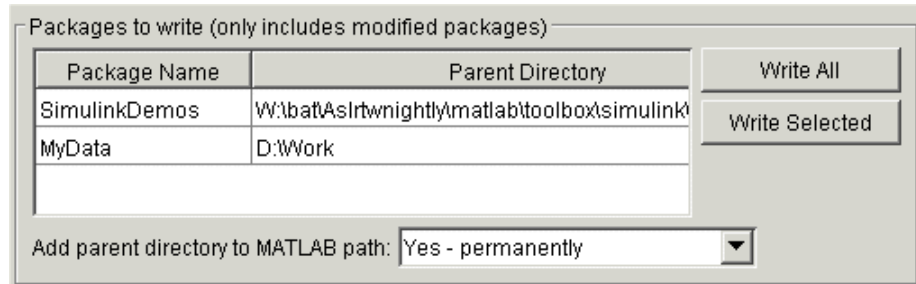
Parent directory (location of @directory):

d:\Work

The Simulink software creates the specified directory, if it does not already exist, when you save the package to your file system in the succeeding steps.

- 5 Click the **Confirm changes** button on the **Data Class Designer**.

The **Packages to write** panel is displayed.



- 6 To enable use of this package in the current and future sessions, ensure that the **Add parent directory to MATLAB path** option is set to Yes - permanently. The default is Yes - for this session only.

This adds the path of the new package's parent directory to the MATLAB path.

- 7 Click **Write all** or select the new package and click **Write selected** to save the new package.

Copying a package

To copy a package, select the package and click the **Copy** button next to the **Package name** field. The Simulink software creates a copy of the package under a slightly different name. Edit the new name, if desired, and click **OK** to create the package in memory. Then save the package to make it permanent.

Renaming a package

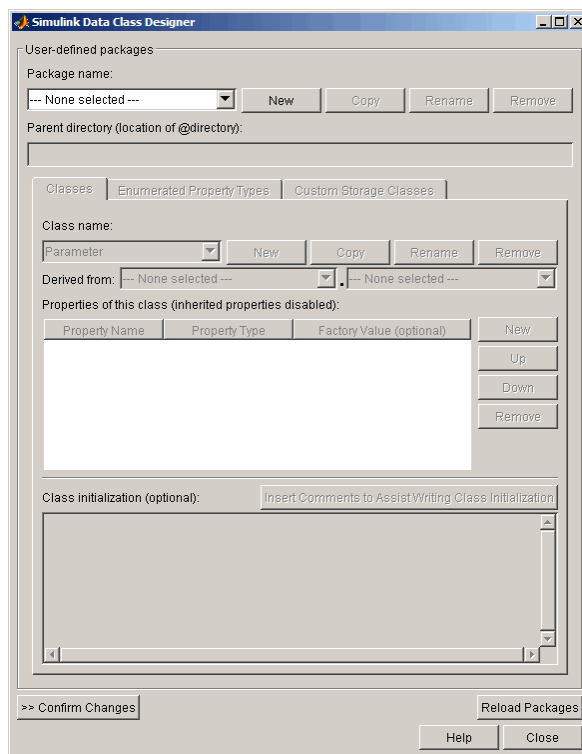
To rename a package, select the package and click the **Rename** button next to the **Package name** field. The field becomes editable. Edit the field to reflect the new name. Save the renamed package.

Removing a package

To remove a package, select the package and click the **Remove** button next to the **Package name** field to remove the package from memory. Click the **Confirm changes** button to display the **Packages to remove** panel. Select the package and click **Remove selected** to remove the package from your file system or click **Remove all** to remove all packages that you have removed from memory from your file system as well.

Working with Classes

You can use the Simulink Data Class Designer to create and manage user-defined classes. To view the Data Class Designer, choose **Tools > Data Class Designer** from the Simulink menu. When opened, the designer first scans the MATLAB path and loads any packages that it finds. The Data Class Designer looks like this when no package is currently selected:



You can use the three tabs in the Data Class Designer, and a few other designer capabilities, to select an existing class, create a new class, copy, rename, or delete a class, and specify various class properties, as described in this section. The rest of the designer manages packages, as described in “Working with Classes” on page 13-45.

Creating a Data Object Class

To create a class within the currently selected package:

- 1** Select **Data Class Designer** from the Simulink **Tools** menu.

The **Data Class Designer** dialog box appears.

The designer initially scans the MATLAB path and loads any packages that it finds.

- 2** Select the name of the package in which you want to create the class from the **Package name** list.

Do not create a class in any of the Simulink built-in packages, i.e., packages in *matlabroot/toolbox/simulink*, or any directory under *matlabroot*. See “Working with Packages” on page 13-41 for information on creating your own class packages.

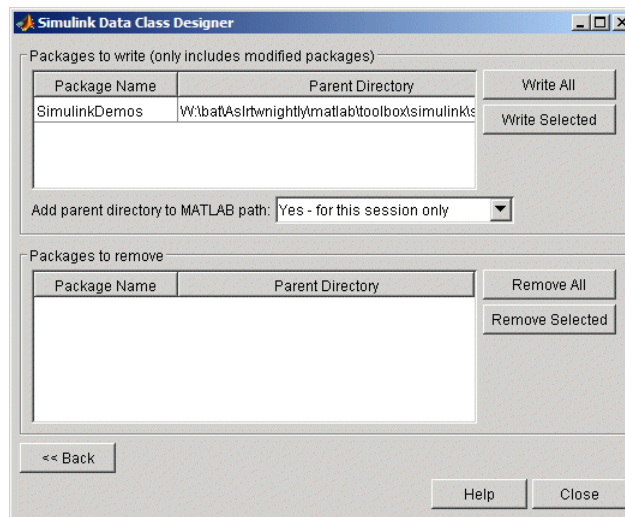
- 3** Click the **New** button on the **Classes** pane of the **Data Class Designer** dialog box.
- 4** Enter the name of the new class in the **Class name** field on the **Classes** pane.

Note The name of the new class must be unique in the package to which the new class belongs. Class names are case sensitive. For example, **Signal** and **signal** is considered to be names of different classes.

- 5** Press **Enter** or click **OK** on the **Classes** pane to create the specified class in memory.

- 6 Select a parent class for the new class (see “Specifying a Parent for a Class” on page 13-48).
- 7 Optionally enable custom storage classes for the class (see “Enabling Custom Storage Classes” on page 13-56).
- 8 Define the properties of the new class (see “Defining Class Properties” on page 13-50).
- 9 If necessary, create initialization code for the new class (see “Creating Initialization Code” on page 13-52).
- 10 Click **Confirm Changes**.

Simulink displays the **Confirm Changes** pane.



- 11 Click **Write All** or select the package containing the new class definition and click **Write Selected** to save the new class definition.

Copying a class

To copy a class, select the class in the **Classes** pane and click **Copy**. The Simulink software creates a copy of the class under a slightly different name.

Edit the name, if desired, click **Confirm Changes**, and click **Write All** or, after selecting the appropriate package, **Write Selected** to save the new class.

Renaming a class

To rename a class, select the class in the **Classes** pane and click **Rename**. The **Class name** field becomes editable. Edit the field to reflect the new name. Save the package containing the renamed class, using the **Confirm changes** pane.

Note You cannot modify an instantiated Simulink data class. If you try to modify a class that has already been instantiated during the current MATLAB session, an error message informs you that you need to restart MATLAB if you want to modify this package.

Removing a class from a package

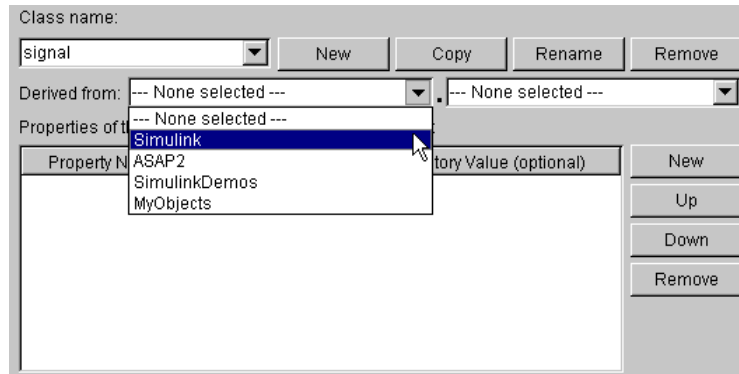
To remove a class definition from the currently selected package, select the class in the **Classes** pane and click **Remove**. The class is removed from the in-memory definition of the class. Save the package that formerly contained the class.

Specifying a Parent for a Class

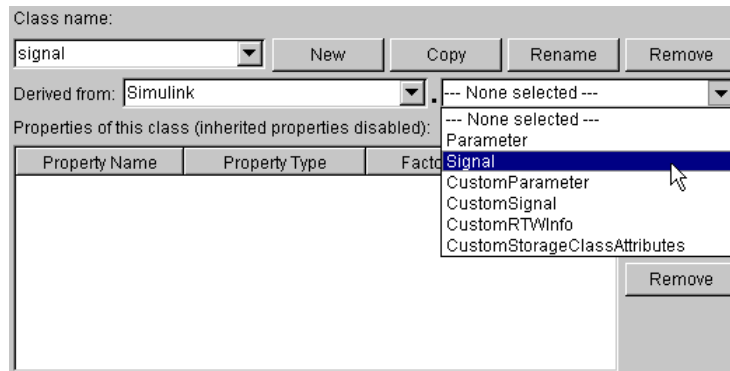
To specify a parent for a class:

- 1 Select the name of the class from the **Class name** field on the **Classes** pane.

- 2** Select the package name of the parent class from the left-hand **Derived from** list box.



- 3** Select the parent class from the right-hand **Derived from** list.



The properties of the selected class derived from the parent class are displayed in the **Properties of this class** field.



The inherited properties are grayed to indicate that they cannot be redefined by the child class.

- 4 Save the package containing the class.

Defining Class Properties

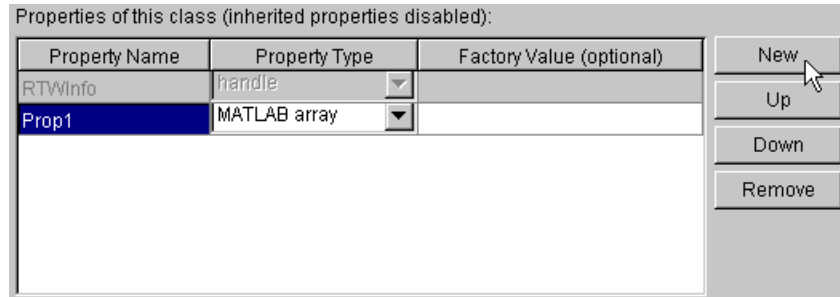
To add a property to a class:

- 1 Select the name of the class from the **Class name** field on the **Classes** pane.

Note You cannot modify an instantiated Simulink data class. If you try to modify a class that has already been instantiated during the current MATLAB session, an error message informs you that you need to restart MATLAB if you want to modify this package.

- Click the **New** button next to the **Properties of this class** field on the **Classes** pane.

A property is created with a default name and value and displays the property in the **Properties of this class** field.



- Enter a name for the new property in the **Property Name** column.

Note The property name must be unique to the class. Unlike class names, property names are not case sensitive. For example, `Value` and `value` are treated as referring to the same property.

- Select the data type of the property from the **Property Type** list.

The list includes built-in property types and any enumerated property types that you have defined (see “Enumerated Property Types” on page 13-53).

- If you want the property to have a default value, enter the default value in the **Factory Value** column.

The default value is the value the property has when an instance of the associated class is created. The initialization code for the class can override this value (see “Creating Initialization Code” on page 13-52 for more information).

The following rules apply to entering factory values for properties:

- Do not use quotation marks when entering the value of a string property. The value that you enter is treated as a literal string.

- The value of a MATLAB array property can be any expression that evaluates to an array, cell array, structure, or object. Enter the expression exactly as you would enter the value on the command line, for example, `[0 1; 1 0]`. The expression that you enter is evaluated to check its validity. A warning is displayed if evaluating the expression results in an error. Regardless of whether an evaluation error occurs, Simulink stores the expression as the factory value of the property. This is because an expression that is invalid at define time might be valid at run-time.
- You can enter any expression that evaluates to a numeric value as the value of a `double` or `int32` property. The expression is evaluated and the result stored as the property's factory value.

6 Save the package containing the class with new or changed properties.

Creating Initialization Code

You can specify code to be executed when the Simulink software creates an instance of a data object class. To specify initialization code for a class, select the class from the **Class name** field of the **Data Class Designer** and enter the initialization code in the **Class initialization** field.

The **Data Class Designer** inserts the code that you enter in the **Class initialization** field in the class instantiation function of the corresponding class. This function is invoked when an instance of this class is created. The class instantiation function has the form

```
function h = ClassName(varargin)
```

where `h` is the handle to the object that is created and `varargin` is a cell array that contains the function's input arguments.

By entering the appropriate code in the **Data Class Designer**, you can cause the instantiation function to perform such initialization operations as

- Error checking
- Loading information from data files
- Overriding factory values
- Initializing properties to user-specified values

For example, suppose you want to let a user initialize the `ParamName` property of instances of a class named `MyPackage.Parameter`. The user does this by passing the initial value of the `ParamName` property to the class constructor:

```
Kp = MyPackage.Parameter('Kp');
```

The following code in the instantiation function would perform the required initialization:

```
switch nargin
    case 0
        % No input arguments - no action
    case 1
        % One input argument
        h.ParamName = varargin{1};
    otherwise
        warning('Invalid number of input arguments');
end
```

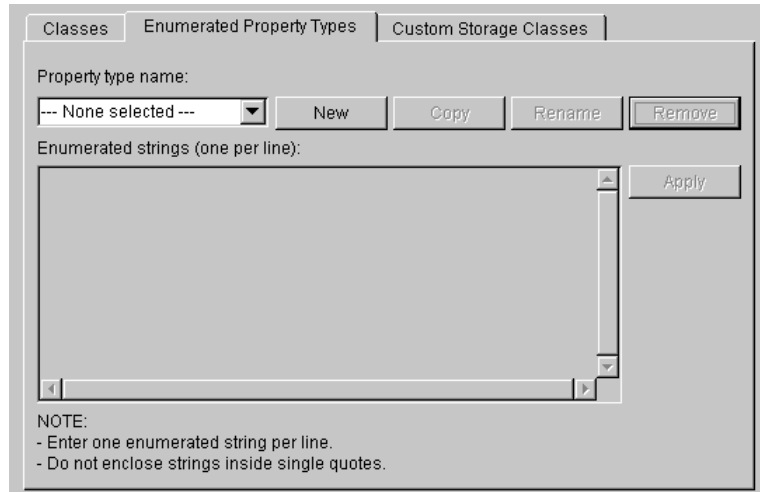
Enumerated Property Types

An *enumerated property type* is a property type whose value must be one of a specified set of values, for example, red, blue, or green. An enumerated property type is valid only in the package that defines it, but must be globally unique throughout all packages.

Note Do not confuse an *enumerated property type* with an *enumerated data type*. For information on enumerated data types, see “Using Enumerated Data in a Simulink Model” on page 14-11. The Data Class Designer cannot be used for defining enumerated data types.

To create an enumerated property type:

- 1 Select the **Enumerated Property Types** tab of the **Data Class Designer**.



- 2 Click the **New** button next to the **Property type name** field.

An enumerated property type is created with a default name.



- 3 Change the default name in the **Property type name** field to the desired name for the property.

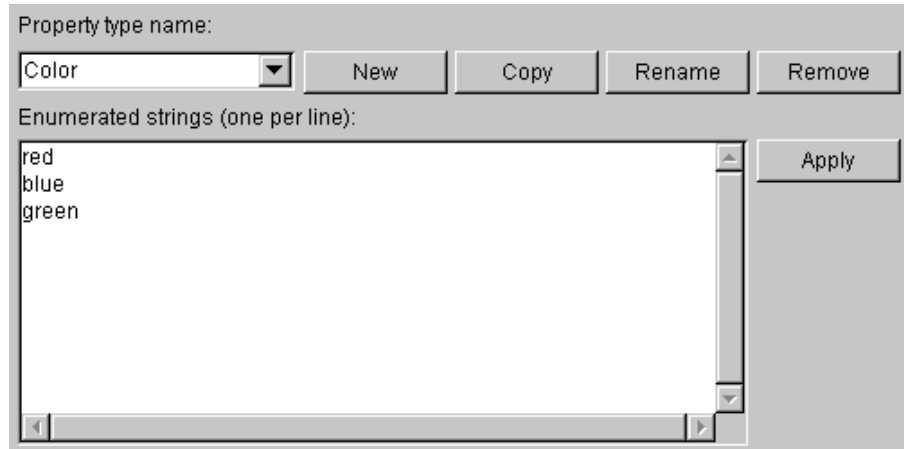
The currently selected package defines an enumerated property type and the type can be referenced only in the package that defines it. However, the name of the enumerated property type must be globally unique. There cannot be any other built-in or user-defined enumerated property type with the same name. An error is displayed if you enter the name of an existing built-in or user-defined enumerated property type.

- 4 Click the **OK** button.

The new property is created in memory and the **Enumerated strings** field on the **Enumerated Property Types** pane is enabled .

- 5** Enter the permissible values for the enumerated property type **Enumerated strings** field, one per line.

For example, the following **Enumerated strings** field shows the permissible values for an enumerated property type named **Color**.



- 6** Click **Apply** to save the changes in memory.
- 7** Click **Confirm changes**. Then click **Write all** to save this change.

You can also use the **Enumerated Property Type** pane to copy, rename, and remove enumerated property types.

- Click the **Copy** button to copy the currently selected property type. A new property that has a new name, but has the same value set as the original property is created.
- Click the **Rename** button to rename the currently selected property type. The **Property name** field becomes editable. Edit the field to reflect the new name.
- Click the **Remove** button to remove the currently selected property.

Always save the package containing the modified enumerated property type.

Note You must restart the MATLAB software if you modify, add, or remove enumerated property types from a class you have already instantiated.

Enabling Custom Storage Classes

If you select `Simulink.Signal` or `Simulink.Parameter` as the parent of a user-defined class, the Simulink Data Class Designer displays a check box labeled **Create your own custom storage classes for this class**. You can ignore this option if you do not intend to use Real-Time Workshop Embedded Coder to generate code from models that reference this data object class.

Otherwise, select this check box to cause Simulink Data Class Designer to create custom storage classes for this data object class. See “Creating Packages that Support CSC Definitions” in the Real-Time Workshop Embedded Coder documentation for more information. The Data Class Designer also provides a tab labeled Custom Storage Classes, but this tab is now obsolete.

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.

Using Enumerated Data

- “Enumerated Data in Simulink” on page 14-2
- “Demos of Enumerated Data Types ” on page 14-3
- “Defining an Enumerated Data Type” on page 14-4
- “Changing an Enumerated Data Type” on page 14-10
- “Using Enumerated Data in a Simulink Model” on page 14-11
- “Simulink Constructs that Support Enumerated Types” on page 14-22
- “Simulink Enumerated Type Limitations” on page 14-25

Enumerated Data in Simulink

An *enumerated data type* is a user-defined type whose values are symbols that belong to a predefined set of symbols. When the data type of an entity is an enumerated type, the value of the entity must be one of the symbols that comprise the type. Enumerated types are often used to represent program states and to control program logic. Numeric types can be used to produce similar results, but enumerated types provide better human readability and stronger error checking.

Simulink supports enumerated data types, as do many other languages and development environments. For brevity, the rest of this chapter refers without qualification to enumerated types, classes, data, and values. The intended reference is specifically to the implementation of enumerated data in Simulink and other MathWorks™ products like Stateflow.

Like everything in Simulink, enumerated types are based on underlying MATLAB constructs. However, enumerated types differ fundamentally from all other user-defined data types supported by Simulink. While other Simulink data types exist as instances of MATLAB classes, an enumerated data type is itself a MATLAB class. The instances of that class are the specific values that comprise the enumerated type.

This chapter provides general information that applies to enumerated types in any context, plus additional information that is specific to Simulink. Information specific to enumerated types in Stateflow charts appears in “Using Enumerated Data in Stateflow Charts”. Information specific to Real-Time Workshop code that implements enumerated types appears in “Enumerated Data Type Considerations”.

Demos of Enumerated Data Types

Two demos exist that show you how to use enumerated types in Simulink and Stateflow:

Data Typing in Simulink — Shows how to use data types in Simulink, including enumerated data types. To see the demo, in the MATLAB Help browser **Contents** pane, choose **Simulink > Demos > Modeling Features > Data Typing in Simulink** .

Modeling a CD Player/Radio — Shows how to use enumerated data types in a Simulink model that contains a Stateflow chart. To see the demo, in the MATLAB Help browser **Contents** pane, choose **Stateflow > General Applications > Modeling a CD Player/Radio using Enumerated Data Types** .

Defining an Enumerated Data Type

In this section...
“Workflow for Defining Enumerated Data” on page 14-4
“Creating a Class Definition File” on page 14-5
“Enumerated Class Definition Syntax” on page 14-5
“Enumerated Class Definition Example” on page 14-5
“Defining Enumerated Values” on page 14-6
“Overriding Default Methods (Optional)” on page 14-7
“Defining Additional Customizations” on page 14-8
“Saving the M-File on the MATLAB Path” on page 14-8

Workflow for Defining Enumerated Data

This section provides complete instructions for defining an enumerated data type and making it available for use. You can use any applicable MATLAB technique when defining an enumerated type, but ordinarily you will need only the techniques described in this section. The essential steps are:

- 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 M-file on the MATLAB path.

For complete information about the MATLAB Object System see *Object-Oriented Programming*.

Note In the current release, you cannot define an enumerated type directly in the MATLAB Command Window, or by using any GUI tool such as the Simulink Data Class Designer. You can define an enumerated type only in an M-file that is saved on the MATLAB path, as described in this section.

Creating a Class Definition File

You can define an enumerated type (or any other MATLAB class) by creating an M-file that defines the class and saving that file anywhere on the MATLAB path. Each file can define only one class definition. To begin a new file, choose **File > New > Class M-File** in the MATLAB Command Window menu.

Enumerated Class Definition Syntax

The general syntax of an enumerated class definition is:

```
classdef(Enumeration) EnumTypeName < Simulink.IntEnumType
    enumeration
        enumerated value definitions
    end
    methods (Static = true)
        static method definitions
    end
end
```

The first line defines the class name, *EnumTypeName*. This name must be unique among data type names and workspace variable names, and is case-sensitive. Every Simulink enumerated class is a subclass of `Simulink.IntEnumType`, which is an abstract class that cannot be instantiated in its own right. Inheriting from `Simulink.IntEnumType` provides the capabilities necessary for the class to be used in Simulink.

The body of the class definition consists of two sections: the `enumeration` section and the `methods` section. The `enumeration` section defines the enumerated values that comprise the type, as described in “Defining Enumerated Values” on page 14-6. The `methods` section is optional, and if present defines static methods that override some default static methods inherited from `Simulink.IntEnumType`, as described in “Overriding Default Methods (Optional)” on page 14-7.

Enumerated Class Definition Example

The following is an example of an enumerated class definition. The rest of this chapter uses this example:

```
classdef(Enumeration) BasicColors < Simulink.IntEnumType
```

```
enumeration
  Red(0)
  Yellow(1)
  Blue(2)
end
methods (Static = true)
  function retVal = getDefaultValue()
    retVal = BasicColors.Blue;
  end
end
end
```

Defining Enumerated Values

An enumerated type can comprise any number of *enumerated values*. An enumerated value consists of two elements: an *enumerated name*, which the software displays where human readability is needed, and an *underlying integer*, which the software uses internally.

To define the enumerated values that comprise an enumerated type, you specify them in the class definition's `enumeration` section. The general syntax of an enumeration section is:

```
enumeration
  EnumName(N)
  ...
end
```

where *EnumName* is the an enumerated value's name, and *N* is the value's underlying integer. Each *EnumName* must be unique within its enumerated type. An *EnumName* can be any legal MATLAB identifier. You can use the function `isvarname` to determine whether a symbol is a legal MATLAB identifier and hence can be an *EnumName*.

The underlying integers in an enumeration section need not be unique within the type or across types. In many cases, the underlying integers of a set of enumerated values will be 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 obtain 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 “Configuring the Hardware Implementation” and “Hardware Implementation Pane” for more information.

Overriding Default Methods (Optional)

Every enumerated class has four associated static methods, which it inherits from *Simulink.IntEnumType*. You can optionally override any or all of these static methods to customize the behavior of an enumerated type. The methods are:

- `getDefaultValue` — Returns the default value of the enumerated data type.
- `getDescription` — Returns a description of the enumerated data type.
- `getHeaderFile` — Specifies a file where the type is defined for generated code.
- `addClassNameToEnumNames` — Specifies whether the class name becomes a prefix in code.

The first of these methods, `getDefaultValue`, is relevant to both simulation and code generation, and is described in this section. The other three methods are relevant only to code generation, and are described in “Overriding Default Methods (Optional)” in the Real-Time Workshop documentation.

To override any of the methods, include a customized version of the method in the enumerated class definition’s `methods` section. If you do not want to override any default methods, omit the `methods` section entirely.

Specifying a Default Enumerated Value

Simulink software and Real-Time Workshop generated code use an enumerated data type’s default value for ground-value initialization of enumerated data when no other initial value is provided. For example, an enumerated signal inside a conditionally executed subsystem that has not yet executed has the enumerated type’s default value. Generated code also uses an enumerated type’s default value if a safe cast fails, as described in “Enumerated Type Safe Casting” in the Real-Time Workshop documentation.

Unless you specify otherwise, the default value for an enumerated type is the lexically first value in the enumerated class definition. To specify a different default value, write your own `getDefaultValue` method and put it into 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 enumerated value 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 in that class.

The seemingly redundant specification of `ThisClass` inside the definition of that same class is necessary because `getDefaultValue` returns an instance of the enumerated class, not just the name of an enumerated value. See “Instantiating an Enumerated Type” on page 14-15 for more information.

Defining Additional Customizations

You can further customize an enumerated type using the same techniques that work with MATLAB classes generally, as described in *Object-Oriented Programming*, except that enumerated classes may not define properties.

Saving the M-File on the MATLAB Path

To make an enumerated data type available for simulation and code generation, save the M-file that defines it somewhere on the MATLAB path. You do not have to explicitly load the file: MATLAB will search the path when needed and find the definition.

The name of the M-file must match the name of the enumerated data type. The match is case-sensitive. For example, the definition of `BasicColors` must reside in an M-file named `BasicColors.m`. If the type were defined in a file

named `basiccolors.m`, or any other file whose name did not match perfectly, MATLAB would not find it.

To add a directory to the MATLAB search path, type `addpath pathname` at the MATLAB command prompt. For more information, see `addpath`, and `savepath` in the MATLAB documentation.

Changing an Enumerated Data Type

You can change the definition of an enumerated data type by editing and saving the M-file that defines it. You do not need to inform MATLAB that the definition has changed: it will read in the changes automatically. If no instances of the type exist, you can use the changed definition immediately. The result is the same as if the changed definition had been in effect when you started MATLAB.

If you have instantiated the type in MATLAB, or simulated or performed Update Diagram on any model that uses the type, you must delete all existing instances of the type before you can use the new definition. The MATLAB Command Window will display a warning about deleting the obsolete instances. To delete all instances of a changed enumerated type:

- 1** Delete the instances from the base workspace in one of these ways:
 - Locate the instances in the workspace and delete them individually.
 - Delete everything from the base workspace by executing `clear`.
- 2** Close all models that you simulated or updated while the previous type definition was in effect.

All subsequent operations that use the enumerated type will access the changed definition.

Using Enumerated Data in a Simulink Model

In this section...

“Simulating with Enumerated Types” on page 14-11

“Referencing an Enumerated Type” on page 14-14

“Instantiating an Enumerated Type” on page 14-15

“Displaying Enumerated Values” on page 14-17

“Enumerated Values in Computation” on page 14-18

Simulating with Enumerated Types

An enumerated data type definition resembles a list of named constants that equate to integer values, but the values in the enumerated type appear in a MATLAB class definition, and the integers are used only for internal operations. The following code defines an enumerated class (or type) named `BasicColors` whose enumerated values are `Red`, `Yellow`, and `Blue`, with `Blue` as the default value:

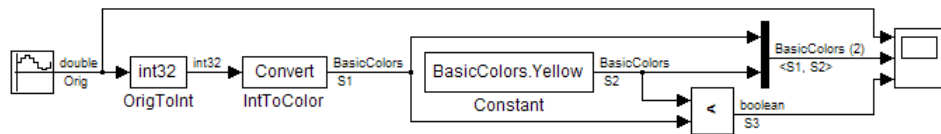
```
classdef(Enumeration) BasicColors < Simulink.IntEnumType
    enumeration
        Red(0)
        Yellow(1)
        Blue(2)
    end
    methods (Static = true)
        function retVal = getDefaultvalue()
            retVal = BasicColors.Blue;
        end
    end
end
```

This is an ordinary MATLAB class definition. The first line defines an integer-based enumerated data type that is derived from the built-in class `Simulink.IntEnumType`. The enumeration section specifies the enumerated values that comprise the type:

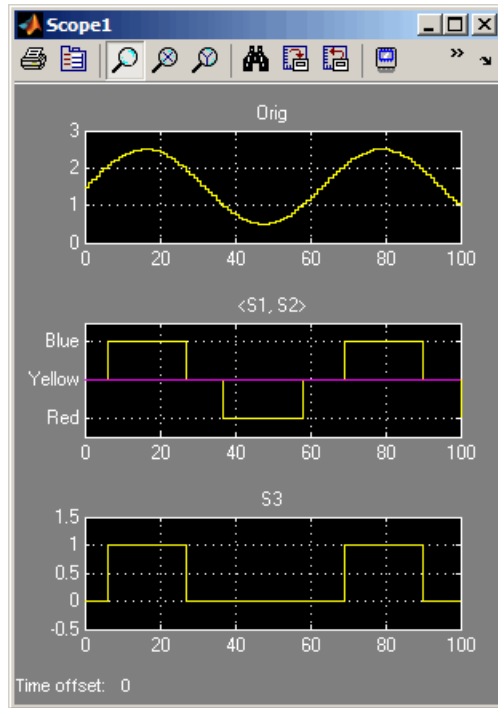
Enumerated Value	Enumerated Name	Underlying Integer Value
Red(0)	Red	0
Yellow(1)	Yellow	1
Blue(2)	Blue	2

The `methods` section is optional, and contains a `getDefaultValue` method that defines the default value for the type to be `BasicColors.Blue`. If that method did not appear, the default value would be `Red`, because that is the lexically first value in the enumerated class definition.

Once this class definition is known to MATLAB, you can use the enumerated type in Simulink and Stateflow models. The following model uses the enumerated type 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, which is shown 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.

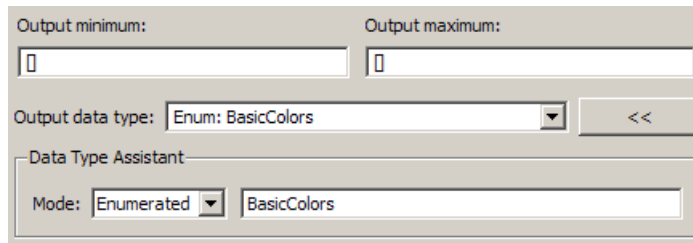
The result is a cycle of colors: Yellow, Blue, Yellow, Red, Yellow, Blue, Yellow, as shown in the second graph. The Constant block outputs Yellow, which appears in the second graph as a straight line. The Relational Operator block compares the constant Yellow to each value in the cycle of colors. It 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 order of the underlying integers of the compared values, *not* the order in which the enumerated values appear lexically in the enumerated class definition. In this case the two orders are

the same, but they need not be. See “Enumerated Values in Computation” on page 14-18 for more information.

Referencing an Enumerated Type

Once you have defined an enumerated type, you can use it much like any other data type. Because an enumerated type is a class rather than an instance, you must prefix the keyword `Enum:` to the type name when you specify an enumerated data type. You can type in `Enum:` followed by the name of an enumerated type, or select `Enum: <class name>` from the **Output data type** pulldown, then replace `<class name>`. To use the Data Type Assistant, set the **Mode** to **Enumerated**, then enter the name of the enumerated type. 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:



The screenshot shows a dialog box for configuring an output signal. At the top, there are two input fields for "Output minimum:" and "Output maximum:", both containing empty square brackets "[]". Below these is a field for "Output data type:" with a dropdown menu showing "Enum: BasicColors" and a button with "<<". At the bottom, there is a "Data Type Assistant" section with a "Mode:" dropdown set to "Enumerated" and a text field containing "BasicColors".

Prefixing `Enum:` has the same effect as prefixing `?` to a class name when working directly in MATLAB, as described in “Obtaining Information About Classes with Meta-Classes”. You can use either syntax in a Simulink model, but `Enum:` is preferable because it is more self-explanatory in the context of a graphical user interface. Only the `?` syntax works in the MATLAB Command Window.

You cannot set a minimum or maximum value for a signal of an enumerated type, because the concepts of minimum and maximum are not relevant to the purpose of enumerated types. If you change the minimum or maximum for a signal of an enumerated type from the default value of `[]`, an error occurs when you update the model.

Instantiating an Enumerated Type

To use a specific value from an enumerated type, you must instantiate the value. You can instantiate a numeric type in MATLAB or a Simulink model. The syntax is the same in either environment.

Instantiating an Enumerated Type in MATLAB

To instantiate an enumerated type 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 before, you can type:

```
>> bcy = BasicColors.Yellow

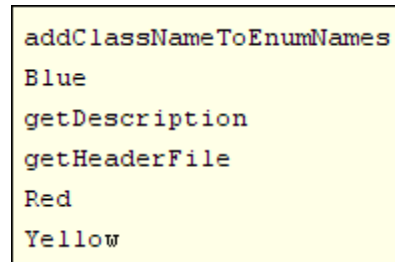
bcy =

    Yellow
```

Tab Completion. Tab completion works for enumerated types. For example, if you enter:

```
BasicColors.<tab>
```

MATLAB displays the elements and methods of `BasicColors` in alphabetical order:

A screenshot of the MATLAB Command Window showing the results of tab completion for the expression 'BasicColors.'. The results are listed in alphabetical order: 'addClassNameToEnumNames', 'Blue', 'getDescription', 'getHeaderFile', 'Red', and 'Yellow'.

```
addClassNameToEnumNames
Blue
getDescription
getHeaderFile
Red
Yellow
```

You can double-click any element or method to insert it at the position where you pressed <tab>. See “Completing Statements in the Command Window — Tab Completion” for more information.

Casting Enumerated Types 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)

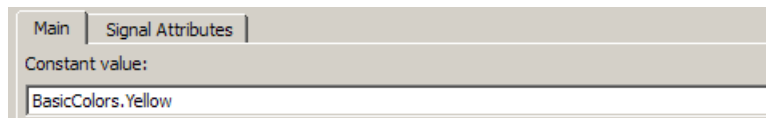
bci =

     2
```

In either case, MATLAB returns the result of the cast in a 1x1 array of the relevant type.

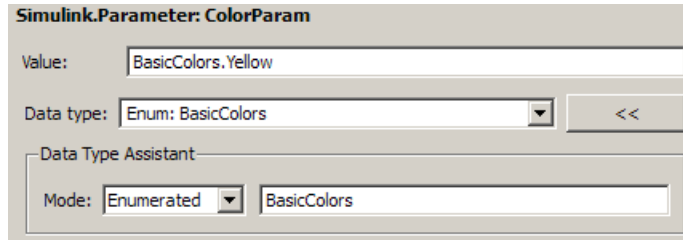
Instantiating an Enumerated Type in Simulink

To instantiate an enumerated type in a Simulink model, you can enter *ClassName.EnumName* as a value in a dialog box. For example, in the previous model the block **Constant**, which outputs the enumerated value **Yellow**, defines that value as follows:

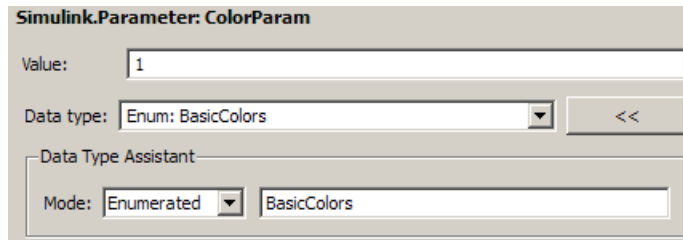


You can enter any valid MATLAB expression that evaluates to an enumerated value. For example, you can enter `BasicColors(2)`, or if you had previously executed `bcy = BasicColors.Yellow` in MATLAB, you could enter `bcy`.

If you create a `Simulink.Parameter` object of an enumerated type, you must specify the parameter value as a class instance, using a valid MATLAB expression such as *ClassName.EnumName*:



You *cannot* specify the parameter value directly as the numeric value of its underlying integer. Thus the following would fail even though the underlying integer of `BasicColors.Yellow` is 1:



Displaying Enumerated Values

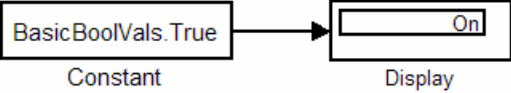
Wherever possible, displaying an enumerated value displays its name, not its underlying integer. However, the underlying integers can affect enumerated value display in Scope blocks and Floating Scope blocks:

- When a Display block displays an enumerated value, it displays the enumerated name, not the underlying integer.
- When a Scope block displays 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.
- When a Floating Scope block displays signals that are all of the same enumerated type, labels appear on the Y axis as they would in a Scope block. If the Floating Scope displays mixed types, no textual labels appear, and any enumerated values are represented by their underlying integers.

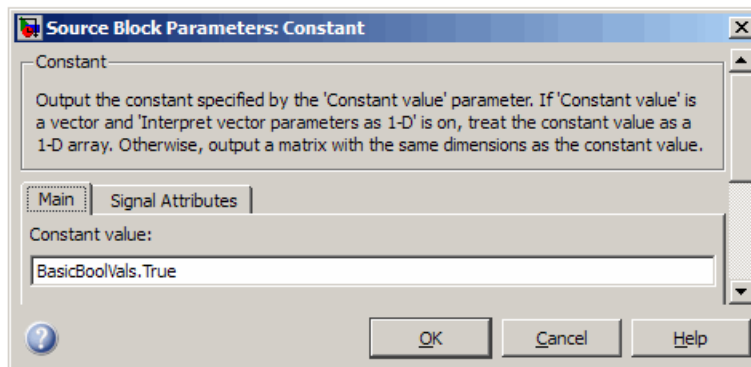
Enumerated Values with Non-Unique Integers

More than one enumerated value in a type can have the same underlying integer, as described in . When this occurs, displaying the value on a scope axis or in a Display block always shows the lexically first value in the enumerated class definition that has the shared underlying integer. For example:

```
classdef(Enumeration) BasicBoolVals < Simulink.IntEnumType
    enumeration
        On(1)
        Off(0)
        True(1)
        False(0)
    end
end
```



The diagram illustrates the relationship between a Constant block and a Display block. The Constant block is labeled 'BasicBoolVals.True' and has an arrow pointing to the Display block, which is labeled 'On'. This demonstrates that when a Constant block outputs a value that corresponds to multiple enumerated values (True and On), the Display block shows the lexically first value (On) that shares that underlying integer value.



Although the Constant block outputs True, both On and True have the same underlying integer, and On is defined first in the enumeration section, so the Display block shows On. Similarly, a Scope axis would show only On, never True, no matter which of the values is input to the Scope.

Enumerated Values in Computation

By design, the Simulink software 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 can not 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. Thus 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 14-19 for more information.

Enumerated types are intended to represent program states and to control program logic in Stateflow blocks and in Simulink 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 lexical order in the enumerated class definition.

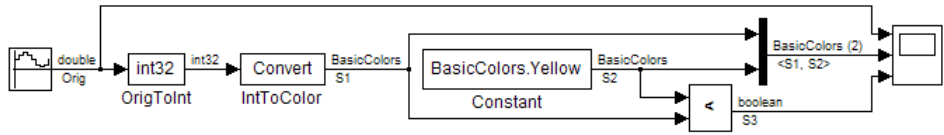
When a block like the Switch block or Multiport Switch block selects among multiple data signals, and any data signal is of an enumerated type, all the data signals must be of that same enumerated type. When a block inputs both control and data signals, as Switch and Multiport Switch do, the control 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.

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.

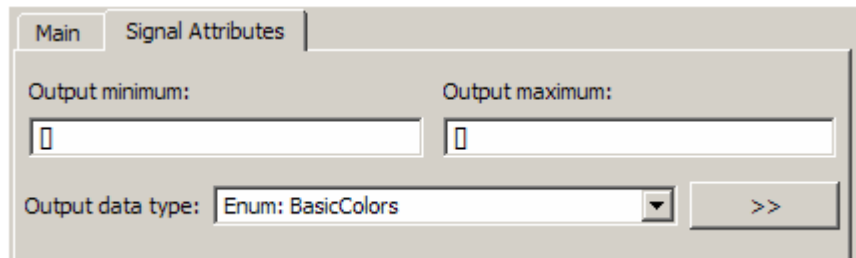
You cannot use a Data Type Conversion block to cast a numeric signal of any non-integer data type to an enumerated type. For example, the model used in “Simulating with Enumerated Types” on page 14-11 needed two Data Conversion blocks to convert a sine wave to enumerated values:



The first block casts `double` to `int32`, and the second block casts `int32` to `BasicColors`. You cannot cast a complex signal to an enumerated type regardless of the data types of its real and imaginary parts.

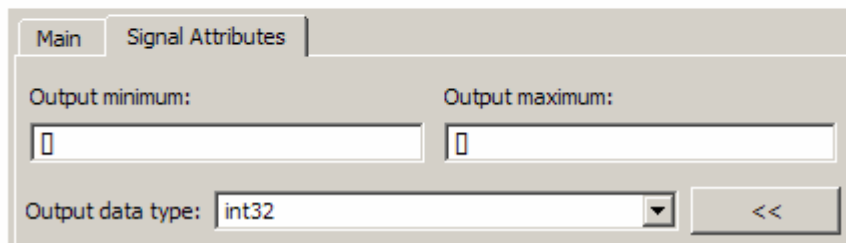
Casting Enumerated Block Parameters

You cannot cast a block parameter of any numeric data type to an enumerated data type. For example, suppose that a `Constant` block specified a **Constant value** of 2 and an **Output data type** of `Enum: BasicColors`:



An error would occur because the specifications implicitly cast a `double` value to an enumerated type. The error occurs even though the numeric value corresponds to one of the enumerated values in the enumerated type.

You cannot cast a block parameter of an enumerated data type to any other data type. For example, suppose that a `Constant` block specified a **Constant value** of `BasicColors.Blue` and an **Output data type** of `int32`:



An error would occur 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 Enumerated Types

In this section...
“Overview” on page 14-22
“Block Support” on page 14-22
“Class Support” on page 14-23
“Logging Enumerated Data” on page 14-24
“Importing Enumerated Data ” on page 14-24

Overview

In general, all Simulink 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. Referenced models, subsystems, masks, buses, logging, and most other Simulink infrastructure 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, even though an enumerated value has an underlying integer that can be used numerically. Underlying integers have no meaning relative to the purpose for which enumerated types exist, so their use for unrelated computational purposes is disallowed. Thus an enumerated type is not considered to be a numeric type.

The Simulink documentation usually mentions nonsupport of enumerated types only where necessary to prevent a misconception or describe an exception. See “Simulink Enumerated Type Limitations” on page 14-25 for information about certain constructs that do not support enumerated types.

Block Support

The following Simulink blocks support enumerated types:

- Constant
- Data Type Conversion

- Display
- Embedded MATLAB Function
- Floating Scope
- Inport
- Multiport Switch
- Outport
- Relational Operator
- Scope
- Signal Specification
- Switch
- Switch Case

All members of these categories of Simulink blocks support enumerated types:

- Bus-capable blocks (see “Bus-Capable Blocks” on page 12-9)
- Pass-through blocks:
 - With state, like Data Store Memory and Unit Delay
 - Without state, like Mux

Many Simulink blocks in addition to those explicitly listed 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 for each Simulink block describes all data types that it 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: Array, Structure, Structure with Time, TimeSeries. To File blocks accept only data of type double, so they do not support enumerated types. See Chapter 15, “Importing and Exporting Data” 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 Chapter 15, “Importing and Exporting Data” for more information.

Simulink Enumerated Type Limitations

In this section...

“Enumerated Types and Bus Initialization” on page 14-25

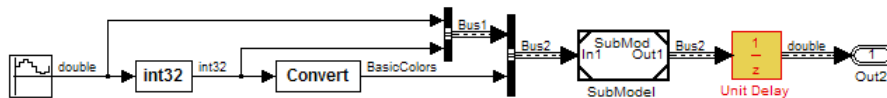
“Enumerated Types on Scopes” on page 14-26

“Enumerated Types for Switch Blocks” on page 14-26

“Nonsupport of Enumerated Types” on page 14-27

Enumerated Types and Bus Initialization

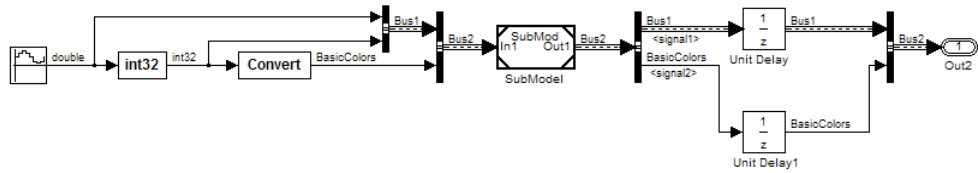
A bus can contain any number of signals that have enumerated data types. With one exception, such a bus can be used like any other bus. The exception is that a bus that contains any enumerated signal cannot pass through any block that requires an initial value, like the Unit Delay block. For example, building the following model generates an error:



The error occurs because the Unit Delay block cannot supply an initial value that is correct for all signals in the bus without implicitly using some enumerated value’s underlying integer computationally. One possible solution is:

- 1** Extract the enumerated signal from the bus.
- 2** Pass the enumerated and bus signals through separate Unit Delay blocks.
- 3** Recombine the enumerated signal into the bus if needed.

The following model illustrates the described technique:



The separate Unit Delay blocks will usually have identical attributes except for **Initial conditions**, which in the upper block must be numeric and in the lower block must be an instance of `BasicColors`, specified as described in “Instantiating an Enumerated Type” on page 14-15. The same technique and rules apply to all bus-capable blocks that require a value that provides an initial state. See “Bus-Capable Blocks” on page 12-9 for more information.

Enumerated Types on Scopes

When a Scope 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 for the first time before any simulation has occurred, or between simulations, the Scope displays only numeric labels. When simulation begins, enumerated names replace the numeric labels, and thereafter appear whenever the Scope is opened.

When a Floating Scope 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 displays more than one type of enumerated signal, or any numeric signal, no textual labels 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 enumerated types. If the control input has an enumerated data type, choose one of these 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 Enumerated Types

The following constructs do not support enumerated types:

- Enumerated types cannot be defined:
 - Dynamically in the MATLAB Command Window
 - Using any GUI tool like the Data Class designer
- Packages cannot contain enumerated type definitions.
- Some blocks that could support enumerated types do not do so:
 - If-action block
 - To File block
 - From File block
- Generated code does not support logging enumerated data.
- Custom targets do not support enumerated types.
- Simulink® HDL Coder™ does not support enumerated types

Importing and Exporting Data

- “Introduction” on page 15-2
- “Logging Signals” on page 15-3
- “Importing Data from a Workspace” on page 15-16
- “Exporting Data to the MATLAB Workspace” on page 15-24
- “Importing and Exporting States” on page 15-29
- “Limiting Output” on page 15-32
- “Specifying Output Options” on page 15-33

Introduction

During simulation you can import input signal and initial state data from a workspace, and export output signal and state data to a workspace. This capability allows you to use standard or custom MATLAB functions to generate a simulated system's input signals and to graph, analyze, or otherwise postprocess the system's outputs.

Logging Signals

In this section...

- “About Signal Logging” on page 15-3
- “Globally Enabling and Disabling Logging” on page 15-4
- “Enabling Logging for a Signal” on page 15-4
- “Displaying Logging Indicators” on page 15-5
- “Specifying a Logging Name” on page 15-5
- “Limiting the Data Logged for a Signal” on page 15-6
- “Logging Virtual Signals” on page 15-6
- “Logging Multidimensional Signals” on page 15-6
- “Logging Composite Signals” on page 15-7
- “Logging Referenced Model Signals” on page 15-7
- “Viewing Logged Signal Data” on page 15-8
- “Accessing Logged Signal Data” on page 15-9
- “Handling Spaces and Newlines in Logged Names” on page 15-9
- “Extracting Partial Data from a Running Simulation” on page 15-12
- “Example: Logging Signal Data in the F14 Model” on page 15-12
- “Signal Logging Limitations” on page 15-15

About Signal Logging

Logging signals refers to the process of saving signal values to the MATLAB workspace during simulation for later retrieval and postprocessing. Simulink allows you to log a signal by

- Connecting the signal to a To Workspace block, Scope block, or viewer.

This method allows you to document in the diagram itself the workspace variables used to store signal data. Results are visible during simulation. Be aware that Scopes store data and can be memory intensive.

- Connecting the signal to a root-level Outport block.

This method reduces diagram clutter by eliminating the need to use Scope blocks to log signals. Data is available only when simulation is paused or completed.

- Setting the signal's signal logging properties.

This method eliminates the need to add blocks. Data is available only when simulation is paused or completed.

All of these methods allow you to specify the names of the workspace variables used to save signal data and to limit the amount of data logged for a particular signal.

See *Simulink Reference* for the To Workspace and Outport blocks for information on using these blocks to log signal data. See the documentation of the `sim` command for some data logging capabilities that are available only for programmatic simulation.

Globally Enabling and Disabling Logging

You can globally enable or disable signal logging for a model by checking or unchecking the **Signal logging** option on the **Data Import/Export** pane of the **Configuration Parameters** dialog box (see “Signal logging”). Simulink logs signals only if this option is checked. If the option is not checked, Simulink ignores the signal logging settings for individual signals.

Enabling Logging for a Signal

To enable signal logging for a signal, select the **Log signal data** option on the signal's **Signal Properties** dialog box. See “Signal Properties Dialog Box” for more information.

Enabling Signal Logging Programmatically

You can enable signal logging programmatically for selected blocks with the outport `DataLogging` property. You can set this property using the `set_param` command. For example:

- 1 At the MATLAB Command Window, open a model. Type

```
vdp
```

- 2 Select a block in that model. For example, select the Mux block.
- 3 Get the port handles of the selected block.

```
get_param(gcb, 'PortHandles')
```

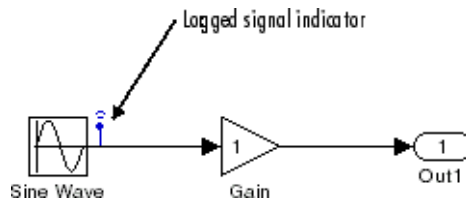
- 4 Enable signal logging for the desired output port.

```
set_param(ans.Outputport(1), 'DataLogging', 'on')
```

The logged signal indicator (↑) appears.

Displaying Logging Indicators

By default, Simulink displays an indicator on each signal whose **Signal Properties > Log signal data** option is enabled. For example, in the following model the output of the Sine Wave block is logged:



A logged signal can also be a test point. See “Working with Test Points” on page 10-61 for information about test points.

To turn display of logging indicators on or off, select or clear **Port/Signal Displays > Testpoint/Logging Indicators** from the Simulink **Format** menu.

Specifying a Logging Name

You can assign a name, called the logging name, to the object used to log data for a signal during simulation. To specify a log name for a signal, select **Custom** from the **Logging name** list on the signal’s **Signal Properties** dialog box and enter the custom name in the adjacent text field.

If you do not specify a custom logging name, Simulink uses the signal name, or if there is no name, Simulink 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 following name: SL_MyBlock1.

Limiting the Data Logged for a Signal

The **Data** panel of the **Signal Properties** dialog box lets you limit the amount of data logged for a signal. For example, you can specify the maximum amount of data to be logged for a signal or a decimation factor that causes Simulink to skip a specified number of time steps before logging a signal value. See “Data” for more information.

Logging 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 functional or mathematical significance. The source of a virtual signal is always a virtual block, like a Mux block or Selector block. See “Virtual Signals” on page 10-12 and “Mux Signals” on page 10-15 for more information.

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 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.TsArray` object. The log gives the duplicate regions unique names, using the syntax: `<signal_name>_reg<#counter>`.

Logging Multidimensional Signals

You can log multidimensional signals, which are signals whose elements are nonscalar. The techniques for logging multidimensional signals are the same as those for logging any other signal. See “Signal Dimensions” on page 10-8 for information about multidimensional signals.

Logging Composite Signals

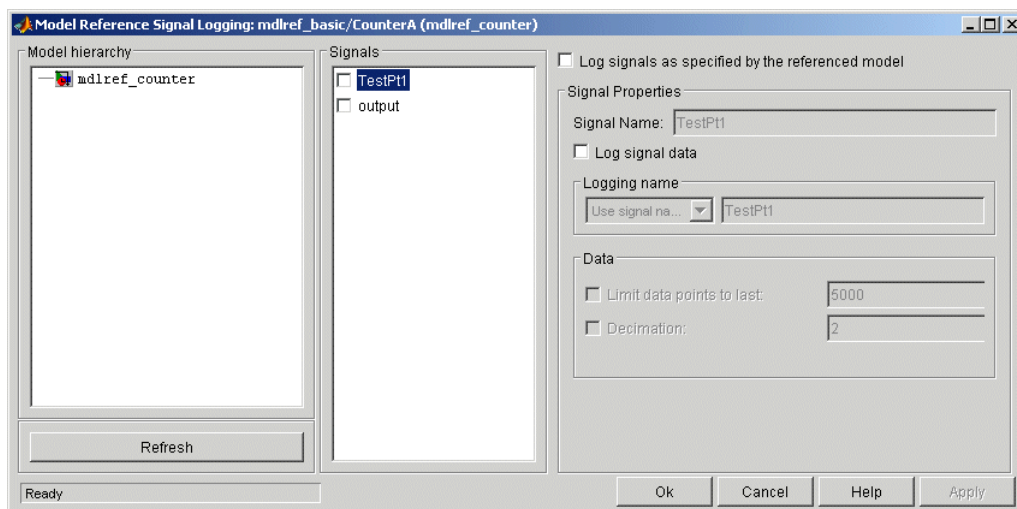
You can log Simulink composite signals, which are called buses. The hierarchy of a bus signal is preserved in the `logouts` object. The logged name of a signal in a virtual bus derives from the name of the source signal. The logged name of a signal in a nonvirtual bus derives from the applicable bus object, and can differ from that of the source signal. See Chapter 12, “Using Composite Signals” and “Using Bus Objects” on page 12-10 for information about those capabilities.

Logging Referenced Model Signals

You can log any signal that is defined as a test point in a referenced model. For information about test points, see “Designating a Signal as a Test Point” on page 10-61. For information about referenced models, see Chapter 6, “Referencing a Model”.

To log test pointed signals in referenced models, select the Model block and then select **Log referenced signals** from the model editor’s **Edit** menu or from the block’s context menu.

The **Model Reference Signal Logging** dialog box appears.



The dialog box contains the following panes and controls.

Model Hierarchy

This pane displays the contents of the referenced model as a tree control with expandable nodes. The top-level node represents the referenced model. Expanding this node displays the subsystems that the referenced model contains and any models that it itself references. Expanding a subsystem node displays the subsystems that it contains and the models that it references.

Refresh Button

Refreshes the dialog box to reflect changes in the model hierarchy.

Signals

This pane displays the test points of the model or subsystem selected in the **Model Hierarchy** pane. See “Working with Test Points” on page 10-61. Check the check box next to a test point’s name to specify that it should be logged.

Log signals as specified by the referenced model

Checking this check box causes Simulink to log the signals that the referenced model specifies should be logged.

Signal Properties

This pane is enabled if **Log signals as specified by the referenced model** is not selected. In this case, the controls on this pane allow you to specify the signal logging properties of the signal selected in the **Signals** pane. The values that you specify override for this instance of the referenced model those specified by the model itself. The controls correspond to the controls of the same name on the **Signal Properties** dialog box. See “Signal Properties Dialog Box” for information on how to use them.

Viewing Logged Signal Data

To view logged signal data, either check **Configuration Parameters > Data Import/Export > Inspect signals when simulation is stopped/paused** or select **Tools > Inspect Logged Signals** from the model editor’s menu bar. The first method causes Simulink to display logged signals in the **MATLAB Time Series Tools** viewer (see “Time Series Tools”) whenever a simulation

ends or you pause a simulation. The second method causes Simulink to display the data immediately.

Note You must run the simulation first before selecting **Tools > Inspect Logged Signals**. Otherwise, selecting this command has no effect.

Accessing Logged Signal Data

Simulink saves signal data that it logs during simulation in a Simulink data object of type `Simulink.ModelDataLogs` that resides in the MATLAB workspace. The name of the object's handle is `logouts` by default. The **Data Import/Export** configuration pane (see “Data Import/Export Pane”) allows you to specify another name for this object. See `Simulink.ModelDataLogs` for information on extracting signal data from this object. The signal logs for particular model elements are contained in the objects in the following table.

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 mux or bus	<code>Simulink.Timeseries</code>
Mux or bus signal	<code>Simulink.TsArray</code>

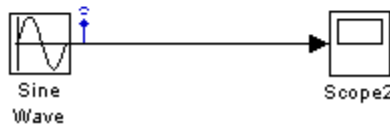
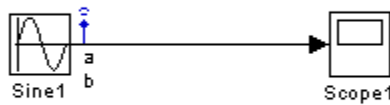
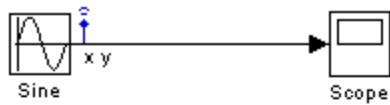
Handling Spaces and Newlines in Logged Names

Names that include space or newline characters can improve the readability of a block diagram, but referencing names that include such characters in a data log requires special techniques. These techniques allow the MATLAB parser to process the names even though spaces and newlines are not legal in MATLAB identifiers. Signal names in data logs can have spaces or newlines in their names under the following circumstances:

- The signal is named, and the name includes blank or newline characters.
- The signal is unnamed, and originates in a block whose name includes blank or newline characters.

- The signal exists in a subsystem or referenced model, and the name of the subsystem or Model block, or of any superior block, includes blank or newline characters.

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:



If you execute these examples with data logging enabled in the Data Import/Export pane, accepting the default logging object name `logsout`, and then type `logsout` in the MATLAB Command Window, MATLAB displays the following data log:

```
logsout =

Simulink.ModelDataLogs (model_name):
  Name                Elements  Simulink Class
  ('x y')              1        Timeseries
  ('a
  b')                  1        Timeseries
  ('SL_Sine
  Wave1')              1        Timeseries
```


You cannot access any of the `Timeseries` objects in this log using TAB name completion, or by typing the name to MATLAB, because the space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```
>> logcout.x y

??? logcout.x y
      |
Error: Unexpected MATLAB expression.
```

To reference a `Timeseries` object whose name contains a space, as in the first example above, single-quote the element containing the space:

```
>> logcout.('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 `Timeseries` object whose name contains a newline, as in the second example above, concatenate to construct the element containing the newline:

```
>> cr=sprintf('\n')
>> logcout.(['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 in the third example above:

```
>> 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. No analogous techniques exist for TAB name completion, which never works with names that contain space or newline characters.

Extracting Partial Data from a Running Simulation

Before a simulation ends, you can extract and write the currently logged signal data from `Simulink.ModelDataLogs` with the `set_param WriteDataLogs` command. The currently logged signal is the partial data logged between when the simulation started and when you request an extraction of the signal data. If you use this command during the simulation, Simulink writes the current logging variable values to the MATLAB workspace. If you use this command at the end of the simulation, Simulink writes the values from the last simulation to the MATLAB workspace.

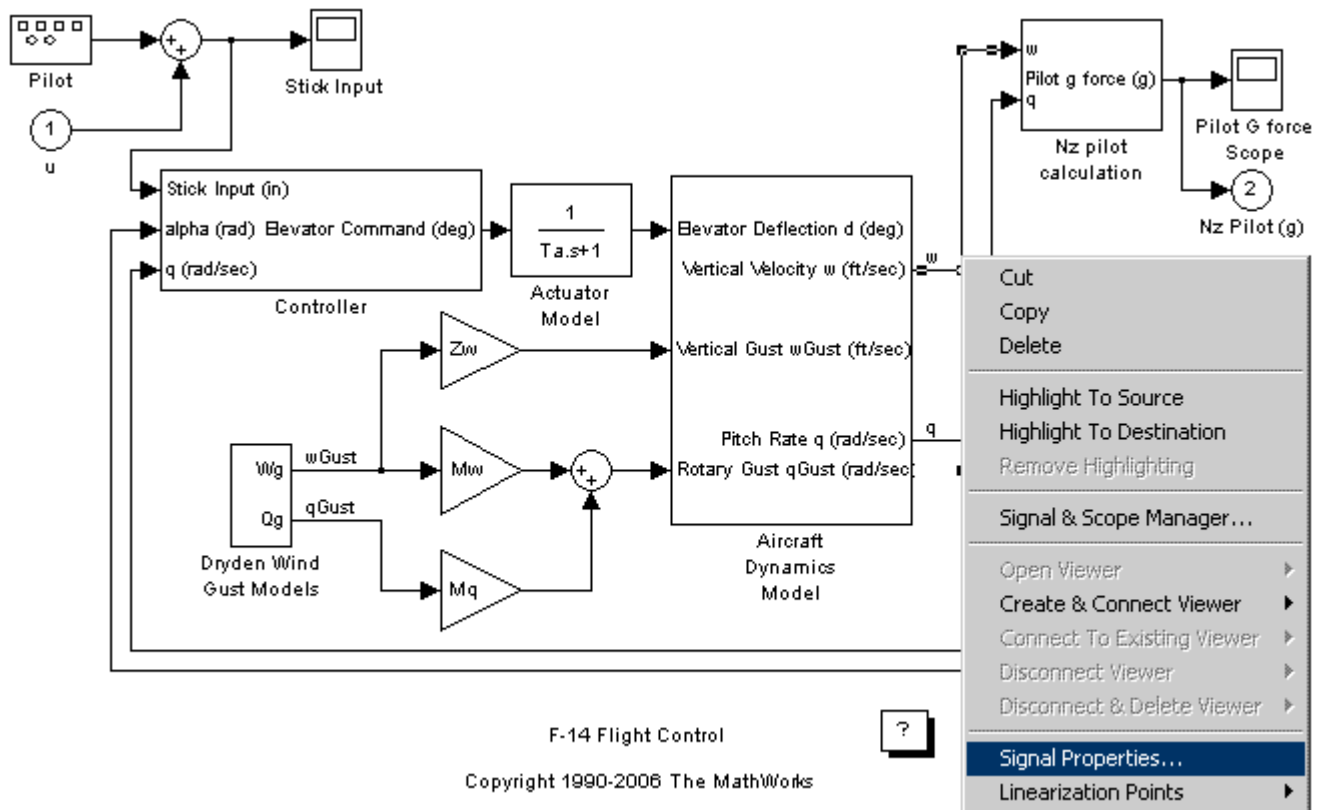
To use this command, type the following at the MATLAB Command Window.

```
set_param(bdroot, 'SimulationCommand', 'WriteDataLogs')
```

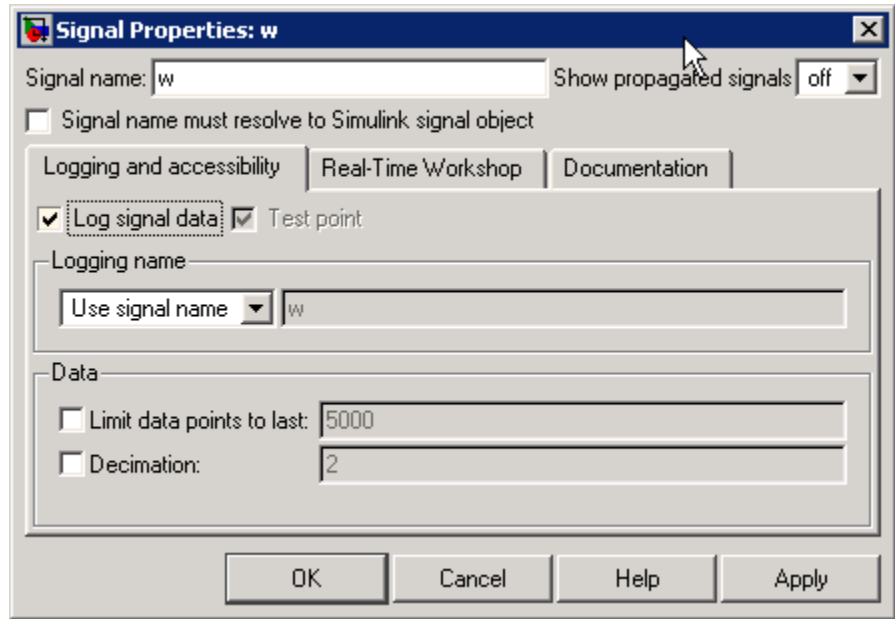
Example: Logging Signal Data in the F14 Model


Enabling signal logging on a signal-by-signal basis allows you to store signal data without modifying the structure of the Simulink diagram. For example, use the following steps to log and access the signal data for the vertical velocity signal `w` in the F14 model.

- 1** Open the F14 model by typing `f14` at the MATLAB command prompt.
- 2** Right-click on the signal labeled `w` and select the **Signal Properties** menu.



- In the **Signal Properties** dialog box that opens, select the **Log signal data** option. The logging name initializes to the signal's name.



- 4 Click the **OK** button on the **Signal Properties** dialog box. The 'blue antenna' icon  appears on the signal labeled w, indicating that this signal will be logged during simulation.
- 5 Ensure that **Configuration Parameters > Data Import/Export Signal logging** is selected and that the logging name is set to the default variable logstdout.
- 6 Run the F14 simulation. The logged signal data is stored in a Simulink.ModelDataLogs object named logstdout in the MATLAB workspace. Typing logstdout at the MATLAB command prompt displays the following

```
logstdout =

Simulink.ModelDataLogs (f14):
  Name           Elements  Simulink Class
  w              1        Timeseries
```

- 7 Type `logout.w` to view the information stored for the signal `w`.

```
logout.w
      Name: 'w'
      BlockPath: 'f14/Aircraft Dynamics Model'
      PortIndex: 1
      SignalName: 'w'
      ParentName: 'w'
      TimeInfo: [1x1 Simulink.TimeInfo]
      Time: [1353x1 double]
      Data: [1353x1 double]
```

- 8 To inspect the signal using the MATLAB **Time Series Tools**, select **Inspect Logged Signals** from the `f14` model editor's **Tools** menu (see “Time Series Tools” in the MATLAB documentation).

Signal Logging Limitations

- When you log data using the `Structure`, `Structure with time`, or `Timeseries` format, each field that contains logged data can contain at most $2^{31}-1$ bytes on a 32-bit platform, and $2^{48}-1$ bytes on a 64-bit platform.
- When you log data using `Array` format, each array that contains logged data can contain at most $2^{31}-1$ bytes on a 32-bit platform, and $2^{48}-1$ bytes on a 64-bit platform.
- Simulink data logging does not support the following types of signals:
 - Output of a Function-Call Generator block
 - Signal connected to the input of a Merge block
 - Outputs of Trigger and Enable blocks
- Rapid Accelerator mode does not support signal logging. For more information, see “Using Scopes and Viewers with Rapid Accelerator Mode” on page 27-16.

Importing Data from a Workspace

In this section...

“Enabling Data Import” on page 15-16

“Importing Time-Series Data” on page 15-17

“Importing Data Arrays” on page 15-18

“Using a MATLAB Time Expression to Import Data” on page 15-19

“Importing Data Structures” on page 15-20

“Specifying Time Vectors for Discrete Systems” on page 15-22

Enabling Data Import

The Simulink software can input data from a workspace and apply it to the model’s top-level input ports during a simulation run. To specify this option:

- 1 Select the **Input** box in the **Load from workspace** area of the “**Data Import/Export Pane**” pane.
- 2 Enter an external input specification in the adjacent edit box and click **Apply**.

The Simulink software resolves symbols used in the external input specification as described in “Resolving Symbols” on page 4-75. See the documentation of the `sim` command for some data import capabilities that are available only for programmatic simulation.

Note The use of the **Input** box is independent of the setting of the **Format** list on the **Data Import/Export** pane.

The input data can take any of the following forms:

- Time series — see “Importing Time-Series Data” on page 15-17
- Array — see “Importing Data Arrays” on page 15-18

- Time expression — see “Using a MATLAB Time Expression to Import Data” on page 15-19
- Structure — see “Importing Data Structures” on page 15-20

The Simulink software linearly interpolates or extrapolates input values as necessary if the **Interpolate data** option is selected for the corresponding Inport.

Importing Time-Series Data

Any root-level Inport block can import data specified by a time-series object (see `Simulink.Timeseries`) residing in a workspace. In addition, any root-level input port defined by a bus object (see `Simulink.Bus`) can import data from a time-series array object (see `Simulink.TSArray`) that has the same structure as the bus object. Time-series objects are a derivation of standard MATLAB time-series objects and, therefore, can be manipulated using the MATLAB Time Series Tools. See “Using Time Series Tools” in the MATLAB Data Analysis documentation for further details.

Importing time-series objects allows you to import data logged by a previous simulation run (see “Logging Signals” on page 15-3). For example, suppose that you have a model that references several other models. You could use data logged from the inputs of the referenced models when simulating the top model as inputs for the referenced models simulated by themselves. This allows you to test the referenced models independently of the top model and each other.

To import data from time-series objects and time-series array objects, enter a comma-separated list of variables or expressions into the **Input** edit field on the **Data Import/Export** pane of the Configuration Parameters dialog box (see “Configuration Parameters Dialog Box”). Each variable or expression in the **Input** list should evaluate to the appropriate time-series object or time-series array object that corresponds to one of the model’s root-level input ports, with the first item corresponding to the first root-level input port, the second to the second root-level input port, and so on. The model `sldemo_md1ref_counter_bus`, referenced by the top model `sldemo_md1ref_bus`, contains an example of importing time-series objects.

To use this demo, open `sldemo_md1ref_bus` and run the simulation. The top model is configured to store logged signals into a variable named `topOut`. Currently, two signals are being logged: `COUNTERBUS` and `OUTPUTBUS`. After running the simulation, you can view the logged signals by typing `topOut` at the MATLAB prompt.

```
topOut =

Simulink.ModelDataLogs (sldemo_md1ref_bus):
      Name          Elements  Simulink Class
      COUNTERBUS    2         TsArray
      OUTPUTBUS     2         TsArray
```

The variable `topOut` is a `Simulink.ModelDataLogs` object that contains, in this case, two `Simulink.TsArray` objects corresponding to the two logged bus signals. The `Simulink.TsArray` object `COUNTERBUS` can be used as the input to the submodel `sldemo_md1ref_counter_bus` to run this model independently of the top model. This is accomplished by entering `topOut.COUNTERBUS` into the **Input** edit field on the **Data Import/Export** pane of the Configuration Parameters dialog box, as shown below.



By using the time-series array object as the input to `sldemo_md1ref_counter_bus`, independently running this model produces the same output as when run within the top model `sldemo_md1ref_bus`.

Importing Data Arrays

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 block signal (in sequential order) and each row is the input value for the corresponding time point.

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.

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:

```
t = (0:0.1:1)';  
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`.

Using a MATLAB Time Expression to Import Data

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 model's input ports. 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)`.

Importing Data Structures

The Simulink software can read data from the workspace in the form of a structure whose name is specified in the **Input** text field. You can import structures that include only signal data or both signal and time data. The type of data structure is evaluated based on the structure itself.

Limitation: You cannot import fixed-point data that is contained in a structure. Consider using a `Simulink.Timeseries` object instead of a structure.

Importing Signal-and-Time Data Structures

To import structures that include both signal and time data, the input structure must have two top-level fields: `time` and `signals`. The `time` field contains a column vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to a model input port.

Each `signals` substructure must contain two fields named `values` and `dimensions`, respectively. The `values` field must contain an array of inputs for the corresponding input port where each input corresponds to a time point specified by the `time` 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 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.

If the inputs for a port are scalar or vector values, the `values` field must be an M -by- N array where M is the number of time points specified by the `time`

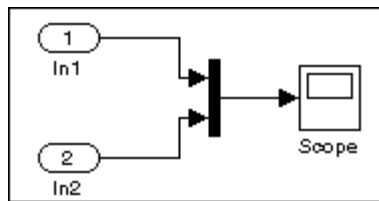
field and N is the length of each vector value. For example, the following code creates an input structure for loading 11 time samples of a two-element signal vector of type `int8` into a model with a single input port:

```
a.time = (0:0.1:1)';
c1 = int8([0:1:10]');
c2 = int8([0:10:100]');
a.signals(1).values = [c1 c2];
a.signals(1).dimensions = 2;
```

To load this data into the model's input port, you would select the **Input** option on the **Data Import/Export** pane and enter `a` in the input expression field.

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.

As another example, consider the following model, which 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 vector, `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;
```

Select the **Input** box for this model, and enter a in the adjacent text field.

Importing Signal-Only Structures

The `Structure` format is the same as the `Structure with time` format except that the `time` field is empty. For example, in the preceding example, you could set the `time` field as follows:

```
a.time = []
```

In this case, the input for the first time step is read from the first element of an input port's value array, the value for the second time step from the second element of the value array, etc. If you enter the structure without time, the `Inport` block must have a discrete sample time.

Per-Port Structures

This 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.

Specifying Time Vectors for Discrete Systems

In some cases, the Simulink software calculates block sample hits at sample times different from those specified by a time vector generated in MATLAB. Typically, these are small floating point inaccuracies that can cause the Simulink product to apparently miss a specified time step in lieu of a different sample point. In order to avoid these numerical inaccuracies, generate the time vector either in MATLAB or in Simulink based on the fundamental sample time of the model.

For example, if the model has a fundamental sample time T_s (see "Purely Discrete Systems" on page 3-22) of 0.001 then the time vector `Tvector` should be calculated with the command

```
Tvector = Ts*[n1, n2, n3...];
```

where n_1, n_2, n_3 , etc. are integers that, when multiplied by the fundamental sample time, yield the desired time vector.

Exporting Data to the MATLAB Workspace

In this section...
“Enabling Data Export” on page 15-24
“Format Options” on page 15-25

Enabling Data Export

You can export a model's states and root-level outputs to the MATLAB workspace during simulation of the model. To do this, select the type of data that you want to export on the **Save to workspace** area of the “**Data Import/Export Pane**” pane of the Configuration Parameters dialog box. The field adjacent to each type specifies the name of a MATLAB workspace variable to be used by the Simulink software to store the exported data.

Each field initially specifies a default variable. You can edit the fields to specify names of your own choosing. Select **Signal logging** to enable signal logging for the model. See “Logging Signals” on page 15-3 for more information. See the documentation of the `sim` command for some data export capabilities that are available only for programmatic simulation.

Note The output is saved to the MATLAB workspace at the base sample rate of the model. Use a To Workspace block if you want to save output at a different sample rate.

The **Save options** area enables you to specify the format and restrict the amount of output saved.

See the documentation of the `sim` command for some capabilities that are available only for programmatic simulation. Format options for model states and outputs are listed below.

Format Options

Array

If you select this option, a model's states and outputs are saved 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 model's states. 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, each row to the outputs at a specific time.

Note You can use array format to save your model's outputs and states only if the outputs are either all scalars or all vectors (or all matrices for states), are either all real or all complex, and are all of the same data type. Use the **Structure** or **Structure with time** output formats (see "Structure with time" on page 15-25) if your model's outputs and states do not meet these conditions.

Structure with time

If you select this format, the model's states and outputs are saved in structures having the names specified in the **Save to workspace** area (for example, xout and yout).

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 is an example of 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 model's states.

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 where each row corresponds to a time sample of the state and where the first two elements of each row correspond to the first column of the sample and the last two elements correspond to the second column of the sample.

Note The Simulink software can read back simulation data saved to the MATLAB workspace in the `Structure` with `time` output format. See “Importing Signal-and-Time Data Structures” on page 15-20 for more information.

Structure

This format is the same as the preceding except that the Simulink software does not store simulation times in the `time` field of the saved structure.

Per-Port Structures

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.

Importing and Exporting States

In this section...

“Introduction” on page 15-29

“Saving Final States” on page 15-29

“Loading Initial States” on page 15-30

Introduction

You can import the initial values of a system’s states, i.e., its initial conditions, at the beginning of a simulation and save the final values of the states at the end of the simulation. This feature allows you to save a steady-state solution and restart the simulation at that known state.

Saving Final States

To save the final values of a model’s states, select **Final states** in the **Save to workspace** area of the **Data Import/Export** pane and enter a name in the adjacent edit field. The states are saved in a workspace variable having the specified name. The saved data has the format that you specify in the **Save options** area of the **Data Import/Export** pane.

When saving states from a referenced model in the structure-with-time format, a boolean subfield is added named `inReferencedModel` to the `signals` field of the saved data structure. This field’s value 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 comprising 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, e.g.,

```
>> xout.signals(1).blockName

ans =

sldemo_md1ref_basic/CounterA|sldemo_md1ref_counter/Previous Output
```

Loading Initial States

To load states, check **Initial state** in the **Load from workspace** area of the **Data Import/Export** pane and specify the name of a variable that contains the initial state values, for example, a variable containing states saved from a previous simulation. The initial values specified by the workspace variable override the initial values specified by the model itself, i.e., the values specified by the initial condition parameters of those blocks in the model that have states.

Use the `structure` or `structure-with-time` option to specify initial states if you want to accomplish any of the following.

- Associate initial state values directly with the full path name to the states. This eliminates errors that could occur if the Simulink software reorders the states, but the initial state array is not correspondingly reordered.
- Assign a different data type to each state's initial value.
- Initialize only a subset of the states.

For example, the following commands create an initial state structure that can be used to initialize the `x2` state of the `vdp` model. The `x1` state is not initialized in the structure and, therefore, the value entered into the state's associated Integrator block is used during the simulation.

```
% Open the vdp demo 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 this `states` variable, open the **Configuration Parameters** dialog box for the `vdp` model. Check **Initial state** in the **Load from workspace** area of the **Data Import/Export** pane and type `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.

Note You must use the structure or structure-with-time format to initialize the states of a top model and the models that it references.

Limiting Output

Saving data to a workspace can slow down the simulation and consume memory. To avoid this, you can limit the number of samples saved to the most recent samples or you can skip samples by applying a decimation factor. To set a limit on the number of data samples saved, select the check box labeled **Limit data points to last** and specify the number of samples to save. To apply a decimation factor, enter a value in the field to the right of the **Decimation** label. For example, a value of 2 saves every other point generated.

Specifying Output Options

In this section...

- “Introduction” on page 15-33
- “Refining Output” on page 15-33
- “Producing Additional Output” on page 15-34
- “Producing Specified Output Only” on page 15-34
- “Comparing Output Options” on page 15-35

Introduction

The **Output options** list on the **Data Import/Export** configuration pane (“Data Import/Export Pane”) enables you to control how much output the simulation generates. You can choose from three options:

- Refine output
- Produce additional output
- Produce specified output only

Refining Output

The `Refine output` choice 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.

To get smoother output, it is much faster to change the refine factor instead of reducing the step size. When the refine factor is changed, the solvers generate additional points by evaluating a continuous extension formula at those points. This option changes the simulation step size so that time steps coincide with the times that you have specified 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; when graphing simulation output, you might find that output from this solver is not

sufficiently smooth. If this is the case, 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 (see “Zero-Crossing Detection” on page 2-24). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Producing Additional Output

The Produce additional output choice enables you to specify directly those additional times at which the solver generates output. When you select this option, an **Output times** field is displayed on the **Data Import/Export** pane. Enter a MATLAB expression in this field that evaluates to an additional time or a vector of additional times. This option causes the solver to produce hit times at the output times you have specified, in addition to the times it needs to accurately simulate the model.

Note This option helps the solver locate zero crossings (see “Zero-Crossing Detection” on page 2-24). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Producing Specified Output Only

The Produce specified output only choice provides simulation output *only* at the simulation start time, simulation stop time, and the specified output times. For example, if the simulation start time is set to 0 and the simulation stop time is set to 60, entering [10: 10: 50] in the Output times field results in simulation output at these times:

0, 10, 20, 30, 40, 50, 60

This option changes the simulation step size so that time steps coincide with the times that you have specified for producing output. The solver may hit other time steps to accurately simulate the model, however the output will not include these points. This choice is useful when you are comparing different simulations to ensure that the simulations produce output at the same times.

Note This option helps the solver locate zero crossings (see “Zero-Crossing Detection” on page 2-24). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Comparing Output Options

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

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.

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

Working with Data Stores

- “About Data Stores” on page 16-2
- “Defining Data Stores with Data Store Memory Blocks” on page 16-6
- “Defining Data Stores with Signal Objects” on page 16-9
- “Accessing Data Stores with Simulink Blocks” on page 16-11
- “Data Store Examples” on page 16-13
- “Ordering Data Store Access” on page 16-16
- “Using Data Store Diagnostics” on page 16-19
- “Data Stores and Software Verification” on page 16-27

About Data Stores

In this section...

- “Introduction” on page 16-2
- “When to Use a Data Store” on page 16-3
- “Creating Data Stores” on page 16-3
- “Accessing Data Stores” on page 16-4
- “Workflow for Configuring Data Stores” on page 16-4

Introduction

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. Two types of data stores exist:

- A *local data store* is accessible anywhere in the model hierarchy at or below the level at which it is defined, except in referenced models. Local data stores can be defined graphically in a model or by creating a model workspace signal object.
- A *global data store* is accessible throughout the model hierarchy, including in referenced models. Global data stores can be defined only in the MATLAB base workspace, and are the only type of data store accessible in referenced models.

The scope of a data store should be no greater than necessary to let it access the relevant parts of the model hierarchy. Some examples of local and global data stores appear in “Data Store Examples” on page 16-13.

Real-Time Workshop Embedded Coder provides a custom storage class that you can use to specify customized data store access functions in generated code. See “Creating and Using Custom Storage Classes” and “GetSet Custom Storage Class for Data Store Memory”.

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. See “Data Stores and Software Verification” on page 16-27 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.

Creating Data Stores

To create a data store, you create a Data Store Memory block or a `Simulink.Signal` object. The block or 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 “Defining Data Stores with Data Store Memory Blocks” on page 16-6.
- A `Simulink.Signal` object can act as a local or global data store. See “Defining Data Stores with Signal Objects” on page 16-9.

Data stores implemented with Data Store Memory blocks:

- Support data store initialization
- Provide finer-grained control of data store scope and options
- Require a block to represent the data store
- Cannot be accessed within referenced models

Data stores implemented with `Simulink.Signal` objects:

- Do not support data store initialization
- Provide coarser-grained 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.

Accessing Data Stores

You cannot read or write a data store using API calls: you must use Simulink blocks designed for the purpose.

- To write a signal to a data store, you 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, you use a Data Store Read block, which reads the value of the data store and outputs that value data as a signal.

The Data Store Write and Data Store Read blocks identify the data store to be read or written by specifying its name as a block parameter. See “Accessing Data Stores with Simulink Blocks” on page 16-11 for more information.

Workflow for Configuring Data Stores

The following is a general workflow for configuring data stores. Many of the actions listed can be executed 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 16-27.
- 2** Create data stores using the techniques described in “Defining Data Stores with Data Store Memory Blocks” on page 16-6 or “Defining Data Stores with Signal Objects” on page 16-9. For greater reliability, consider assigning rather than inheriting data store attributes, as described in “Specifying Data Store Memory Block Attributes” on page 16-6.

- 3** Create the blocks that will read and write the data store, as described in “Accessing Data Stores with Simulink Blocks” on page 16-11.
- 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 “Ordering Data Store Access” on page 16-16.
- 5** Apply the techniques described in “Using Data Store Diagnostics” on page 16-19 as needed to prevent data store errors, or diagnose them if they occur during simulation.
- 6** If you intend to generate code for your model, see “Data Store Memory Considerations” in the Real-Time Workshop documentation.

See Chapter 6, “Referencing a Model” for information about referenced models.

Defining Data Stores with Data Store Memory Blocks

In this section...

“Creating the Data Store” on page 16-6

“Specifying Data Store Memory Block Attributes” on page 16-6

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, though not within Model blocks, drag the block into the subsystem.

Once you have created 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 be used to read and write the data store. See Data Store Memory for information about all of the block’s parameters.

You can specify data store properties beyond those definable with Data Store Memory block parameters by selecting the block’s **Data store name must resolve to Simulink signal object** option and providing a signal object. See “Specifying Attributes Using a Signal Object” on page 16-7 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:

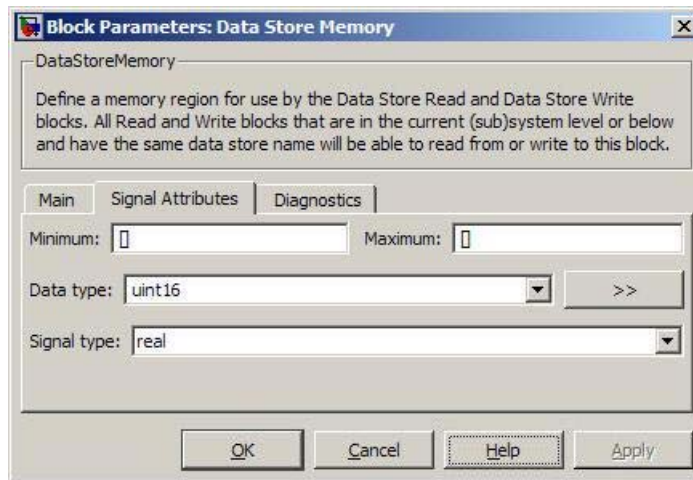
- Data type

- 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.

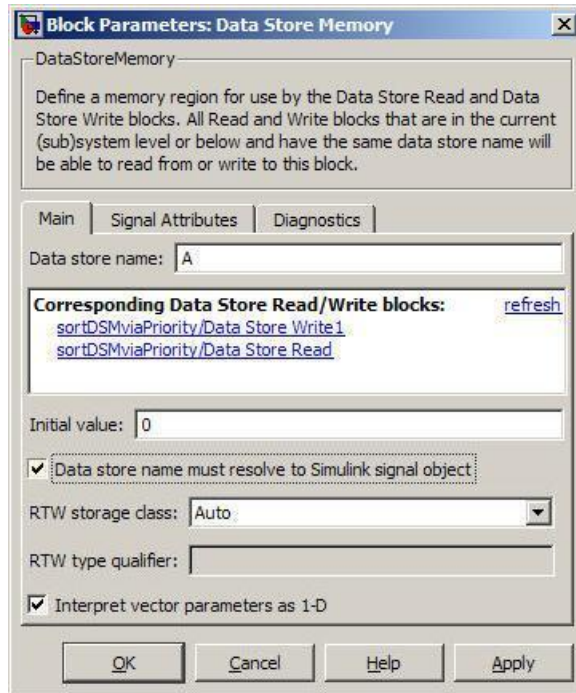
Specifying Attributes Using Block Parameters

You can use the Data Store Memory block dialog box to specify the data type and complexity of a data store. The next figure illustrates such a specification:



Specifying Attributes Using a Signal Object

You can use a `Simulink.Signal` object to specify data store attributes. The technique is the same whether the data object is explicitly associated with a Data Store Memory block, or establishes an implicit data store, as described in “Defining Data Stores with Signal Objects” on page 16-9. The next figure shows a Data Store Memory block named A that specifies resolution to a `Simulink.Signal` object, and the fields of that signal object as shown in the MATLAB Command Window:



A =

```

Simulink.Signal (handle)
  RTWInfo: [1x1 Simulink.SignalRTWInfo]
  Description: ''
  DataType: 'uint16'
  Min: -4
  Max: 4
  DocUnits: ''
  Dimensions: -1
  Complexity: 'real'
  SampleTime: 1
  SamplingMode: 'Sample based'
  InitialValue: ''

```

The signal object specifies values for all three data attributes that the data store would otherwise inherit: `DataType`, `Complexity`, and `SampleTime`.

Defining Data Stores with Signal Objects

In this section...

“Creating the Data Store” on page 16-9

“Local and Global Data Stores” on page 16-9

“Specifying Signal Object Data Store Attributes” on page 16-9

Creating the Data Store

To use a `Simulink.Signal` object to define a data store, create the object in a workspace that is visible to every component that needs to access the data store. Simulink creates an associated data store when you use the signal object for data storage. 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.

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.

Specifying Signal Object Data Store Attributes

Data store attributes that a signal object does not define have the same default values that they do in a Data Store Memory block. Some restrictions exist on the parameter values of a signal object used as a data store. These restrictions vary depending on whether the data store is global or local.

Parameter	Local Store Value	Global Store Value
<code>DataType</code>	Can be set or inherited	Must be set explicitly

Parameter	Local Store Value	Global Store Value
Complexity	Can be set or inherited	Must be set explicitly
SampleTime	Can be set or inherited	Can be set or inherited
SamplingMode	Can be set or inherited	Must be 'Sample based'

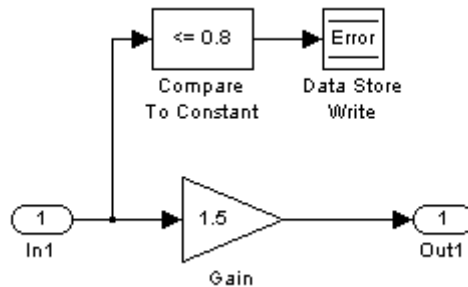
Once you have created the object, set the properties of the signal object to the values that you want the corresponding data store properties to have. For example, the following commands define a data store named `Error` in the MATLAB base workspace:

```
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;
```

Accessing Data Stores with Simulink Blocks

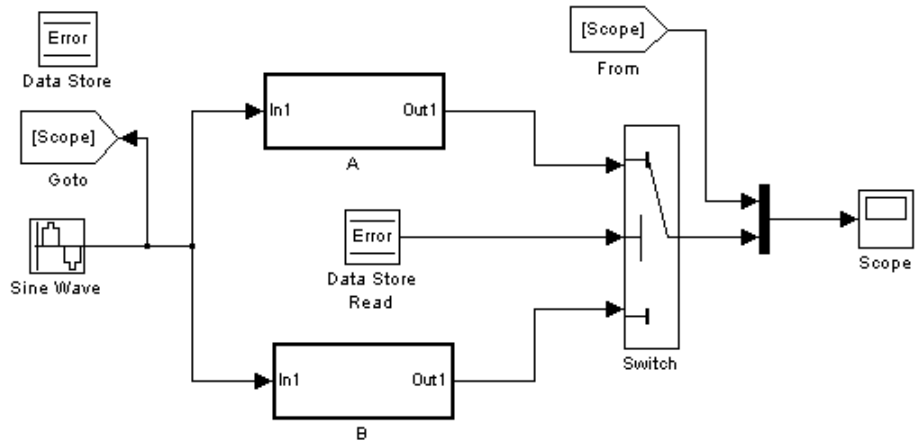
To set the value of a data store at each time step:

- 1 Create an instance of a Data Store Write block at the level of your model that computes the value.
- 2 Set the block's **Data store name** parameter to the name of the data store to be updated.
- 3 Connect the output of the block that computes the value to the input of the Data Store Write block.

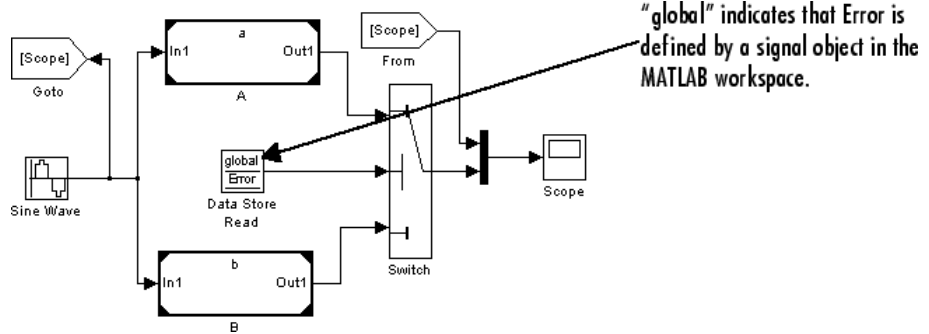


To get the value of a data store at each time step:

- 1 Create an instance of a Data Store Read block at the level of your model that needs the value.
- 2 Set the block's **Data store name** parameter to the name of the data store to be read.
- 3 Connect the output of the data store read block to the input of the block that needs the data store's value.



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's name.



Data Store Examples

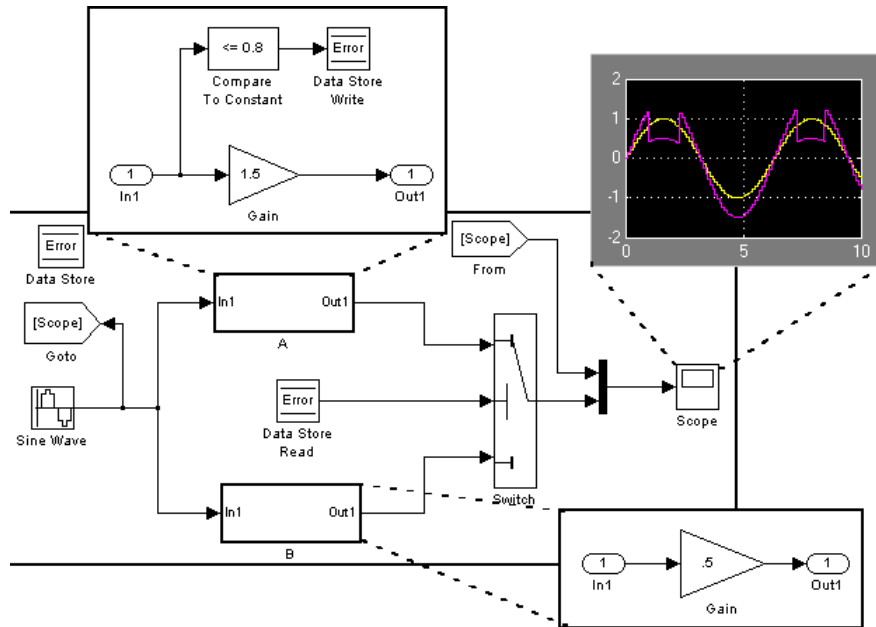
In this section...
“Overview” on page 16-13
“Local Data Store Example” on page 16-13
“Global Data Store Example” on page 16-14

Overview

The following examples illustrate techniques for defining and accessing data stores. See “Ordering Data Store Access” on page 16-16 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 “Using Data Store Diagnostics” on page 16-19 for techniques you can use to help detect and correct potential data store errors without needing to run any simulations.

Local Data Store Example

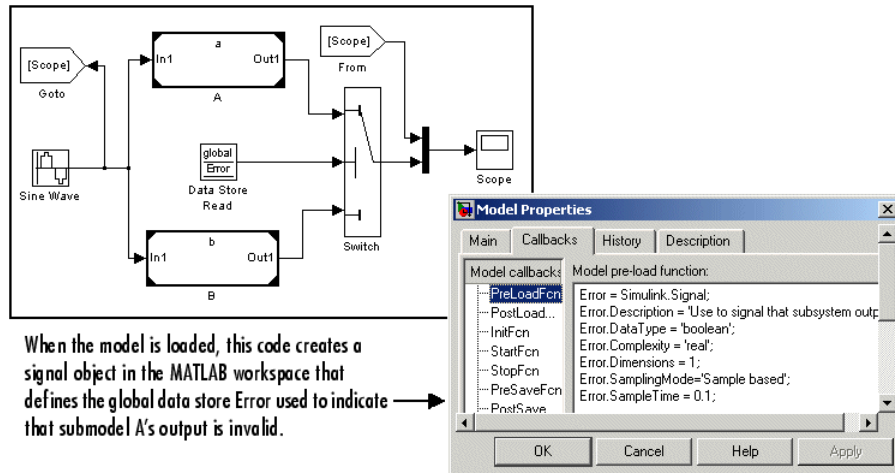
The following model illustrates creation and access of a local data store, which is visible only in a model or particular subsystem.



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.

Ordering Data Store Access

In this section...
“About Data Store Access Order” on page 16-16
“Ordering Access Using Function Call Subsystems” on page 16-16
“Ordering Access Using Block Priorities” on page 16-17

About Data Store Access Order

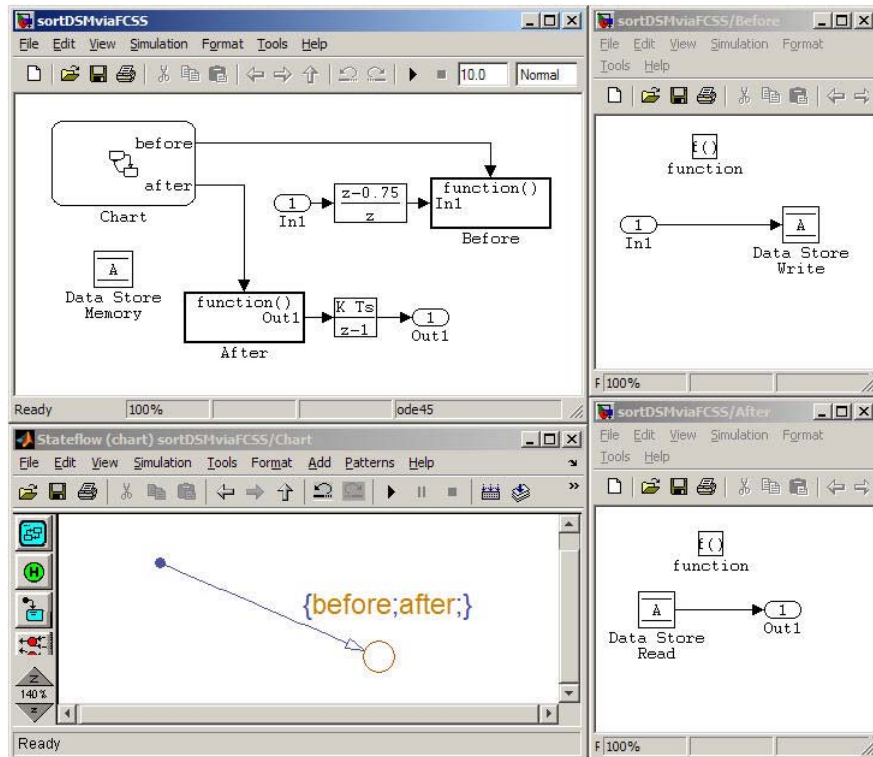
To obtain correct results from data stores, you must control the order of execution of the data store’s reads and writes. If a data store’s read occurs before its write, latency is introduced into the algorithm: the read obtains the value that was computed and stored in the previous time step, rather 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 “Using Data Store Diagnostics” on page 16-19 for techniques you can use to detect and correct potential data store errors without running simulations.

Ordering Access Using Function Call Subsystems

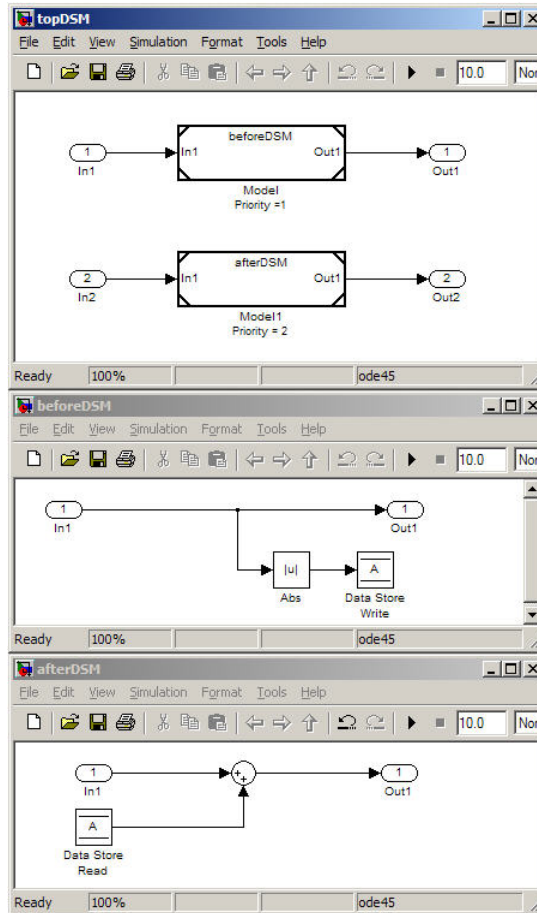
You can use function call subsystems to control the execution order of model components that access data stores. The next figure shows this technique:



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 next figure shows this technique:



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.

Using Data Store Diagnostics

In this section...

“About Data Store Diagnostics” on page 16-19

“Detecting Access Order Errors” on page 16-19

“Detecting Multitasking Access Errors” on page 16-22

“Detecting Duplicate Name Errors” on page 16-24

“Data Store Diagnostics in the Model Advisor ” on page 16-26

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 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:

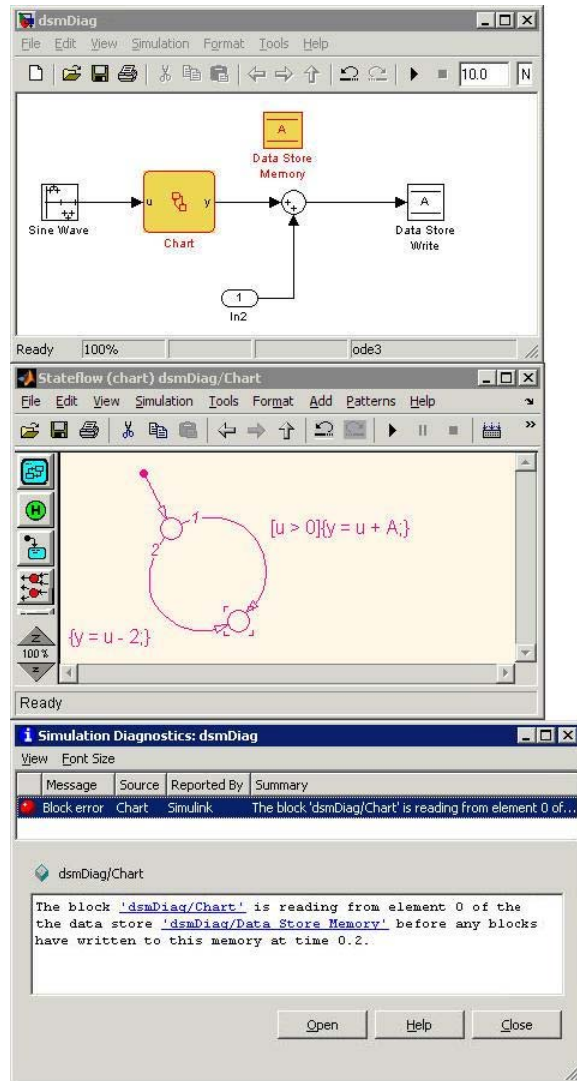
- **Detect read before write:** Detect when a data store is read from before written to within a given time step
- **Detect write after read:** Detect when a data store is written to after being read from within a given time step
- **Detect write after write:** Detect when a data store is written to multiple times within a given time step

These diagnostics appear in the **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's **Diagnostics** tab, where each can have the value `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 Data Store Memory for more information.

The most conservative technique is to set all data store diagnostics to `Enable all as errors` in **Configuration Parameters > Diagnostics > Data Validity > Data Store Memory Block**. However, this setting is not best in all cases, because it can flag intended behavior as erroneous. For example, the next figure shows a model that uses block priorities to force the Data Store Read block to execute before the Data Store Write block:



An error occurred during simulation because the data store A is read from the 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.

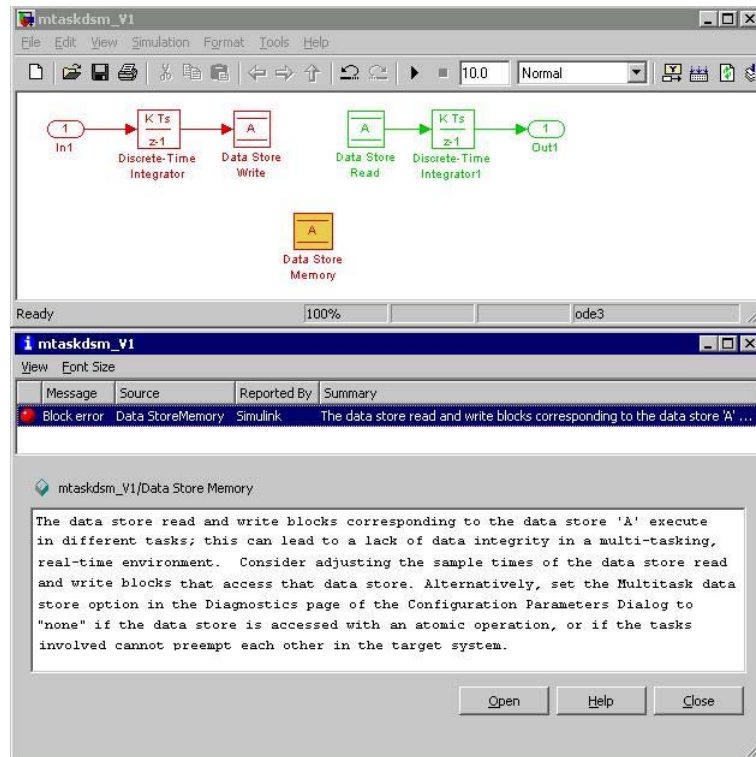
Detecting Multitasking Access Errors

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.

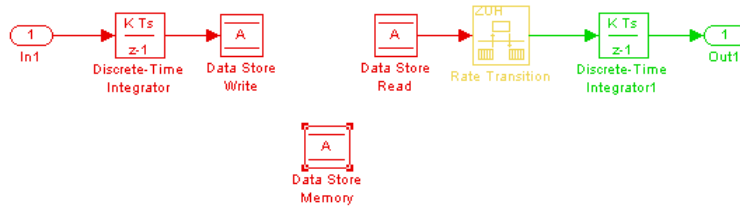
If the first task had only partly updated the data store when the second task interrupted, the resulting data in the store will be inconsistent. For example, if the value is a vector, some of its elements may have been written in the current time step, while the rest were written in the previous step. If the value is a multi-word datum, 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 **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.



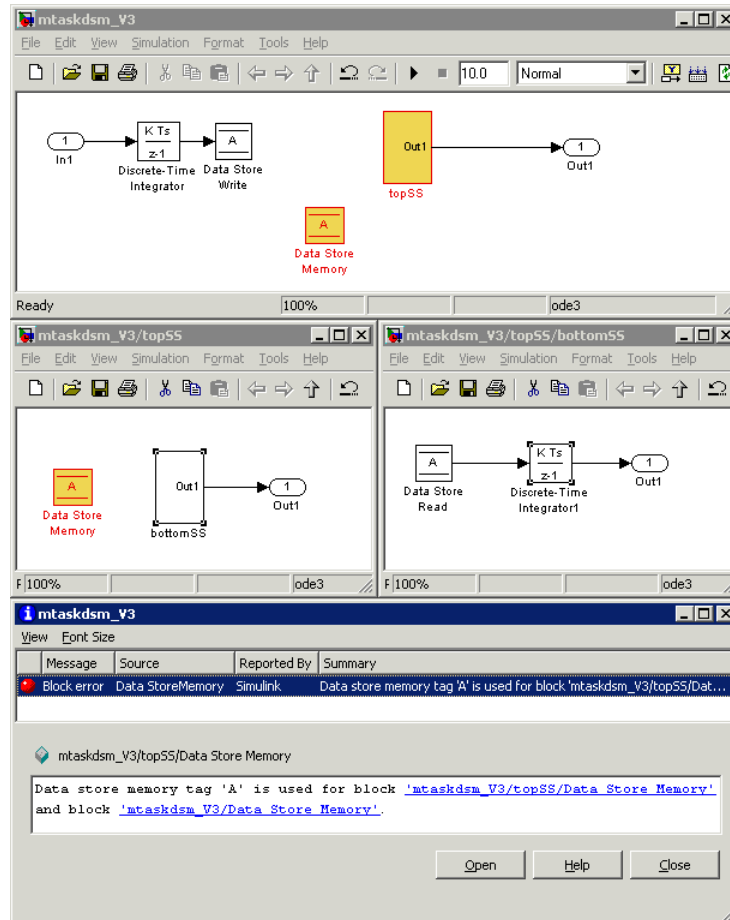
With this change, the data store write can continue to occur at the faster rate. This may be important if that data store must be read at that faster rate elsewhere in the model.

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 **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 for proper usage of Data Store Memory blocks”

“Check sample times of Data Store blocks”

“Runtime diagnostics 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-178B, “Software Considerations in Airborne Systems and Equipment Certification,” Section 6.3.3b.

Working with Lookup Tables

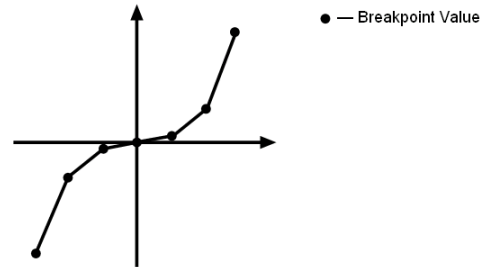
- “About Lookup Table Blocks” on page 17-2
- “Anatomy of a Lookup Table” on page 17-5
- “Lookup Tables Block Library” on page 17-6
- “Guidelines for Choosing a Lookup Table” on page 17-8
- “Entering Breakpoints and Table Data” on page 17-11
- “Characteristics of Lookup Table Data” on page 17-18
- “Methods for Estimating Missing Points” on page 17-23
- “Lookup Table Editor” on page 17-28
- “Example of a Logarithm Lookup Table” on page 17-39
- “Examples for Prelookup and Interpolation Blocks” on page 17-43
- “Lookup Table Glossary” on page 17-44

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, the Simulink software 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.

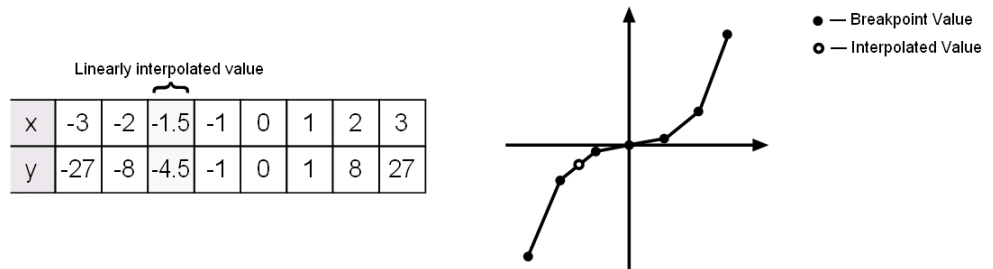
The following example illustrates a one-dimensional lookup table that approximates the function $y = x^3$. The lookup table defines its output (y) data discretely over the input (x) range $[-3, 3]$. The following table and graph illustrate the input/output relationship:

x	-3	-2	-1	0	1	2	3
y	-27	-8	-1	0	1	8	27

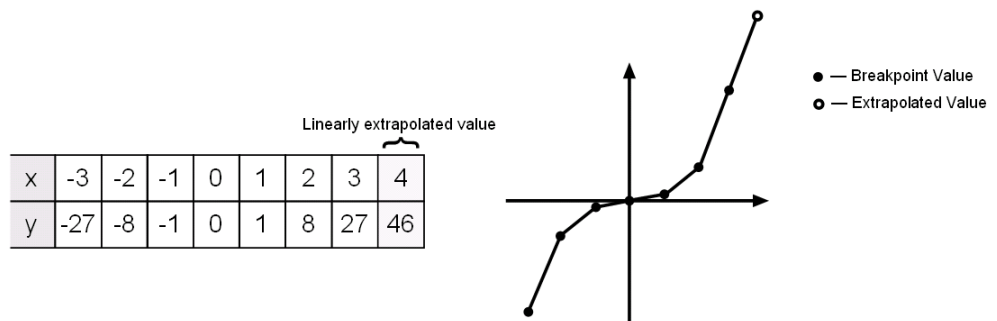


An input of -2 enables the table to look up and retrieve the corresponding output value (-8). Likewise, the lookup table outputs 27 in response to an input of 3.

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 may result in speed gains when simulating a model. Consider using lookup tables in lieu of mathematical function evaluations when

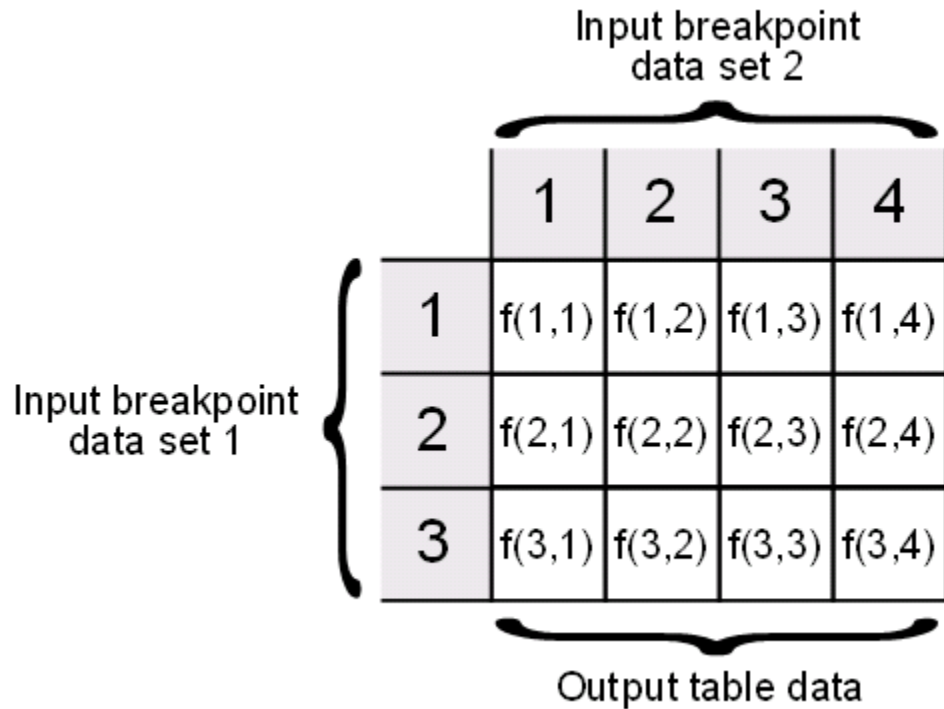
- An analytical expression is expensive to compute
- No analytical expression exists, but the relationship has been determined empirically

The Simulink software 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

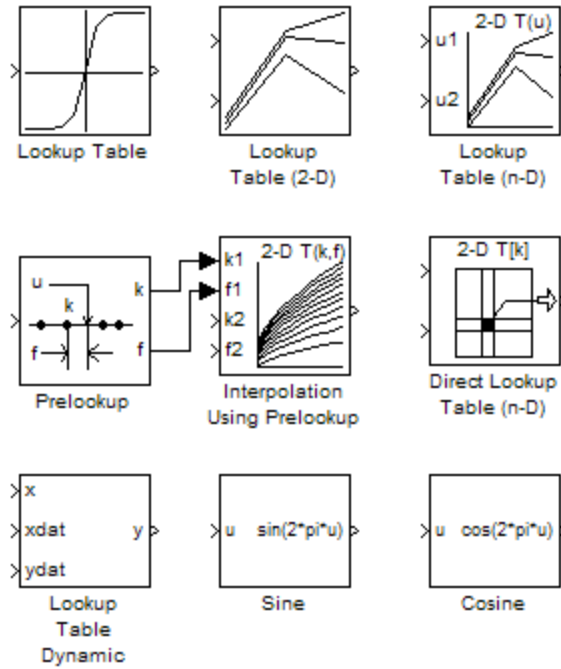
The following figure illustrates the anatomy of a two-dimensional lookup table. Vectors or *breakpoint data sets* and an array, referred to as *table data*, constitute the lookup table.



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
Lookup Table	Approximate a one-dimensional function.
Lookup Table (2-D)	Approximate a two-dimensional function.
Lookup Table (n-D)	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 17-8

“Data Set Numeric and Data Types” on page 17-8

“Data Accuracy and Smoothness” on page 17-8

“Dynamics of Table Inputs” on page 17-9

“Efficiency of Performance” on page 17-9

“Summary of Lookup Table Block Features” on page 17-10

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 Lookup Table or Lookup Table Dynamic block. If you are approximating a two-dimensional function, consider the Lookup Table (2-D) block. Blocks such as the Lookup Table (n-D) and Direct Lookup Table (n-D) allow you to approximate an N -dimensional function of even higher order.

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) and Lookup Table (n-D) blocks are the only lookup table blocks that support complex table data. Most lookup table blocks support integer and fixed-point data in addition to `double` and `single` data types.

Data Accuracy and Smoothness

The desired accuracy and smoothness of the data returned by a lookup table determine which of the blocks should be used. 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, Lookup Table (2-D), and Lookup Table Dynamic blocks

perform linear interpolation and extrapolation, while the Lookup Table (n-D) 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 and 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 Lookup Table (n-D) 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. Certain lookup table blocks also provide a search algorithm that is tailored for breakpoint data sets composed of evenly spaced breakpoints. Note that 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, in which it 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, the block allows you to specify an input used to select and interpolate the 2-D tables. These features make table lookup operations more efficient, reducing computational effort and thus simulation time.

Summary of Lookup Table Block Features

The following table summarizes many features of lookup table blocks in Simulink. Use the table to identify features that correspond to particular lookup table blocks, then select the block that best meets your requirements.

Feature	Lookup Table	Lookup Table (2-D)	Lookup Table Dynamic	Lookup Table (n-D)	Direct Lookup Table (n-D)	Prelookup	Interp. Using Prelookup
Interpolation Methods							
Flat (none)	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline				•			
Extrapolation Methods							
Clipping	•	•	•	•	•	•	•
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 table data			•		•		
Input range checking				•	•	•	•

Entering Breakpoints and Table Data

In this section...

“Entering Data in a Block Parameter Dialog Box” on page 17-11

“Entering Data in the Lookup Table Editor” on page 17-13

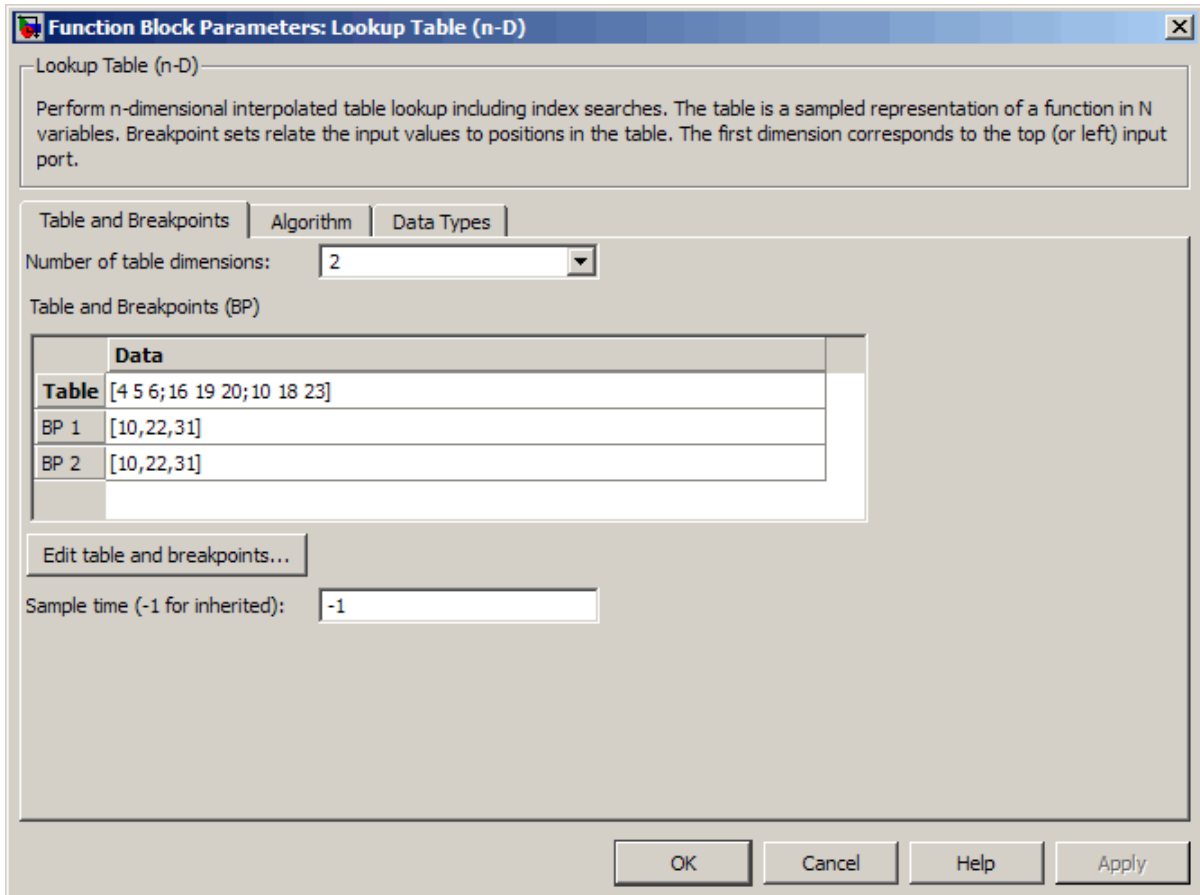
“Entering Data Using the Lookup Table Dynamic Block’s Inports” on page 17-16

Entering Data in a Block Parameter Dialog Box

Use the following procedure to populate a Lookup Table (n-D) 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 Lookup Table (n-D) block from the Lookup Tables block library to a Simulink model.
- 2 In the model window, double-click the Lookup Table (n-D) block.

The block parameter dialog box appears.

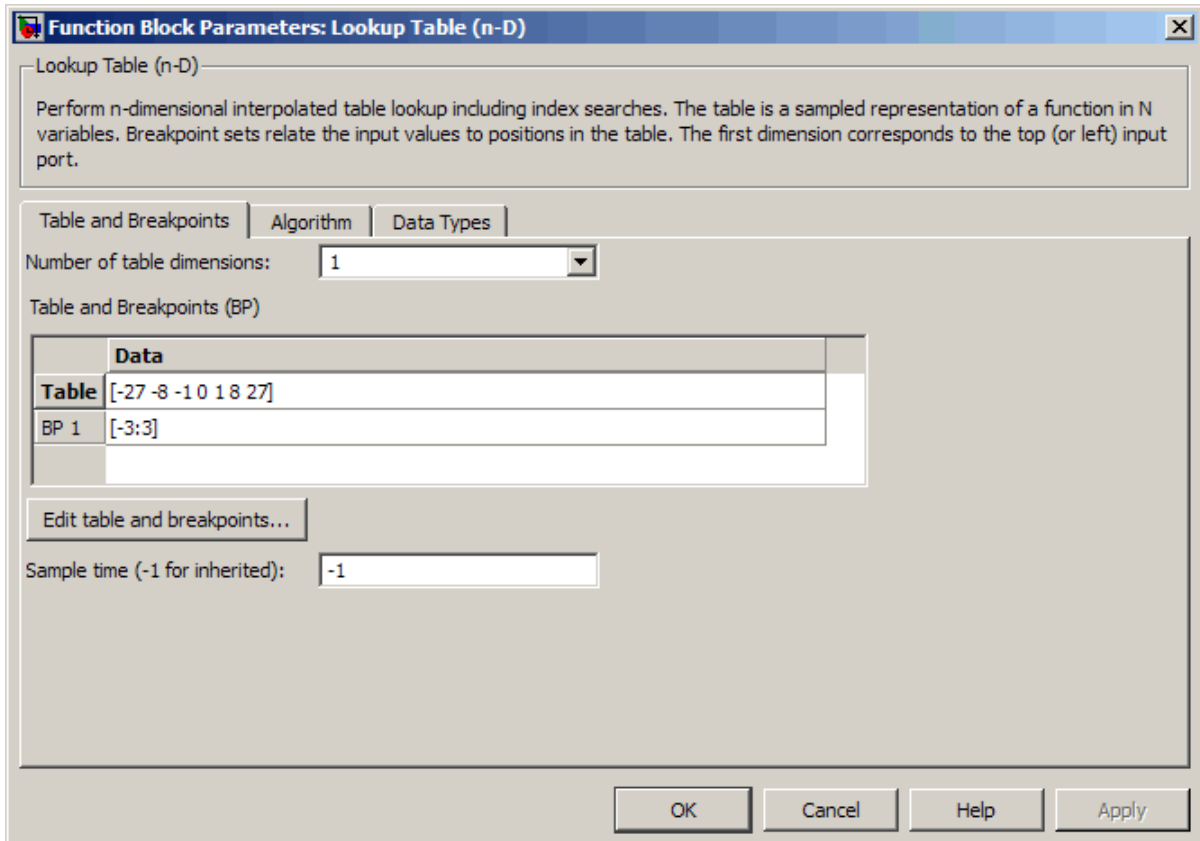


The dialog box displays the default values for the block.

- 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** field, enter [-27 -8 -1 0 1 8 27].
 - In the **BP 1** field, enter [-3:3].

- Click **Apply**.

The block dialog box looks something like this:



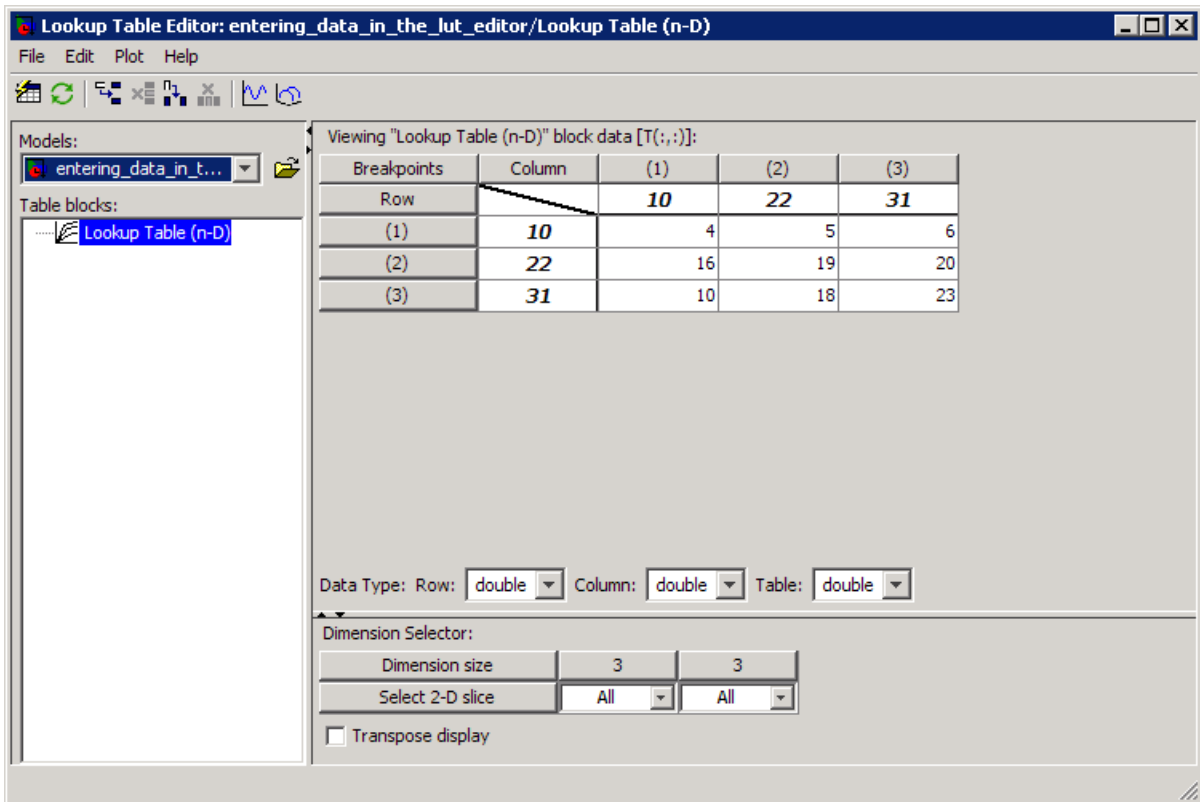
- 4 Click **OK** to apply the changes and close the dialog box.

Entering Data in the Lookup Table Editor

Use the following procedure to populate a Lookup Table (n-D) 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 Lookup Table (n-D) block from the Lookup Tables block library to a Simulink model window.
- 2 Open the Lookup Table Editor by selecting **Lookup Table Editor** from the Simulink **Tools** menu or by clicking the **Edit** button on the parameter dialog box of the Lookup Table (n-D) block.

The Lookup Table Editor appears.



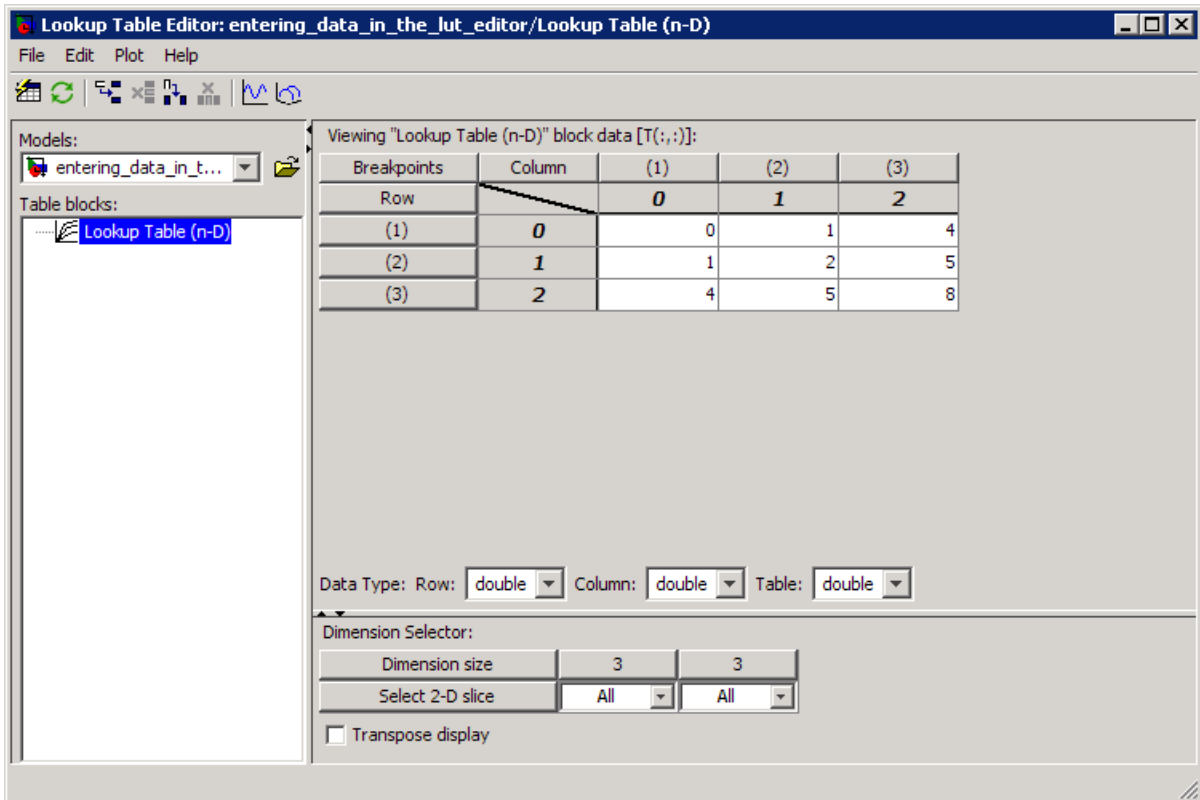
It displays the default data for the Lookup Table (n-D) block.

- 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 similar to the following.



- 4** In the Lookup Table Editor, select **File > Update Block Data** to update the data in the Lookup Table (n-D) block.
- 5** Close the Lookup Table Editor.

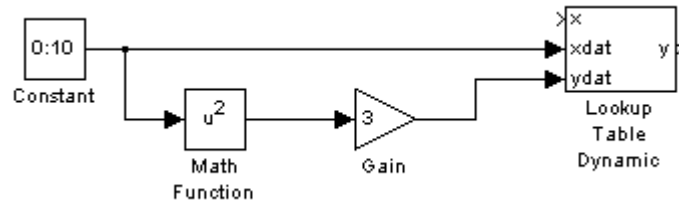
Entering Data Using the Lookup Table Dynamic Block's Inports

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 window.
- 2** Copy the blocks needed to implement the equation $y = 3x^2$ to the Simulink model window:
 - A Constant block to define the input range, from the Sources library
 - A Math Function block to square the input range, from the Math Operations library
 - A 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 parameter dialog boxes:
 - Specify **0:10** as the Constant block's **Constant value** parameter.
 - Specify **square** as the Math Function block's **Function** parameter.
 - Specify **3** as the Gain block's **Gain** parameter.
- 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 input of the Lookup Table Dynamic block labeled **ydat**. This signal is the table data for y .

The model should look similar to the following.



Characteristics of Lookup Table Data

In this section...

“Sizes of Breakpoint Data Sets and Table Data” on page 17-18

“Monotonicity of Breakpoint Data Sets” on page 17-19

“Formulation of Evenly-Spaced Breakpoints” on page 17-20

“Representation of Discontinuities in Lookup Tables” on page 17-20

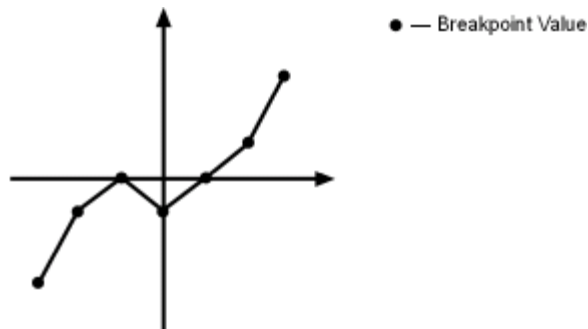
Sizes of Breakpoint Data Sets and Table Data

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*, i.e., 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.

Formulation of Evenly-Spaced Breakpoints

You can represent evenly-spaced breakpoints in a data set by using one of these methods.

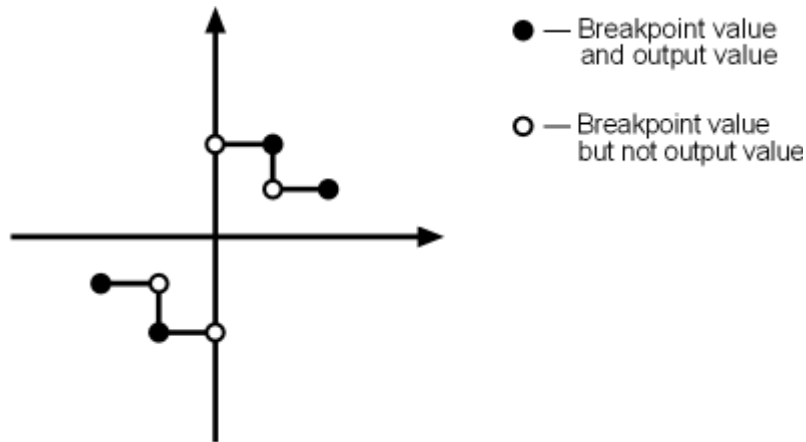
Formulation	Example	When to Use This Formulation
[first_value:spacing:last_value]	[10:10:110]	The lookup table does <i>not</i> use floating-point data types.
(spacing * [first_value/spacing:last_value/spacing])	(0.02 * [0:500])	The lookup table uses floating-point data types.

Because floating-point data types cannot precisely represent some numbers, the second formulation works better for those types. For example, use (0.02 * [0:500]) instead of [0:0.02:10].

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]
```



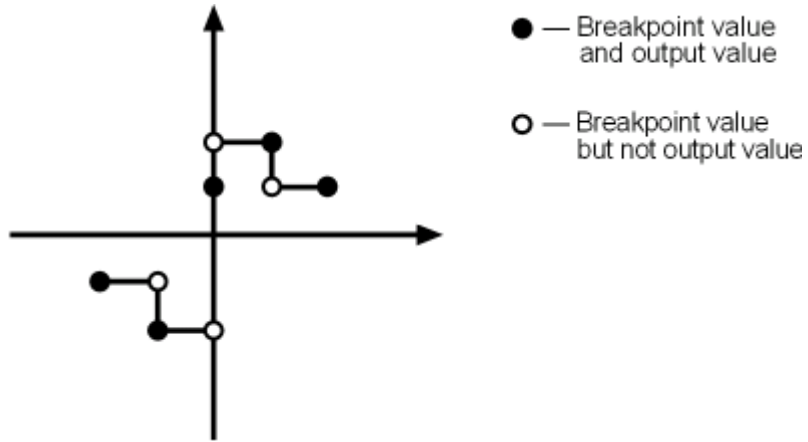
This example has discontinuities at $x = -1, 0,$ and $+1$.

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: $[-2 -1 -1 0 0 0 1 1 2]$
 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.

Methods for Estimating Missing Points

In this section...

“About Estimating Missing Points” on page 17-23

“Interpolation Methods” on page 17-23

“Extrapolation Methods” on page 17-24

“Rounding Methods” on page 17-25

“Example Output for Lookup Methods” on page 17-26

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:

- **None (Flat)** — Disables interpolation and uses the rounding operation titled `Use Input Below`. For more information, see “Rounding Methods” on page 17-25.
- **Linear interpolation** — Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input.
- **Cubic spline interpolation** — Fits a cubic spline to the adjacent breakpoints, and returns the point on that spline corresponding to the input.

Note Blocks such as the Lookup Table Dynamic block do not allow you to choose an interpolation method. The Interpolation-Extrapolation option in the **Lookup Method** field of its 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 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:

- **None (Clip to Range) or (Use End Values)** — Disables extrapolation and returns the table data corresponding to the end of the breakpoint data set range.
- **Linear extrapolation** — 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 extrapolation** — 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 Blocks such as the Lookup Table Dynamic block do not allow you to choose an extrapolation method. The Interpolation-Extrapolation option in the **Lookup Method** field of its block parameter dialog box performs linear extrapolation.

In addition to these methods, some lookup table blocks, such as the Lookup Table (n-D) 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 **Action for out-of-range input** list on the block parameter dialog box.

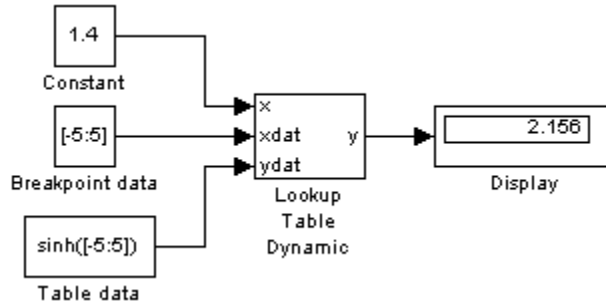
Rounding Methods

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. Most lookup table blocks let 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

Suppose 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 $\sinh(1.0)$.

Lookup Table Editor

In this section...
“When to Use the Lookup Table Editor” on page 17-28
“Layout of the LUT Editor” on page 17-28
“Browsing LUT Blocks” on page 17-30
“Editing Table Values” on page 17-31
“Adding and Removing Rows and Columns in a Table” on page 17-31
“Displaying N-Dimensional Tables in the Editor” on page 17-32
“Plotting LUT Tables” on page 17-35
“Editing Custom LUT Blocks” on page 17-36

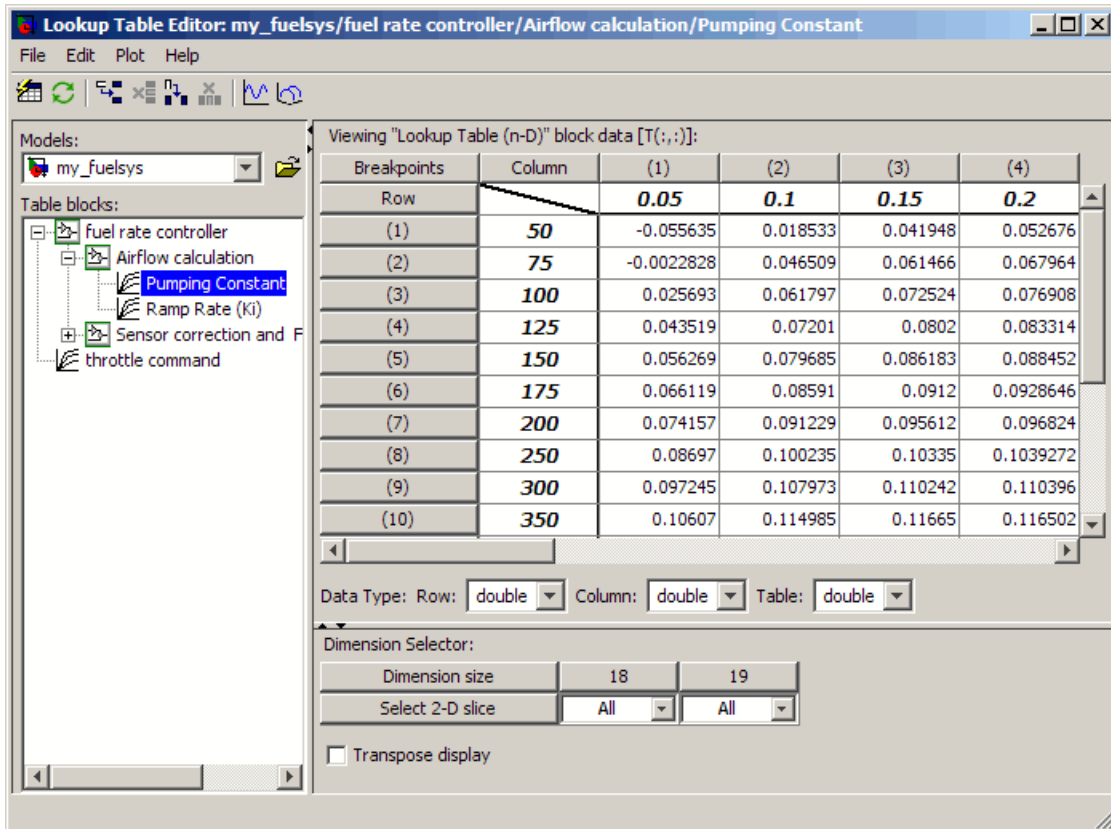
When to Use the Lookup Table Editor

Use the Lookup Table Editor to inspect and change the table elements of any lookup table (LUT) block in a model, including custom LUT blocks that you create using the Simulink Mask Editor (see “Editing Custom LUT Blocks” on page 17-36). 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 LUT Editor, you can skip these steps.

Note You cannot use the LUT Editor to change the dimensions of a lookup table. You must use the block parameter dialog box for this purpose.

Layout of the LUT Editor

To open the editor, select **Lookup Table Editor** from the Simulink **Tools** menu. The editor appears.

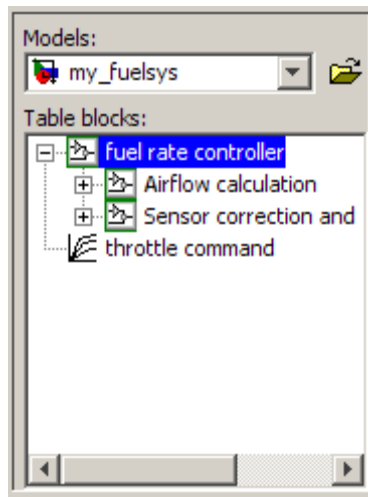


The editor contains two panes and a toolbar.

- Use the left pane to browse and select LUT blocks in any open model (see “Browsing LUT Blocks” on page 17-30).
- Use the right pane to edit the lookup table of the selected block (see “Editing Table Values” on page 17-31).
- 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 LUT Blocks

The **Models** list in the upper-left corner of the LUT Editor lists the names of all models open in the current MATLAB session. To browse LUT table blocks for any open models, select the model name from the list. A tree-structured view of LUT blocks for the selected model appears in the **Table blocks** field beneath the **Models** list.



The tree view initially lists all LUT blocks that reside at the model root level. It also displays any subsystems that contain LUT blocks. Clicking the expand button (+) to the left of the subsystem name expands the tree to show LUT blocks in that subsystem. The expanded view also shows any subsystems in the expanded subsystem. You can continue expanding subsystem nodes to display LUT blocks at any level in the model hierarchy.

Clicking any LUT 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 17-31).

Note If you want to browse the LUT blocks in a model that is not currently open, you can tell the LUT 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 LUT Editor, you can edit the lookup table of the LUT 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	-0.055635	0.018533	0.041948
(2)	75	-0.0022828	0.046509	0.061466
(3)	100	0.025693	0.061797	0.072524
(4)	125	0.043519	0.07201	0.0802
(5)	150	0.056269	0.079685	0.086183
(6)	175	0.066119	0.08591	0.0912
(7)	200	0.074157	0.091229	0.095612

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 17-32). To change any value that appears, double-click the value. The LUT 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 LUT Editor records your changes by maintaining a copy of the table. To update the copy that the LUT block maintains, select **File > Update Block Data** in the LUT Editor. To restore the LUT Editor’s copy to the values stored in the block, select **File > Reload Block Data**.

Adding and Removing Rows and Columns in a Table

In the LUT 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 LUT 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 LUT block currently selected in the LUT Editor's tree view has more than two dimensions, the table view displays a two-dimensional slice of the table.

Viewing "Lookup Table (n-D)" block data [T(:, :, 1)]:

Breakpoints	Column	(1)	(2)	(3)	(4)	(5)	(6)
Row		1	2	3	4	5	6
(1)	0	1	11	21	31	41	51
(2)	10	2	12	22	32	42	52
(3)	20	3	13	23	33	43	53
(4)	30	4	14	24	34	44	54
(5)	40	5	15	25	35	45	55
(6)	50	6	16	26	36	46	56
(7)	60	7	17	27	37	47	57
(8)	70	8	18	28	38	48	58
(9)	80	9	19	29	39	49	59
(10)	90	10	20	30	40	50	60

Data Type: Row: Column: Table:

Dimension Selector:

Dimension size	10	6	5
Select 2-D slice	<input type="text" value="All"/>	<input type="text" value="All"/>	<input type="text" value="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 top row of the selector array displays the size of each dimension. The remaining rows specify 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, click the **Select row axis** and **Select column axis** radio buttons in the columns that correspond to the dimensions that you want to view. Then select the indexes of the slice from the pop-up index lists in the remaining columns.

To transpose the table display, select the **Transpose display** check box.

For example, the following selector displays slice (,;,1) of a 3-D lookup table.

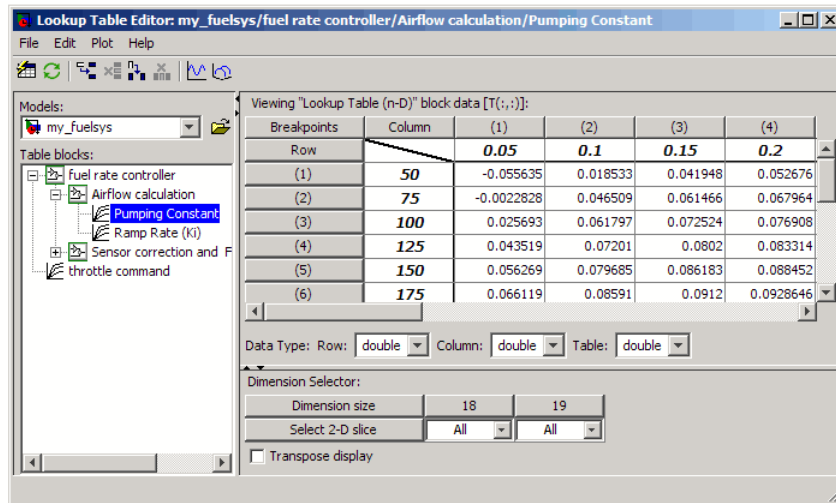
Dimension Selector:

Dimension size	10	6	5
Select 2-D slice	All ▾	All ▾	1 ▾

Transpose display

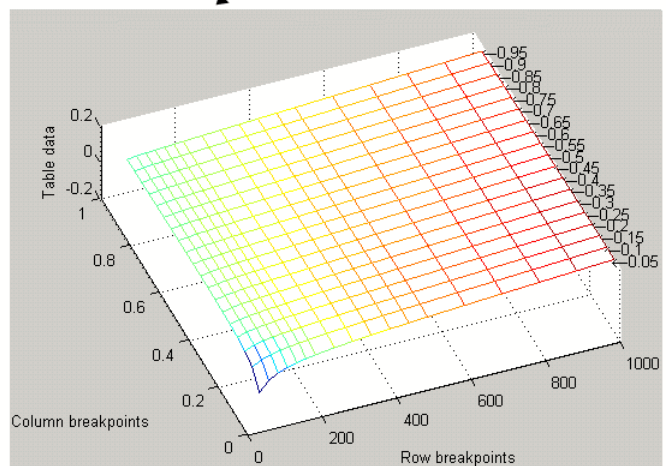
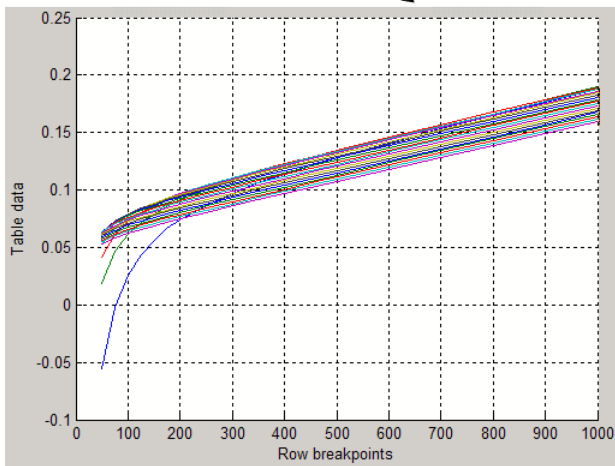
Plotting LUT Tables

To display a linear or mesh plot of the table or table slice in the LUT Editor, select **Plot > Linear** or **Plot > Mesh**.



Linear

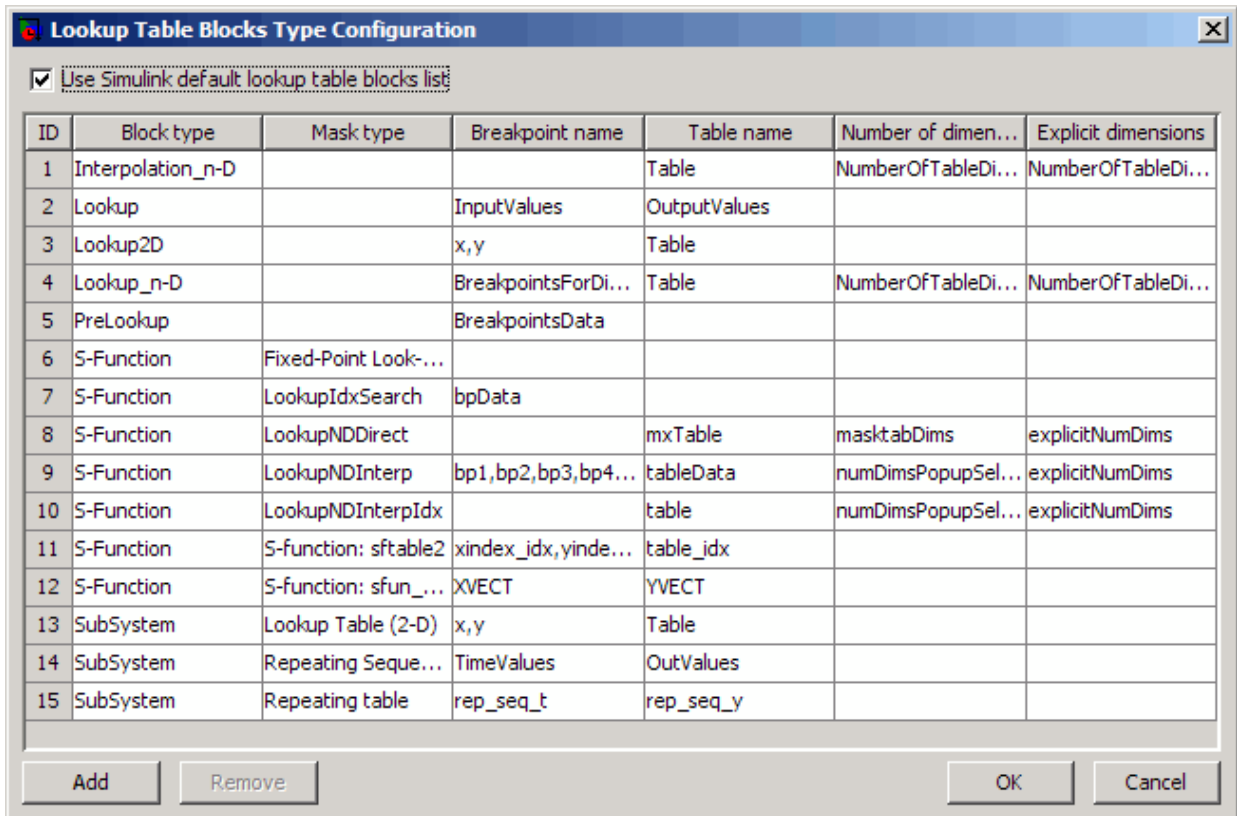
Mesh



Editing Custom LUT Blocks

You can use the LUT Editor to edit custom lookup table blocks that you or others have created. To do this, you must first configure the LUT Editor to recognize the custom LUT blocks in your model. Once you have configured the LUT Editor to recognize custom blocks, you can edit them as if they were standard blocks.

To configure the LUT Editor to recognize custom LUT blocks, select **File > Configure**. The **Lookup Table Blocks Type Configuration** dialog box appears.



By default, the dialog box displays a table of the LUT block types that the LUT Editor currently recognizes. This table includes the standard LUT blocks. Each row of the table displays key attributes of a LUT block type.

Adding a Custom LUT 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 LUT block. The block type is the value of the block's <code>BlockType</code> parameter.
Mask type	Mask type of the custom LUT block. The mask type is the value of the block's <code>MaskType</code> parameter.
Breakpoint name	Names of the custom LUT block's parameters that store its breakpoints.
Table name	Name of the block parameter that stores the custom block's lookup table.
Number of dimensions	Leave empty.
Explicit dimensions	Leave empty.

- 3 Click **OK**.

Removing Custom LUT Block Types

To remove a custom LUT block type from the list recognized by the LUT Editor, select the custom entry in the table in the **Lookup Table Blocks Type Configuration** dialog box. Then click **Remove**.

To remove all custom LUT block types, select the **Use Simulink default lookup table blocks list** check box at the top of the dialog box.

Example of 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 the Simulink model window:

- A Constant block to input the signal, from the Sources library
- A Lookup Table (n-D) block to approximate the common logarithm, from the Lookup Tables library
- A Display block to display the output, from the Sinks library

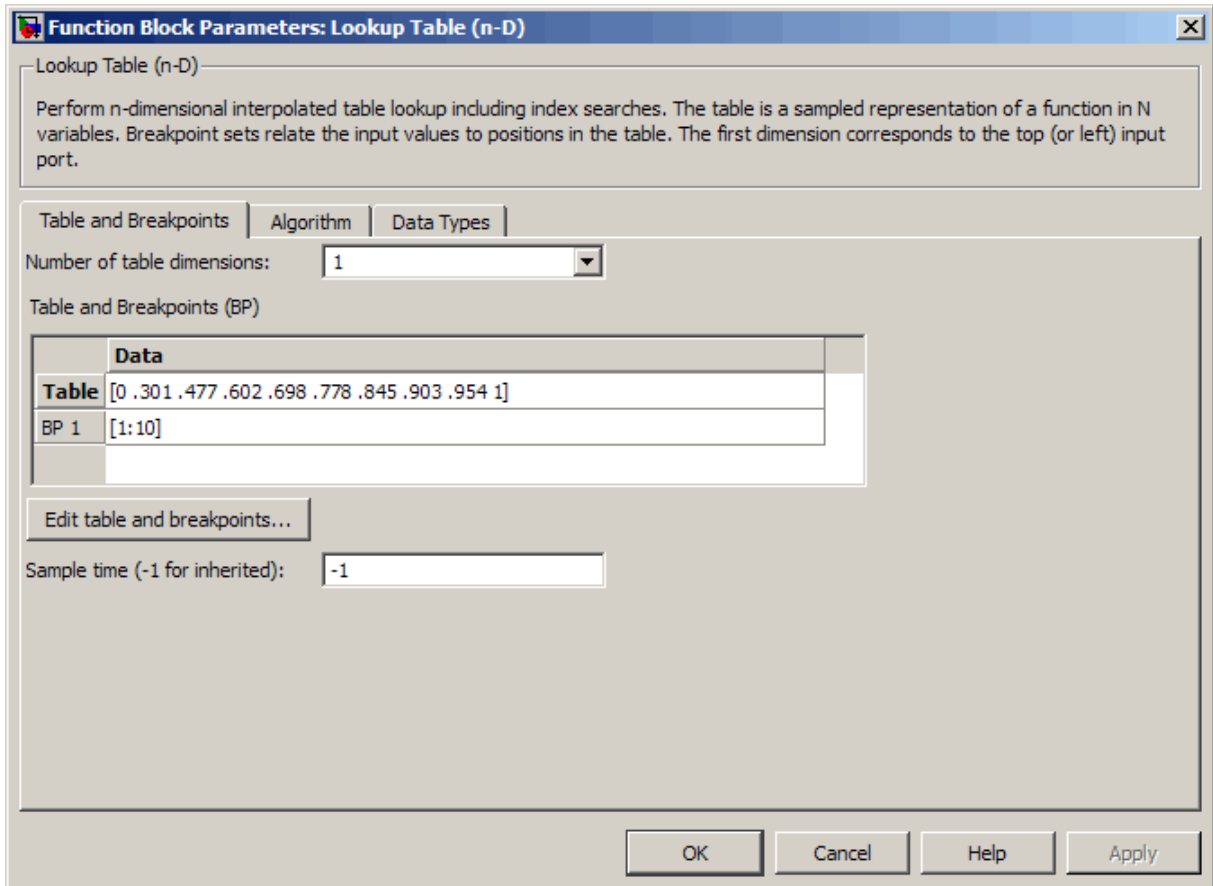
2 Assign the table data and breakpoint data set to the Lookup Table (n-D) block:

- In the **Number of table dimensions** field, enter 1.
- In the **Table** field, enter
[0 .301 .477 .602 .698 .778 .845 .903 .954 1].

Alternatively, you can enter the MATLAB expression `log10(1:10)` in this field, which evaluates to the equivalent vector of output values.

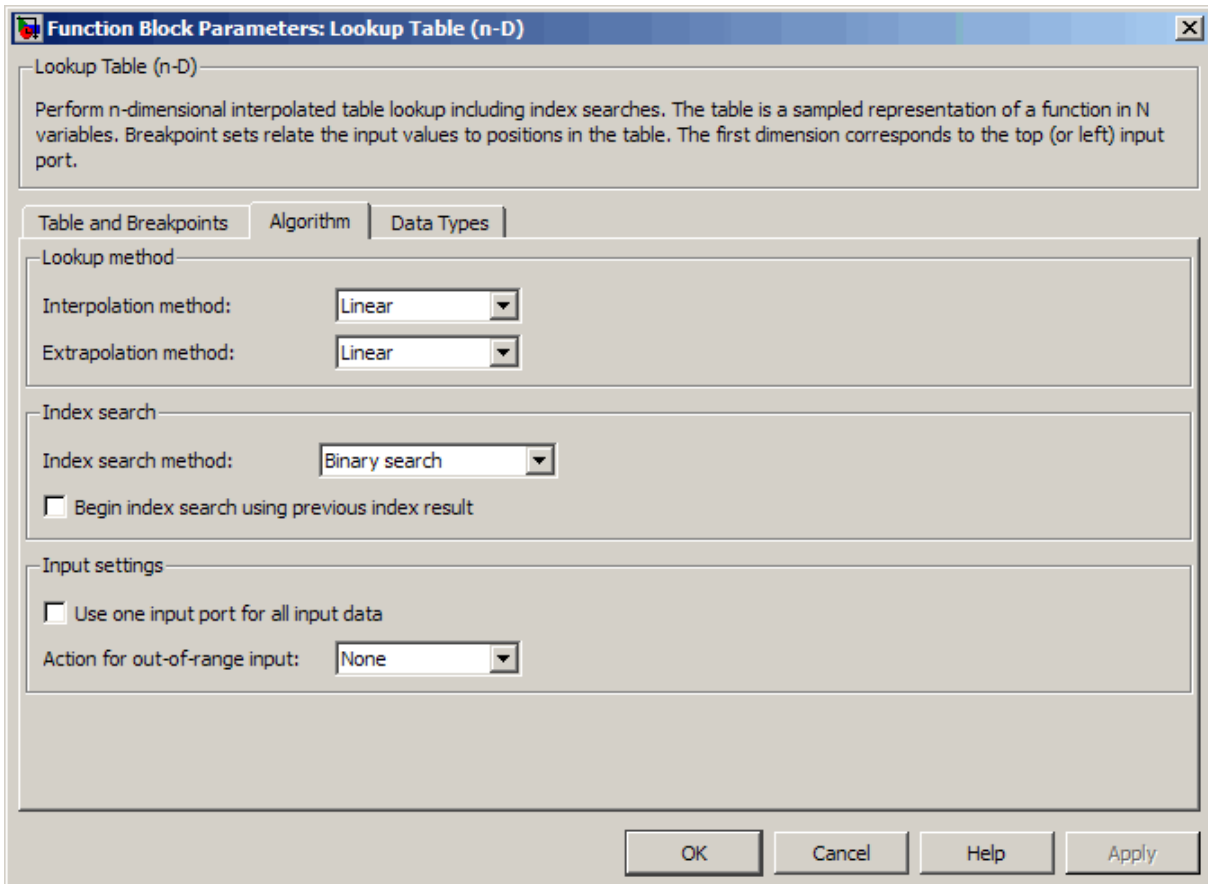
- In the **BP 1** field, enter [1:10].
- Click **Apply**.

The dialog box looks something like this:



3 Click the **Algorithm** tab in the Lookup Table (n-D) block dialog box.

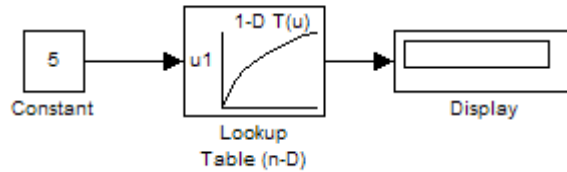
By default, the **Interpolation method** and **Extrapolation method** are both **Linear**.



Click **OK** to close the dialog box.

- 4 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.

5 Connect the blocks as follows.



6 In the model window, select **Simulation > Start** to run the simulation.

The following behavior applies to the Lookup Table (n-D) block.

Value of the Constant Block	Action by the Lookup Table (n-D) Block	Example of Block Behavior	
		Input Value	Output Value
Equals a breakpoint	Returns the corresponding output value	5	0.698
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

Examples for 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	
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 	
	Provides easier sharing of: <ul style="list-style-type: none"> • Breakpoint data among Prelookup blocks • Table data among Interpolation Using Prelookup blocks 	
	Enables reuse of utility functions during Real-Time Workshop code generation	
Set the data type for intermediate results explicitly	Enables use of higher precision for internal computations than for table data or output data	

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 software 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.

Modeling with Simulink

- “General Considerations when Building Simulink Models” on page 18-2
- “Modeling a Continuous System” on page 18-8
- “Best-Form Mathematical Models” on page 18-11
- “Example: Converting Celsius to Fahrenheit” on page 18-15

General Considerations when Building Simulink Models

In this section...

“Avoiding Invalid Loops” on page 18-2

“Shadowed Files” on page 18-4

“Model Building Tips” on page 18-6

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 “Modeling a Continuous System” on page 18-8) 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 “Function-Call Subsystems” on page 5-24 for a description of function-call subsystems)
- Self-triggering subsystems and loops containing non-latched triggered subsystems (see “Triggered Subsystems” on page 5-14 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:

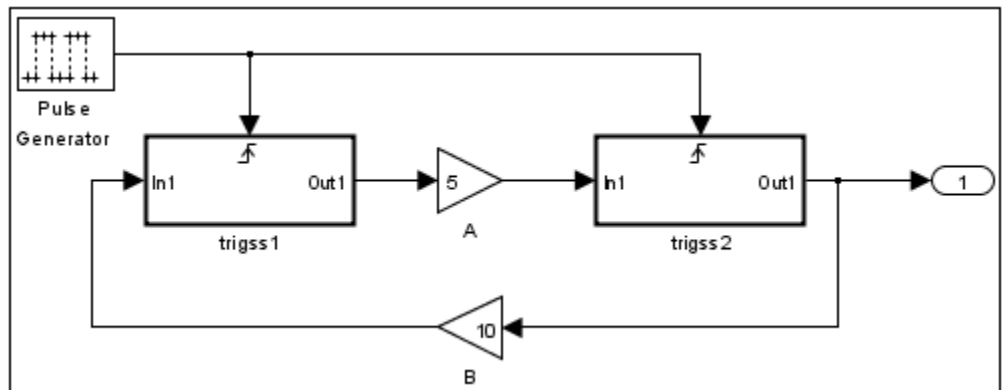
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1(sl_subsys_trigerr1)`
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2(sl_subsys_trigerr2)`

- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call systems/sl_subsys_fcncallerr3 (sl_subsys_fcncallerr3)`

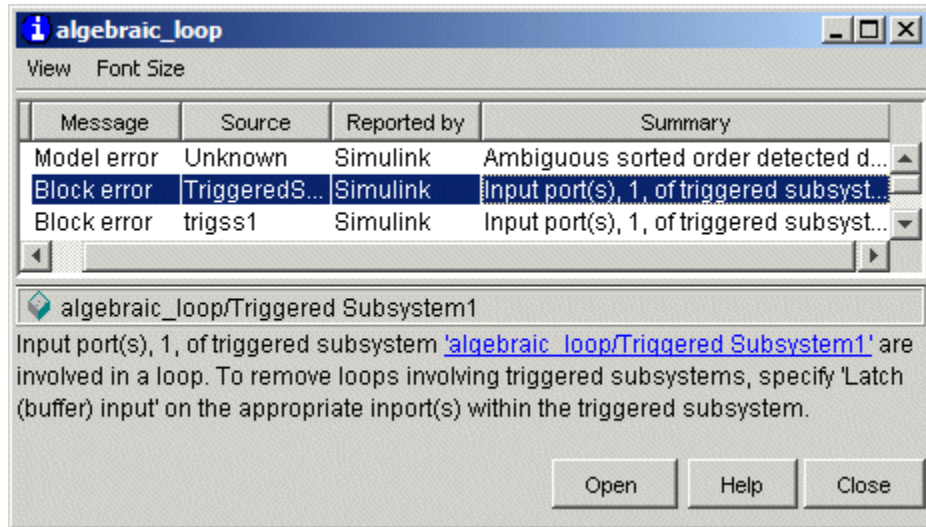
You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update Diagram** from the model's **Edit** 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.mdl`) 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 which the Simulink software uses to find Model files are similar to those used by the MATLAB software. See "How the Search Path Determines Which Function to Use" in the MATLAB documentation. However, there is an important difference between the way in which Simulink block diagram 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 directory 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 directory, 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 demo 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 directory which contains a different library called `hydlib`, and try to run a model in that directory, the library in that directory 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.mdl` 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.mdl` is being used. To see which file called `mylibrary.mdl` is loaded into memory, enter:

```
which mylibrary  
  
C:\work\Model1\mylibrary.mdl
```

To see all the files called `mylibrary` which are on the MATLAB path (note that this can include M-files), enter:

```
which -all mylibrary  
  
C:\work\Model1\mylibrary.mdl  
C:\work\Model2\mylibrary.mdl  % 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 “Creating Subsystems” on page 4-37. The Model Browser provides useful information about complex models (see “The Model Browser” on page 19-26).

- 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 “Naming Signals” on page 10-3 and “Annotating Diagrams” on page 4-26.

- 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.

Modeling a Continuous System

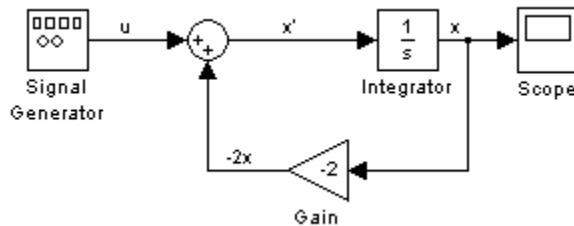
To model the differential equation

$$x'(t) = -2x(t) + u(t)$$

where $u(t)$ is a square wave with an amplitude of 1 and a frequency of 1 rad/sec. 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.

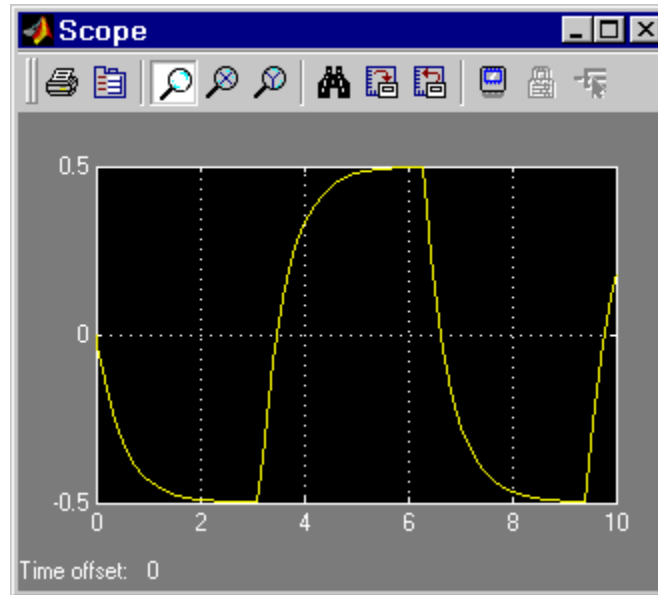
In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see “Drawing a Branch Line” on page 4-19.

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

$$x = u/(s + 2)$$

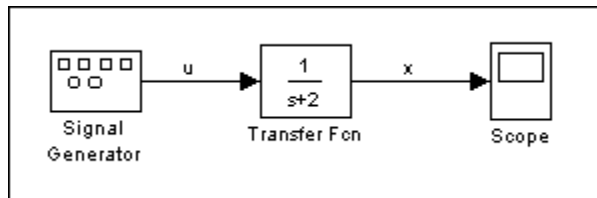
or,

$$x/u = 1/(s + 2)$$

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...

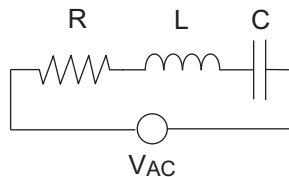
“Series RLC Example” on page 18-11

“Solving Series RLC Using Resistor Voltage” on page 18-12

“Solving Series RLC Using Inductor Voltage” on page 18-13

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

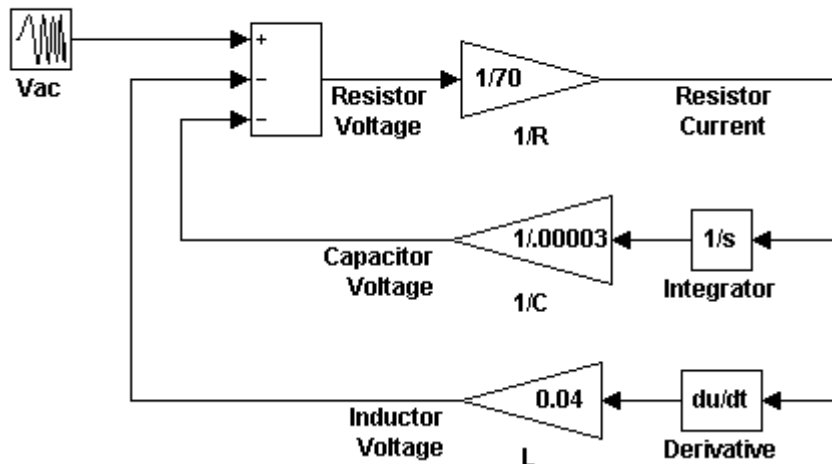
$$V_R = Ri$$

$$Ri = V_{AC} - L \frac{di}{dt} - \frac{1}{C} \int_{-\infty}^t i(t) dt$$

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 through 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 2-35 for more information.

Solving Series RLC Using Inductor Voltage

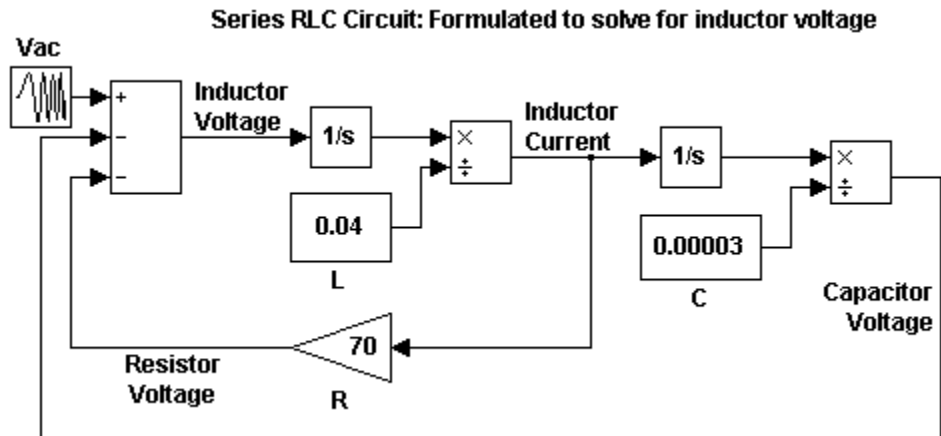
To avoid using a Derivative block, formulate the equation to solve for the inductor voltage.

$$V_L = L \frac{di}{dt}$$

$$L \frac{di}{dt} = V_{AC} - Ri - \frac{1}{C} \int_{-\infty}^t i(t) dt$$

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.

Example: Converting Celsius to Fahrenheit

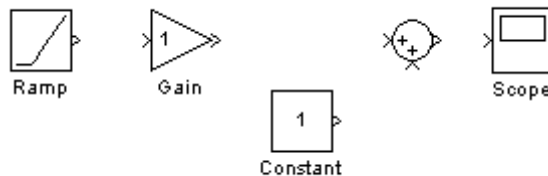
To model the equation that converts Celsius temperature to Fahrenheit

$$T_F = 9/5(T_C) + 32$$

First, consider the blocks needed to build the model:

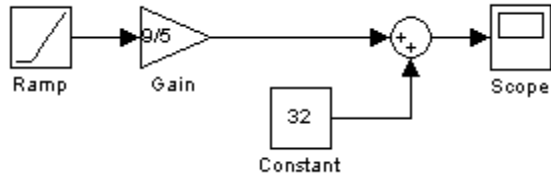
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library
- A Gain block to multiply the input signal by 9/5, from the Math Operations library
- A Sum block to add the two quantities, also from the Math Operations 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 **Start** from the **Simulation** menu to run the simulation. The simulation runs for 10 seconds.

Exploring, Searching, and Browsing Models

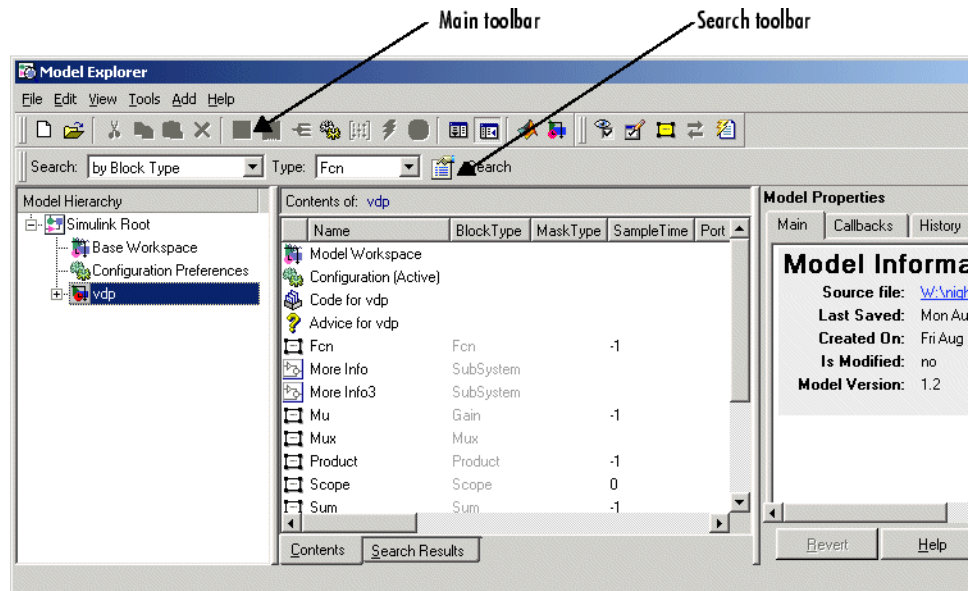
- “The Model Explorer” on page 19-2
- “The Finder” on page 19-20
- “The Model Browser” on page 19-26
- “Model Dependencies” on page 19-29

The Model Explorer

In this section...
“Introduction to the Model Explorer” on page 19-2
“Model Hierarchy Pane” on page 19-3
“Contents Pane” on page 19-6
“Dialog Pane” on page 19-11
“Main Toolbar” on page 19-12
“Search Bar” on page 19-14
“Setting the Model Explorer’s Font Size” on page 19-19

Introduction to the Model Explorer

The Model Explorer allows you to quickly locate, view, and change elements of a Simulink model or Stateflow chart. To display the Model Explorer, select **Model Explorer** from the Simulink **View** menu or select an object in the block diagram and select **Explore** from its context menu. The Model Explorer appears.



Model Hierarchy pane

Contents pane

Dialog pane

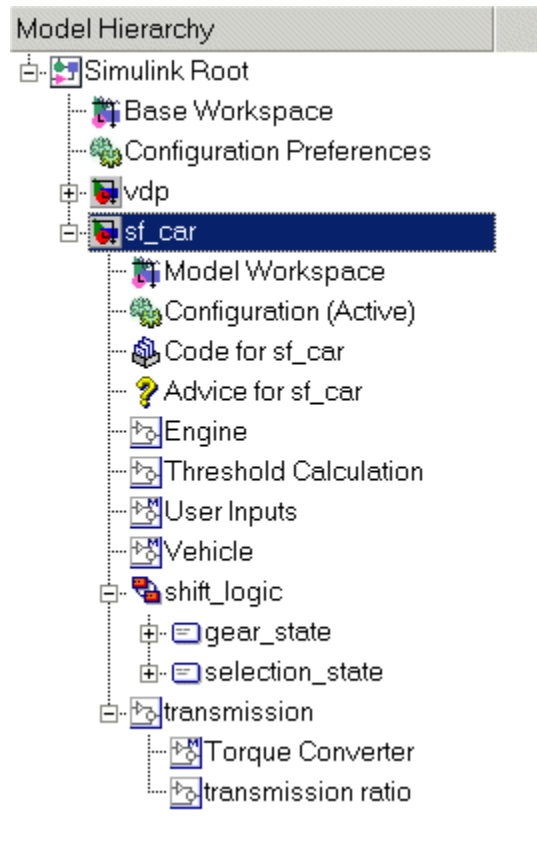
The Model Explorer includes the following components:

- **Model Hierarchy** pane (see “Model Hierarchy Pane” on page 19-3)
- **Contents** pane (see “Contents Pane” on page 19-6)
- **Dialog** pane (see “Dialog Pane” on page 19-11)
- **Main** toolbar (see “Main Toolbar” on page 19-12)
- **Search** bar (see “Search Bar” on page 19-14)

You can use the Model Explorer’s **View** menu to hide the **Dialog** pane and the toolbars, thereby making more room for the other panes.

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy.



Simulink Root

The first node in the view represents the Simulink root. Expanding the root node displays nodes representing the MATLAB workspace (the Simulink base workspace) and each model and library loaded in the current session.

Base Workspace

This node represents the MATLAB workspace. The MATLAB workspace is the base workspace for Simulink models. Variables defined in this workspace are visible to all open Simulink models, i.e., to all models whose nodes appear beneath the **Base Workspace** node in the **Model Hierarchy** pane.

Configuration Preferences

If you check the **Show Configuration Preferences** option on the Model Explorer's **View** menu, the expanded Simulink Root node also displays a Configuration Preferences node. Selecting this node displays the preferred model configuration (see “Setting Up Configuration Sets” on page 20-2) for new models in the adjacent panes. 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 Dialog Box” on page 20-11).

Model Nodes

Expanding a model node displays nodes representing the model's configuration sets (see “Setting Up Configuration Sets” on page 20-2), top-level subsystems, model references, and Stateflow charts. Expanding a node representing a subsystem displays its subsystems, if any. Expanding a node representing a Stateflow chart displays the chart's top-level states. Expanding a node representing a state shows its substates.

Displaying Node Contents

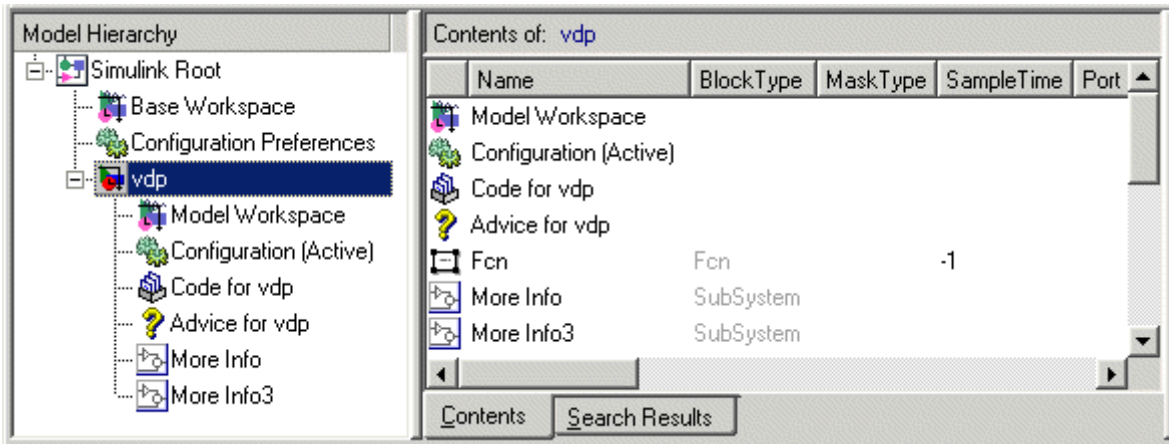
To display the contents of an object displayed in the **Model Hierarchy** pane (e.g., a model or configuration set) in the adjacent **Contents** pane, select the object. To open a graphical object (e.g., a model, subsystem, or chart) in an editor window, right-click the object. A context menu appears. Select **Open** from the context menu. To open an object's properties dialog, select **Properties** from the object's context menu or from the **Edit** menu. See “Setting Up Configuration Sets” on page 20-2 for information on using the **Model Hierarchy** pane to delete, move, and copy configuration sets from one model to another.

Expanding Model References

To expand a node representing a model reference (see Chapter 6, “Referencing a Model”), you must first open the referenced model. To do this, right-click on the node to display its context menu, then select **Open Model** from the menu. The model to which the reference refers to is then opened, a node for it is displayed in the **Model Hierarchy** pane, and all references to the model are made expandable. You cannot edit the contents of a reference node, however. To edit the referenced model, you must expand its node.

Contents Pane

The **Contents** pane displays either of two tabular views selectable by tabs. The **Contents** tab displays the contents of the object selected in the **Model Hierarchy** pane. The **Search Results** tab displays the results of a search operation (see “Search Bar” on page 19-14) .



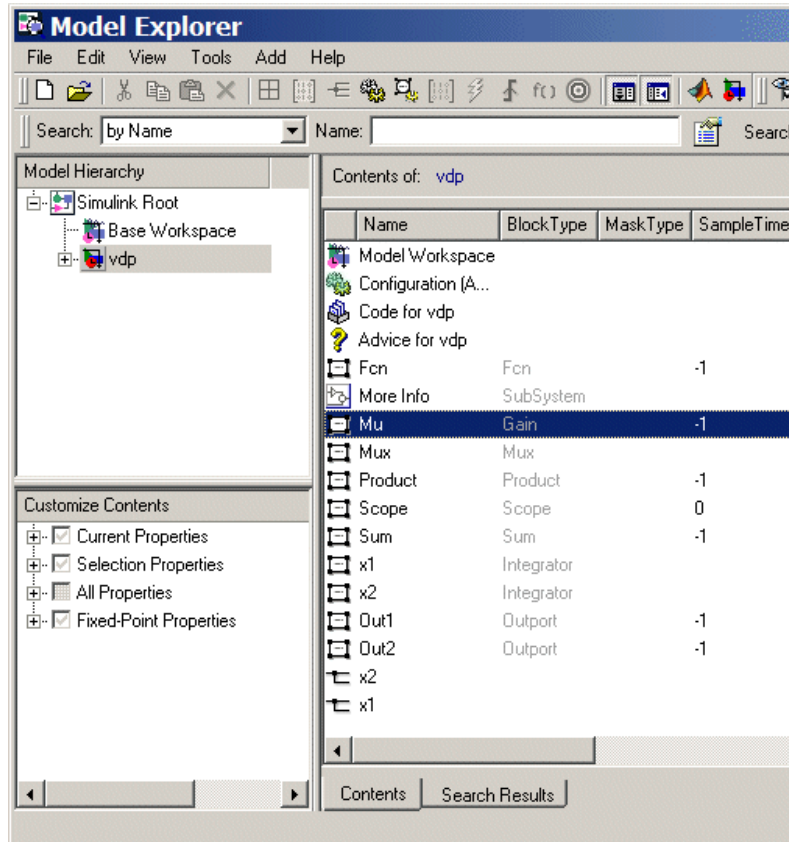
In both views, the table rows correspond to objects (e.g., blocks or states); the table columns, to object properties (e.g., name and type). The table cells display the values of the properties of the objects contained by the object selected in the **Model Hierarchy** pane or found by a search operation.

The objects and properties displayed in the **Contents** pane depend on the type of object (e.g., subsystem, chart, or configuration set) selected in the **Model Hierarchy** pane. For example, if the object selected in the **Model Hierarchy** pane is a model or subsystem, the **Contents** pane by default displays the name and type of the top-level blocks contained by that model or subsystem. If the selected object is a Stateflow chart or state, the **Contents** pane by default shows the name, scope, and other properties of the events and data that make up the chart or state.

Customize Contents Pane

The **Customize Contents** pane allows you to select the properties that the **Contents** pane displays for the object selected in the **Model Hierarchy**

pane. When visible, the pane appears in the lower-left corner of the Model Explorer window.



A splitter divides the **Customize Contents** pane from the **Model Hierarchy** pane above it. Drag the splitter up or down to adjust the relative size of the two panes.

The **Customize Contents** pane contains a tree-structured property list. The list's top-level nodes group object properties into the following categories:

- **Current Properties**

Properties that the **Contents** pane currently displays.

- **Selection Properties**
Properties of the object currently selected in the **Contents** pane.
- **All Properties**
Properties of the contents of all models displayed in the Model Explorer thus far in this session.
- **Fixed Point Properties**
Fixed-point properties of blocks.

By default, the **Contents** pane displays a standard subset of properties for the currently selected model. The **Customize Contents** pane allows you to perform the following customizations:

- To display additional properties of the selected model, expand the **All Properties** node, if necessary, and check the desired properties.
- To delete some but not all properties from the **Contents** pane, expand the **Current Properties** node, if necessary, and uncheck the properties that you do not want to appear in the **Contents** pane.
- To delete all properties from the **Contents** pane (except the selected object's name), uncheck **Current Properties**.
- To display only the properties of the currently selected object, uncheck **Current Properties** to clear the properties display, then check **Selection Properties**.
- To add or remove fixed-point block properties from the **Contents** pane, check or uncheck **Fixed Point Properties**.

Customizing the Contents Pane

The Model Explorer's **View** menu allows you to control the type of objects and properties displayed in the **Contents** pane.

- To display only object names in the **Contents** pane, uncheck the **Show Properties** item on the **View** menu.
- To customize the set of properties displayed in the **Contents** pane, select **Customize Contents** from the **View** menu or click the **Customize Contents** button on the Model Explorer's main toolbar (see "Main Toolbar")

on page 19-12). The **Customize Contents** pane appears. Use the pane to select the properties you want the **Contents** pane to display.

- To specify the types of subsystem or chart contents displayed in the **Contents** pane, select **List View Options** from the **View** menu. A menu of object types appears. Check the types that you want to be displayed (e.g., **Blocks** and **Named Signals/Connections** or **All Simulink Objects** for models and subsystems).

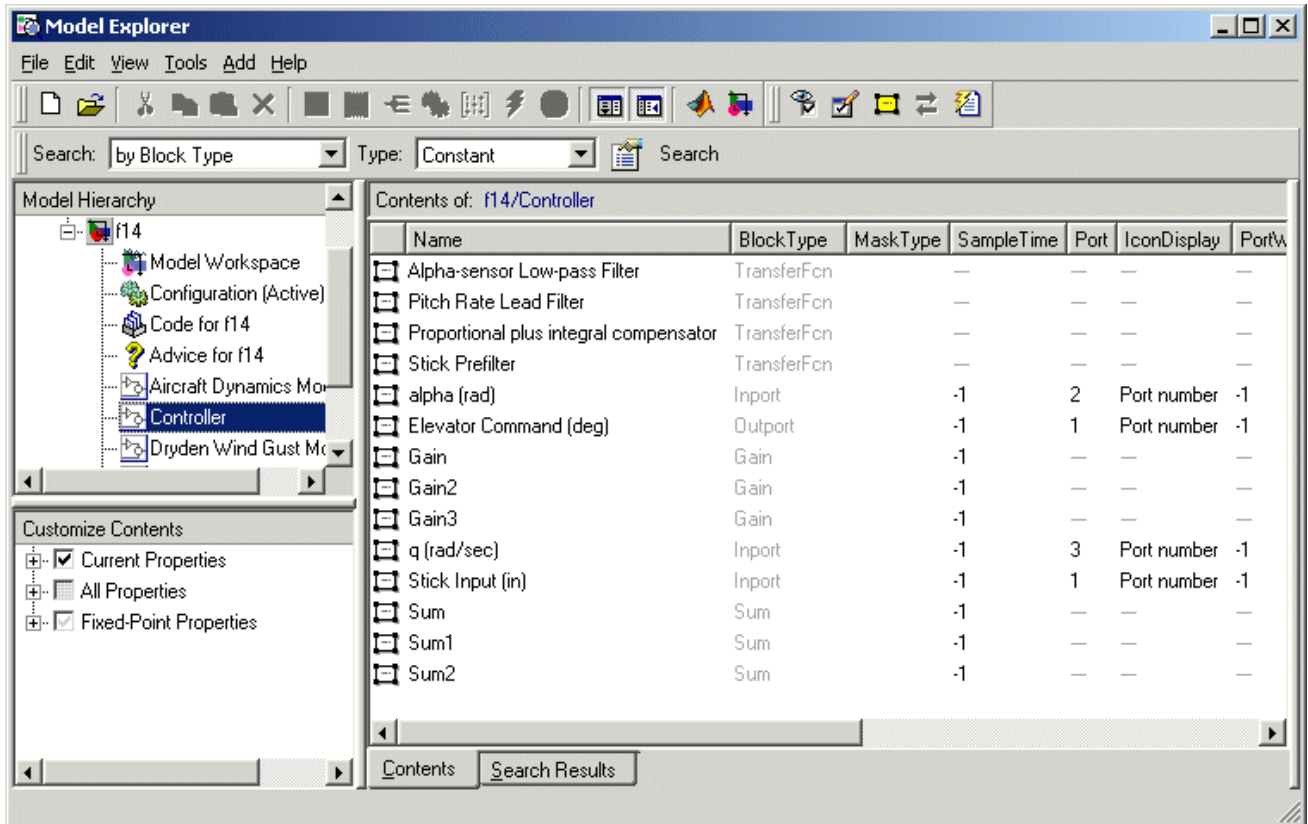
Reordering the Contents Pane

The **Contents** pane by default displays its contents in ascending order by name. To order the contents in ascending order by any other displayed property, click the head of the column that displays the property. To change the order from ascending to descending, or vice versa, click the head of the property column that determines the current order.

Marking Nonexistent Properties

Some of the properties that the **Contents** pane is configured to display may not apply to all the objects currently listed in the **Contents** pane. You can configure the Model Explorer to indicate the inapplicable properties.

To do this, select **Mark Nonexistent Properties** from the Model Explorer's **View** menu. The Model Explorer now displays dashes for the values of properties that do not apply to the objects displayed in the **Contents** pane.



Changing Property Values

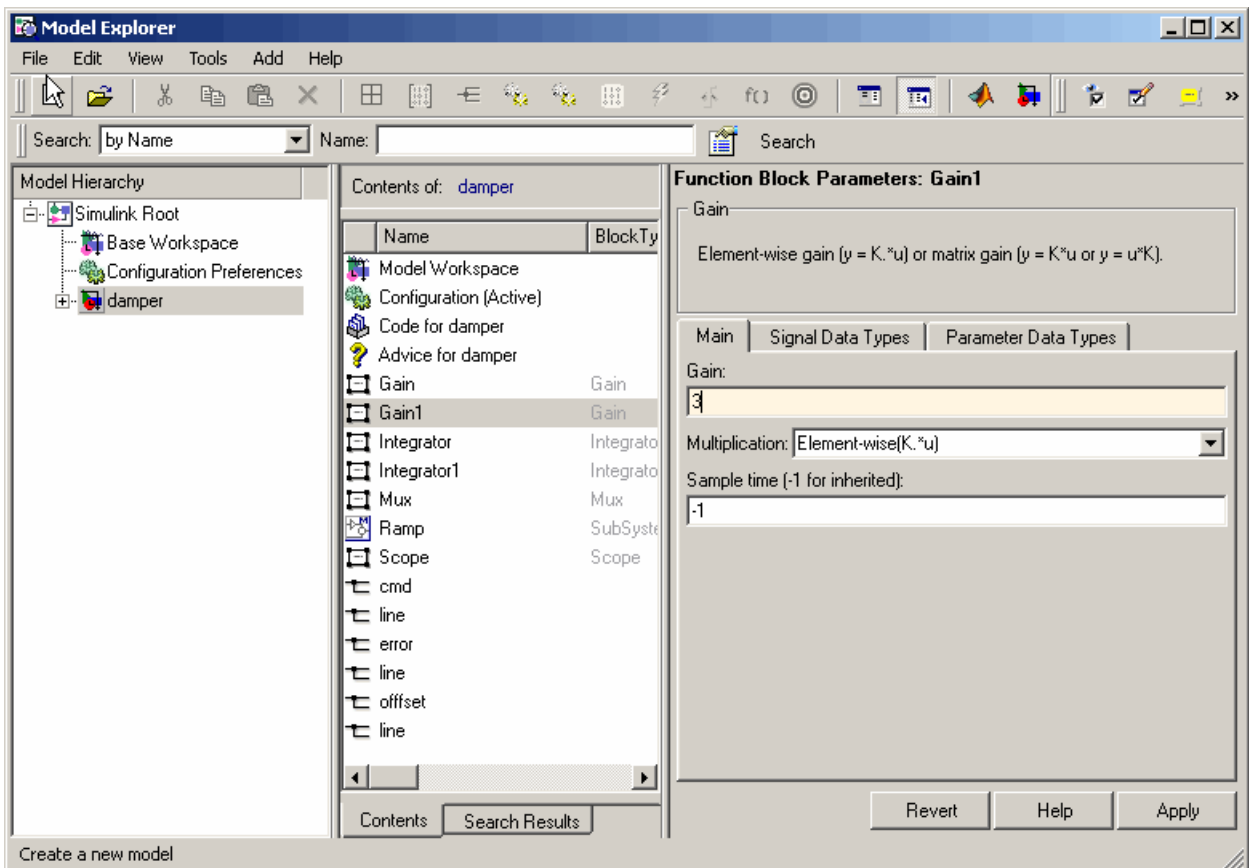
You can change modifiable properties displayed in the **Contents** pane (e.g., a block's name) by editing the displayed value. To edit a displayed value, first select the row that contains it. Then click the value. An edit control replaces the displayed value (e.g., an edit field for text values or a pull-down list for a range of values). Use the edit control to change the value of the selected property.

To assign the same property value to multiple objects displayed in the **Contents** pane, select the objects and then change one of the selected objects to have the new property value. The Model Explorer assigns the new property value to the other selected objects as well.

Dialog Pane

Use the **Dialog** pane to view and change properties of the blocks or signals in your model.

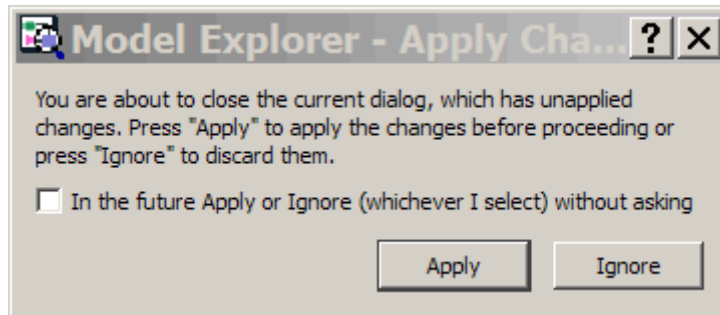
- 1 To show the **Dialog** pane, select **Dialog View** from the **View** menu, located on the Model Explorer's main toolbar.
- 2 In the **Contents** pane, select an object (such as a block or signal). The properties are displayed in the **Dialog** pane.



- 3 Change a property (for example, the gain of a gain block) in the **Dialog** pane.

- 4 Click **Apply** to accept the change, or click **Revert** to return to the original value.

By default, clicking outside a dialog with unapplied changes causes the **Model Explorer – Apply Changes** dialog box to appear:

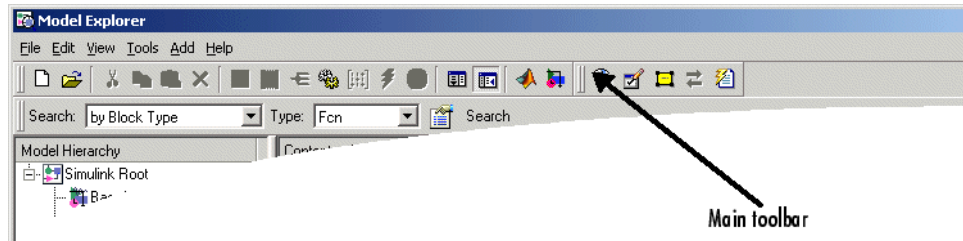


Click **Apply** to accept the changes or **Ignore** to revert to the original settings. To suppress this dialog box in the future, check **In the future Apply or Ignore (whichever I select) without asking** on the dialog box before selecting the **Apply** or **Ignore** button. This causes Simulink to perform the action you select, i.e., either apply or ignore the changes, in the future without displaying the warning dialog box. If, in the future, you want Simulink to revert to its default behavior of warning you about unapplied changes, check the **Prompt if dialog has unapplied changes** item on the Model Explorer **Tools** menu.










Note If the **Prompt if dialog has unapplied changes** item is checked and you uncheck it, Simulink applies unapplied changes without warning you. If you want Simulink to ignore unapplied changes without warning you, use the check box on the warning dialog box as described above.









Main Toolbar

The Model Explorer's main toolbar appears near the top of the Model Explorer window under the Model Explorer's menu.



The toolbar contains buttons that select commonly used Model Explorer commands:

Button	Usage
	Create a new model.
	Open an existing model.
	Cut the objects (e.g., variables) selected in the Contents pane from the object (e.g., a workspace) selected in the Model Hierarchy pane. Save a copy of the object on the system clipboard.
	Copy the objects selected in the Contents pane to the system clipboard.
	Paste objects from the clipboard into the object selected in the Model Explorer's Model Hierarchy pane.
	Delete the objects selected in the Contents pane from the object selected in the Model Hierarchy pane.
	Add a MATLAB variable to the workspace selected in the Model Hierarchy pane.
	Add a Simulink.Parameter object to the workspace selected in the Model Hierarchy pane.
	Add a Simulink.Signal object to the workspace selected in the Model Hierarchy pane.

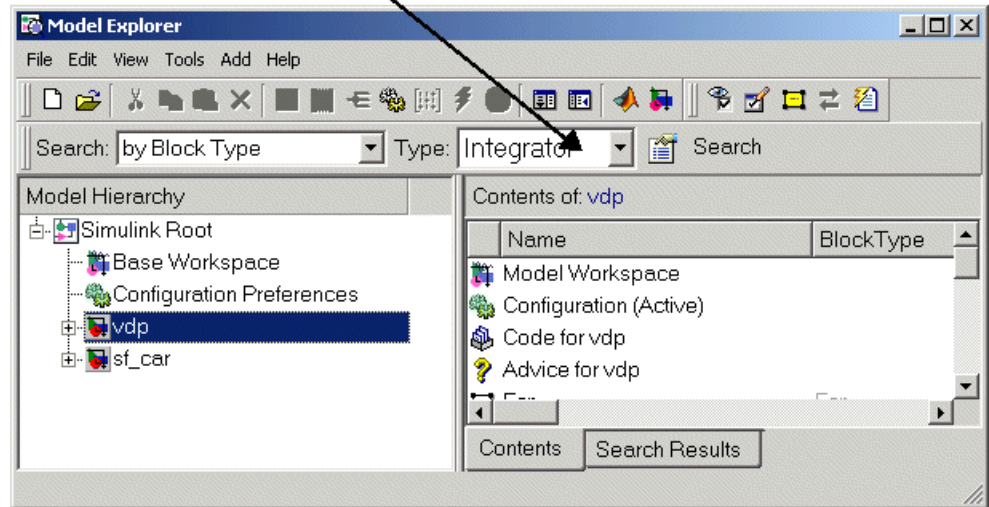
Button	Usage
	Add a configuration set to the model selected in the Model Hierarchy pane.
	Add a Stateflow datum to the machine or chart selected in the Model Hierarchy pane.
	Add a Stateflow event to the machine or chart selected in the Model Hierarchy pane or to the state selected in the Model Explorer.
	Add a code generation target to the model selected in the Model Hierarchy pane.
	Turn the Model Explorer's Dialog pane on or off.
	Customize the Model Explorer's Contents pane.
	Bring the MATLAB desktop to the front.
	Display the Simulink Library Browser.

To show or hide the main toolbar, select **Main Toolbar** from the Model Explorer's **View** menu.

Search Bar

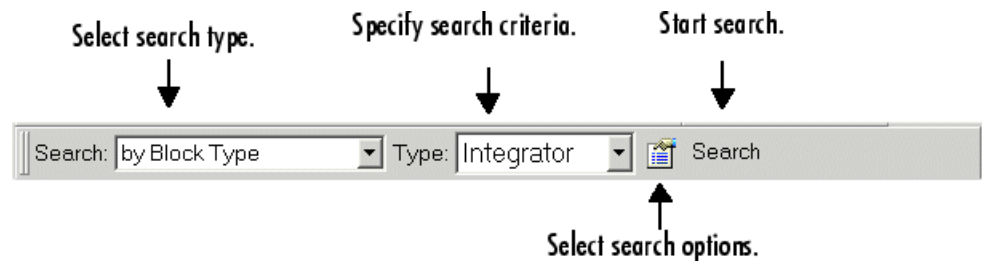
The Model Explorer's search bar allows you to select, configure, and initiate searches of the object selected in the **Model Hierarchy** pane. It appears at the top of the Model Explorer window.

Search bar



To show or hide the search bar, check or uncheck **Search Bar** in the Model Explorer's **View > Toolbars** menu.

The search bar includes the following controls:



Search Type

Specifies the type of search to be performed. Options include:

- by Block Type

Search for blocks of a specified block type. Selecting this search type causes the search bar to display a block type list control that allows you to select the target block type from the types contained by the currently selected model.

- **by Property Name**

Searches for objects that have a specified property. Selecting this search type causes the search bar to display a control that allows you to specify the target property's name by selecting from a list of properties that objects in the search domain can have.

- **by Property Value**

Searches for objects whose property matches a specified value. Selecting this search type causes the search bar to display controls that allow you to specify the name of the property, the value to be matched, and the type of match (equals, less than, greater than, etc.).

- **for Fixed Point**

Searches a model for all blocks that support fixed-point computations.

- **by Name**

Searches a model for all objects that have the specified string in the name of the object.

- **by Stateflow Type**

Searches for Stateflow objects of a specified type.

- **for Library Links**

Searches for library links in the current model.

- **by Class**

Searches for Simulink objects of a specified class.

- **for Model References**

Searches a model for references to other models.

- **by Dialog Prompt**

Searches a model for all objects whose dialogs contain a specified prompt.

- **by String**

Searches a model for all objects in which a specified string occurs.

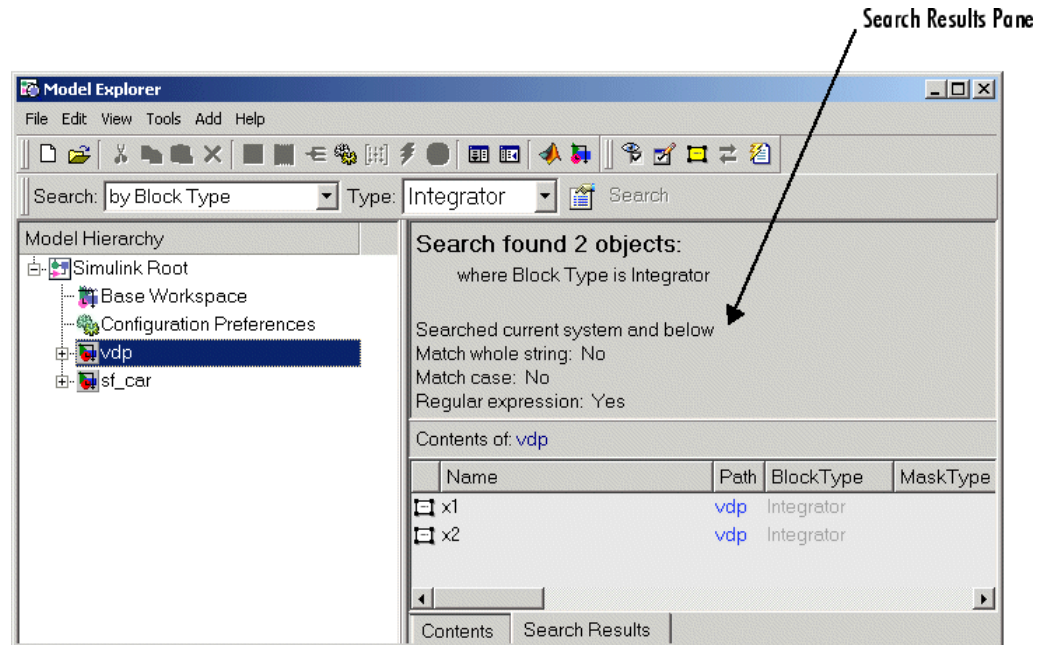
Search Options

Specifies options that apply to the current search. The options include:

- **Search Current System and Below**
Search the current system and the subsystems that it includes directly or indirectly.
- **Look Inside Masked Subsystems**
Search includes masked subsystems.
- **Look Inside Linked Subsystems**
Search includes linked subsystems.
- **Match Whole String**
Do not allow partial string matches, e.g., do not allow sub to match substring.
- **Match Case**
Consider case when matching strings, e.g., Gain does not match gain.
- **Regular Expression**
The Model Explorer considers a string to be matched as a regular expression.
- **Evaluate Property Values During Search**
This option 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 disabled (the default), the Model Explorer compares the unevaluated property value to the search value.
- **Refine Search**
Causes the next search operation to search for objects that meet both the original and new search criteria (see “Refining a Search” on page 19-18).

Search Button

Initiates the search specified by the current settings of the search bar on the object selected in the Model Explorer's **Model Hierarchy** pane. The Model Explorer displays the results of the search in the tabbed **Search Results** pane.



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 one of them to have the new property value.

Refining a Search

To refine the previous search, check the **Refine Search** option on the search bar's **Search Options** menu. 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 the previous search criteria and the new criteria.

Setting the Model Explorer's Font Size

You use Model Explorer to change the font size used in the Simulink dialog boxes and in Model Explorer.

To change the font size used in Model Explorer and in the Simulink Dialog boxes, first open Model Explorer (see “The Model Explorer” on page 19-2).

- Press the **Ctrl+** keys to increase the font size

Alternatively, from the Model Explorer's **View** menu, select **Increase Font Size**

- Press the **Ctrl-** keys to decrease the font size

Alternatively, from the Model Explorer's View menu, select **Decrease Font Size**

Note These changes simultaneously alter the font size used by Model Explorer and in the Simulink dialog boxes. The changes remain in effect across Simulink sessions.

The Finder

In this section...

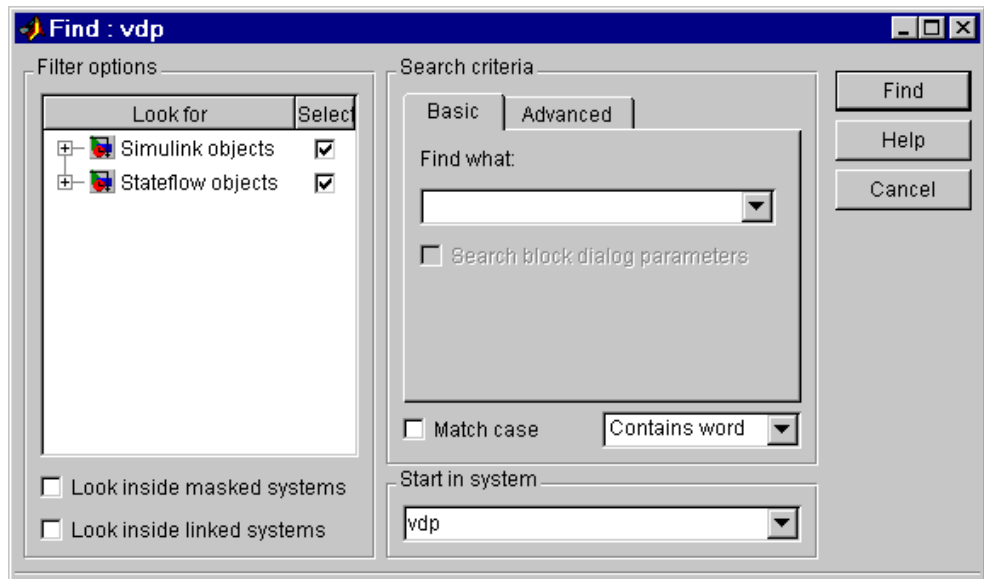
“About the Finder” on page 19-20

“Filter Options” on page 19-22

“Search Criteria” on page 19-23

About the Finder

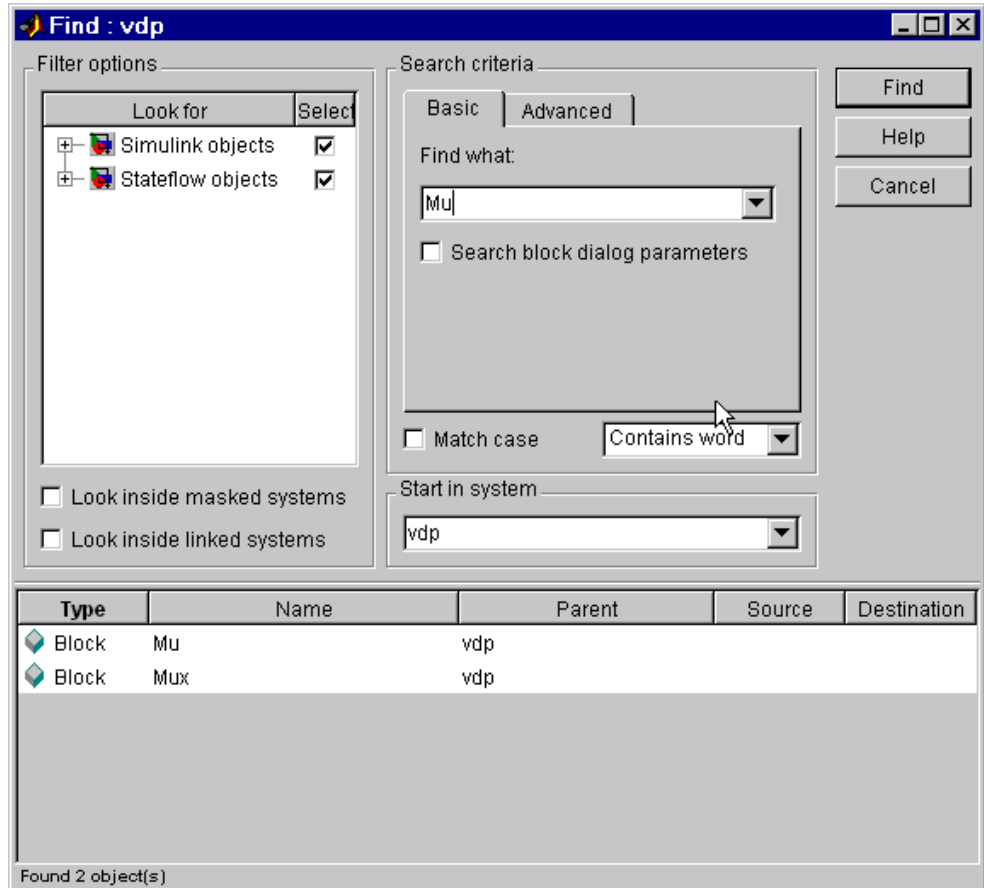
The Finder locates blocks, signals, states, or other objects in a model. To display the Finder, select **Find** from the Simulink Model Editor’s **Edit** menu. The **Find** dialog box appears.



Use the **Filter options** (see “Filter Options” on page 19-22) and **Search criteria** (see “Search Criteria” on page 19-23) panels to specify the characteristics of the object you want to find. Next, if you have more than one system or subsystem open, select the system or subsystem where you want the search to begin from the **Start in system** list. Finally, click the

Find button. Simulink searches the selected system for objects that meet the criteria you have specified.

Any objects that satisfy the criteria appear in the results panel at the bottom of the dialog box.

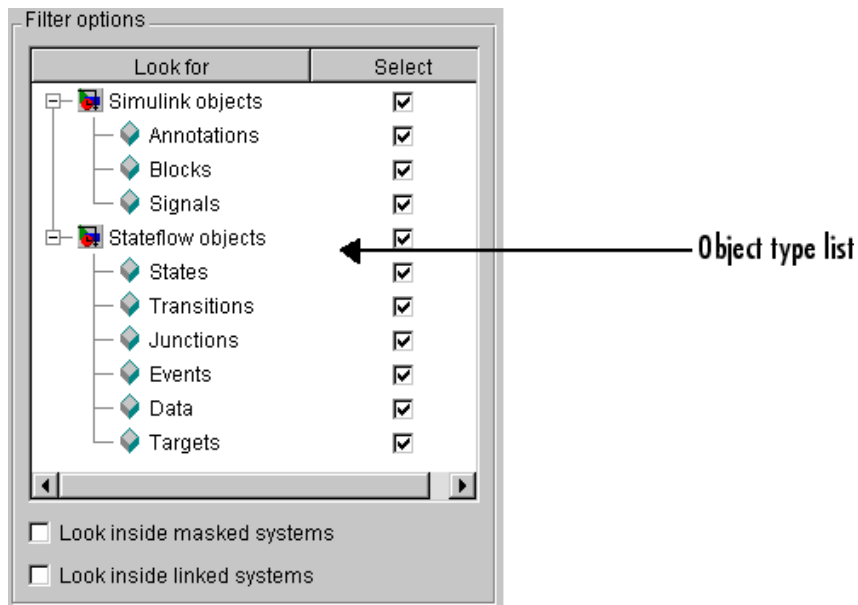


You can display an object by double-clicking its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object. To sort the results list, click any of the buttons at the top of each column. For example, to sort the results by object type, click the **Type** button. Clicking a button once sorts the list in

ascending order, clicking it twice sorts it in descending order. To display an object's parameters or properties, select the object in the list. Then press the right mouse button and select **Parameter** or **Properties** from the resulting context menu.

Filter Options

The **Filter options** panel allows you to specify the kinds of objects to look for and where to search for them.



Object type list

The object type list lists the types of objects that Simulink can find. By clearing a type, you can exclude it from the Finder's search.

Look inside masked subsystem

Selecting this option causes Simulink to look for objects inside masked subsystems.

Look inside linked systems

Selecting this option causes Simulink to look for objects inside subsystems linked to libraries.

Search Criteria

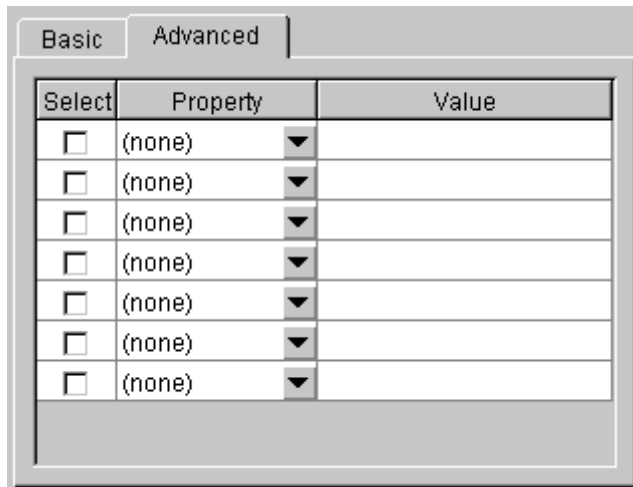
The **Search criteria** panel allows you to specify the criteria that objects must meet to satisfy your search request.

Basic

The **Basic** panel allows you to search for an object whose name and, optionally, dialog parameters match a specified text string. Enter the search text in the panel's **Find what** field. To display previous search text, select the drop-down list button next to the **Find what** field. To reenter text, click it in the drop-down list. Select **Search block dialog parameters** if you want dialog parameters to be included in the search.

Advanced

The **Advanced** panel allows you to specify a set of as many as seven properties that an object must have to satisfy your search request.



Select	Property	Value
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼
<input type="checkbox"/>	(none)	▼

To specify a property, enter its name in one of the cells in the **Property** column of the **Advanced** pane or select the property from the cell's property list. To display the list, select the down arrow button next to the cell. Next enter the value of the property in the **Value** column next to the property name. 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. If you want to exclude the property, clear the check box.

Match case

Select this option if you want Simulink 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 list that specifies other match options that you can select.

- Match whole word

Specifies a match if the property value and the search text are identical except possibly for case.

- Contains word

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.

Character	Meaning
<code>\</code>	Escape character. Causes the next character to have its ordinary meaning. For example, the regular expression <code>\.</code> matches <code>.a</code> and <code>.2</code> and any other two-character string that begins with a period.
<code>*</code>	Matches zero or more instances of the preceding character. For example, <code>ba*</code> matches <code>b</code> , <code>ba</code> , <code>baa</code> , etc.
<code>+</code>	Matches one or more instances of the preceding character. For example, <code>ba+</code> matches <code>ba</code> , <code>baa</code> , etc.
<code>[]</code>	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.

The Model Browser

In this section...
“About the Model Browser” on page 19-26
“Navigating with the Mouse” on page 19-28
“Navigating with the Keyboard” on page 19-28
“Showing Library Links” on page 19-28
“Showing Masked Subsystems” on page 19-28

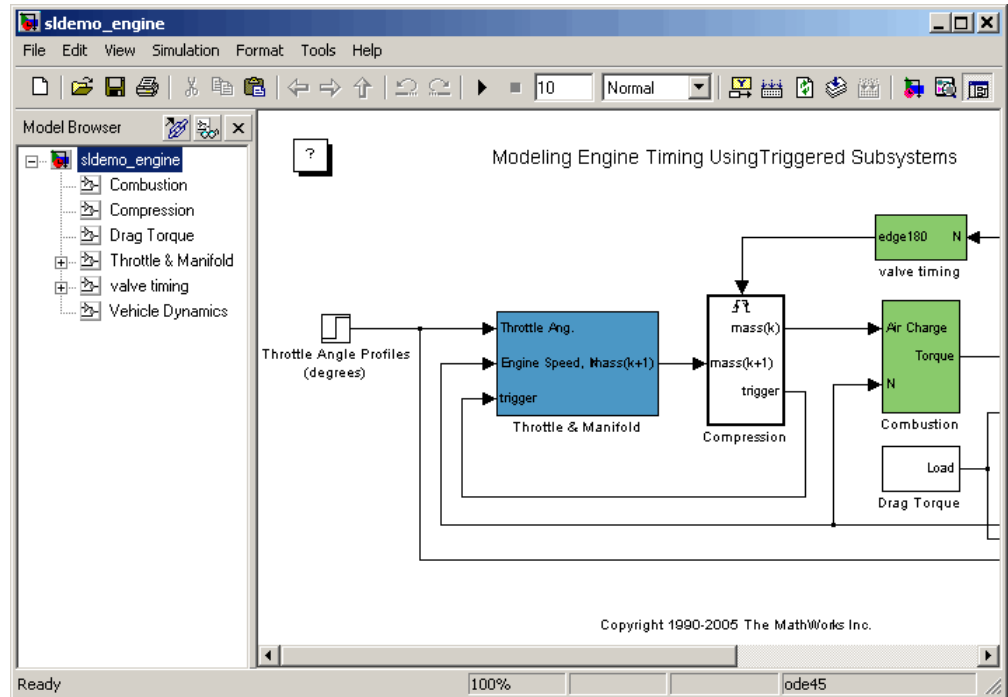
About the Model Browser

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, select **Model Browser Options > Model Browser** from the Simulink **View** menu.



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, select **Preferences** from the Simulink **File** menu.

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 **Show Library Links** from the **Model Browser Options** submenu of the **View** menu.

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 **Look Under Masks** from the **Model Browser Options** submenu of the **View** menu.

Model Dependencies

In this section...

- “What Are Model Dependencies?” on page 19-29
- “Generating Manifests” on page 19-30
- “Command-Line Dependency Analysis” on page 19-35
- “Editing Manifests” on page 19-38
- “Comparing Manifests” on page 19-41
- “Exporting Files in a Manifest” on page 19-42
- “Scope of Dependency Analysis” on page 19-44
- “Best Practices for Dependency Analysis” on page 19-47
- “Using the Model Manifest Report” on page 19-48
- “Using the Model Dependency Viewer” on page 19-52

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 without which the model cannot run. These required files are called *model dependencies*.

The Simulink Manifest Tools allow you to analyze a model to determine its model dependencies. After you identify these dependencies, you can:

- View the files required by your model in a “manifest” file.
- 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 “Using the Model Dependency Viewer” on page 19-52.

Generating Manifests

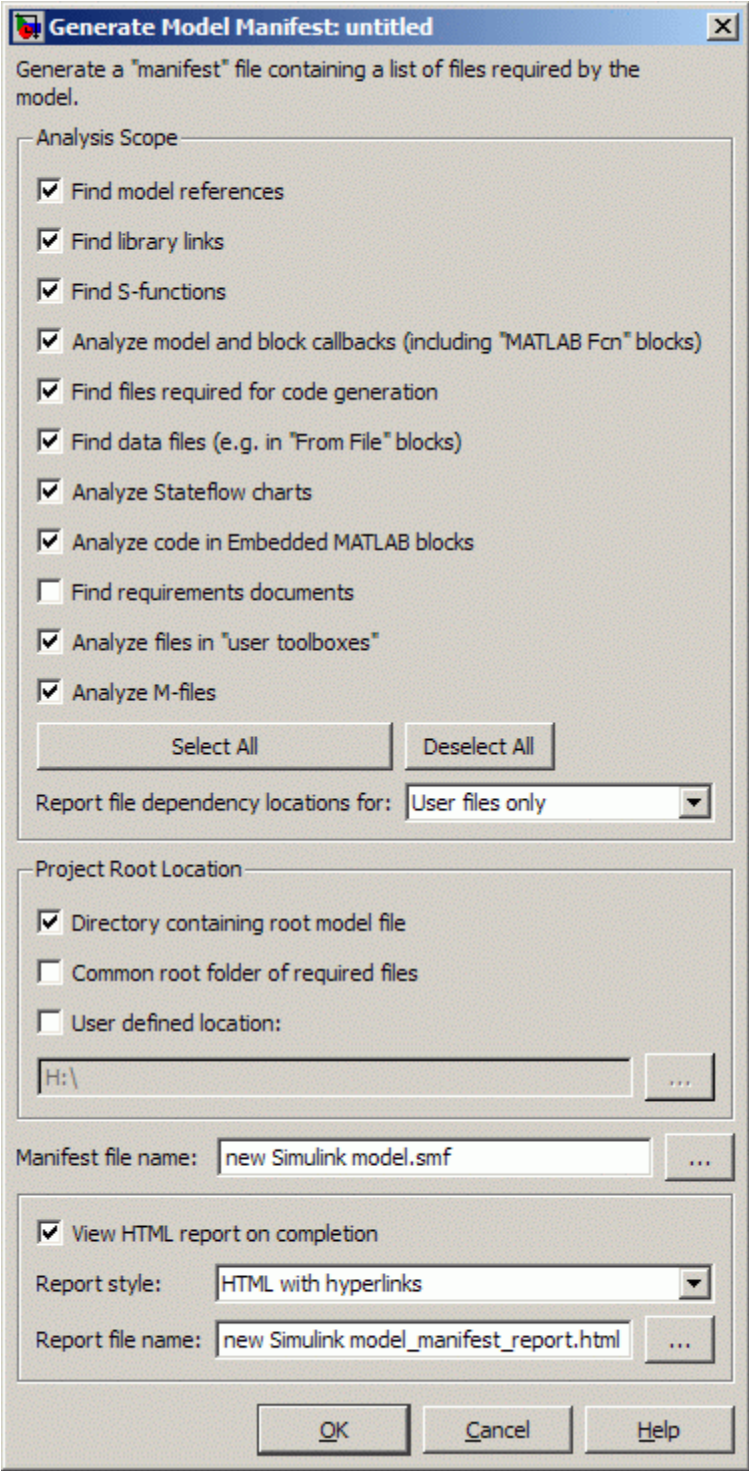
Generating a manifest performs the dependency analysis and saves the list of model dependencies to a manifest file. You must generate the manifest before using any of the other Simulink Manifest Tools.

Note The model dependencies identified in a manifest depend upon the *Analysis Scope* you specify. For example, performing an analysis without selecting **Find Library Links** may not find all the Simulink Blocksets that your model requires, since these are often included in a model as library links. See “Manifest Analysis Scope Options” on page 19-33.

To generate a manifest:

- 1 Select **Tools > Model Dependencies > Generate Manifest**.

The Generate Model Manifest dialog box appears.



- 2** Select the **Analysis scope** check boxes to specify the type of dependencies you want to detect (see “Manifest Analysis Scope Options” on page 19-33).
- 3** 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.
- 4** If desired, change the **Project Root Location**. Select one of the check box options:
 - Directory containing root model file (the default).
 - Common root folder of required files.
 - User-defined location — for this option, enter a path in the edit box, or browse to a location.
- 5** Specify the **Manifest file name** and location in which to save the file.
- 6** To generate a report when you generate the manifest, select **View HTML report on completion**, then specify the Report style (Plain HTML or HTML with Hyperlinks) and Report file name.
- 7** 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 “Using the Model Manifest Report” on page 19-48 for an example.

The manifest is an XML file with the extension `.smf` located (by default) in the same directory 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 model references	Searches for Model blocks in the model, and identifies any referenced models as dependencies.
Find library links	Searches for links to library blocks in the model, and identifies any library links as dependencies.
Find S-functions	Searches for S-Function blocks in the model, and identifies S-function files (M-code and C) as dependencies. See the source code item in “Special Cases” on page 19-46.
Analyze model and block callbacks (including “MATLAB Fcn” blocks)	Searches for file dependencies introduced by the M-code in MATLAB Fcn blocks, block callbacks, and model callbacks. For more detail on how callbacks are analyzed, see “M-Code Analysis” on page 19-45.

Check Box Option	Description
Find files required for code generation	<p>Searches for file dependencies introduced by Real-Time Workshop custom code, and Real-Time Workshop 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.</p> <p>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 Target Function Library definition files (.m or .mat). See also “Required Toolboxes” on page 19-49, and the source code item in “Special Cases” on page 19-46.</p>
Find data files (e.g. in “From File” blocks)	<p>Searches for explicitly referenced data files, such as those in From File blocks, and identifies those files as dependencies. See “Special Cases” on page 19-46.</p>
Analyze Stateflow charts	<p>Searches for file dependencies introduced through the use of syntax such as <code>m1.mymean(myvariable)</code> in models that use Stateflow.</p>
Analyze code in Embedded MATLAB blocks	<p>Searches for Embedded MATLAB Function blocks in the model, and identifies any file dependencies (outside toolboxes) introduced in the M-code. Toolbox dependencies introduced by an Embedded MATLAB Function block are not detected.</p>

Check Box Option	Description
Find requirements documents	Searches for requirements documents linked using the Requirements Management Interface. Note that requirements links created with Telelogic® DOORS® software are not included in manifests. For more information, see “Managing Model Requirements” in the Simulink Verification and Validation documentation. This option is disabled if you do not have a Simulink Verification and Validation license, and Simulink ignores any requirements links in your model.
Analyze files in “user toolboxes”	Searches for file dependencies introduced by files in user-defined toolboxes. See “Special Cases” on page 19-46.
Analyze M-files	Searches for file dependencies introduced by M-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 “M-Code Analysis” on page 19-45.

See also “Scope of Dependency Analysis” on page 19-44 for more information.

Command-Line Dependency Analysis

- “Check File Dependencies” on page 19-35
- “Check Toolbox Dependencies” on page 19-36

Check File Dependencies

To programmatically check for file dependencies, use the method `dependencies.fileDependencyAnalysis` as follows.

```
[files, missing, depfile, manifestfile] =
dependencies.fileDependencyAnalysis('modelName')
```

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* — the full-path of the file containing information about any user-defined files associated with the model *modelName*.
- *manifestfile* — (optional) 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 *manifestfile*. *manifestfile* can be a full-path or just a file name (in which case the file is created in the current directory).

If you try this analysis on a demo model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the demo models.

Check Toolbox Dependencies

To check which toolboxes are required, enter:

```
[names,dirs] =  
dependencies.toolboxDependencyAnalysis(files_in)
```

files_in must be a cell array of strings containing *.m* or *.mdl* files on the MATLAB path. Simulink model names (without *.mdl* 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 directories.

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.mdl'
missing =
    []
depfile =
    []
manifestfile =
    []
```

- 2 Call `toolboxDependencyAnalysis` on the files output of step 1.

For example:

```
tbxes = dependencies.toolboxDependencyAnalysis(files)

tbxes =
[1x24 char]    'MATLAB'    'Real-Time Workshop'    'Simulink'
```

To view long product names examine the `tbxes` cell array as follows:

```
tbxes{:}

ans =
Image Processing Toolbox
```

```
ans =  
MATLAB  
  
ans =  
Real-Time Workshop  
  
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 Real-Time Workshop is always reported as required. See “Required Toolboxes” on page 19-49 for more examples of how your installed products and analysis scope settings can affect reported toolbox requirements.

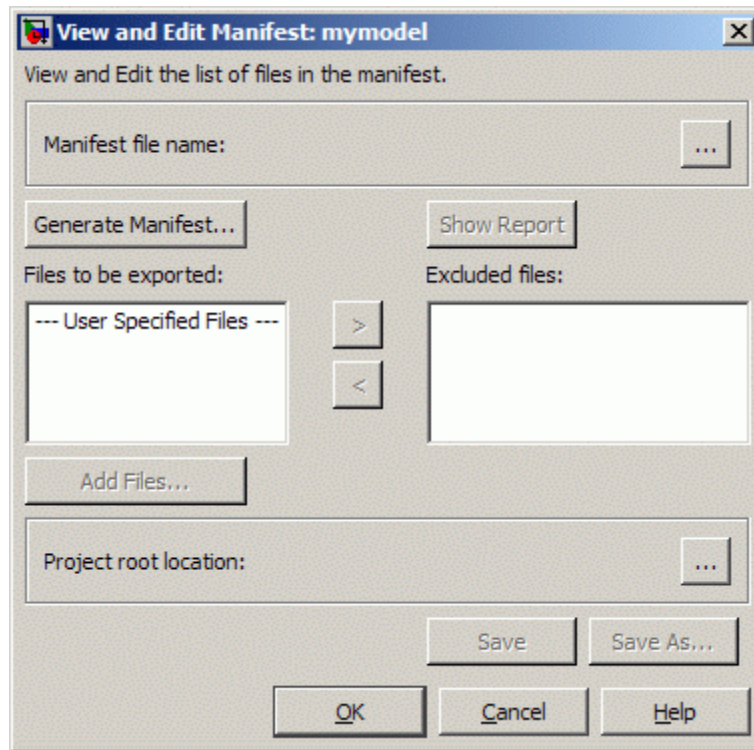
Editing Manifests


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 **Tools > Model Dependencies > View/Edit Manifest Contents**.

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 “Generating Manifests” on page 19-30).

- 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 added manually, as well as those you manually excluded. Simulink uses the .smd file to remember your changes the next time you generate a manifest. The user dependencies (.smd) file has the same name and directory 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 “Using the Model Manifest Report” on page 19-48 for an example.

8 When you are finished editing the manifest, click **OK**.

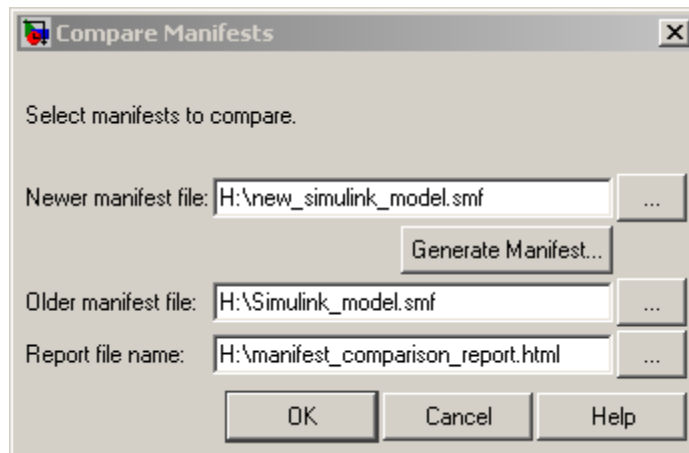
Comparing Manifests

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.

To compare manifest files:

1 Select **Tools > Model Dependencies > Compare Manifests**.

The Compare Manifests dialog box appears.



2 Select the newer manifest file.

Note You can click **Generate Manifest** to create a new manifest. See “Generating Manifests” on page 19-30 for more information. After you generate the manifest, you return to the Compare Manifests dialog box.

3 Select the older manifest file.

4 Specify a report file name and location.

Note The default report file is `manifest_comparison_report.html` in the current working directory.

5 Click **OK**.

The two manifests are compared and displays the Model Manifest Differences Report. The report provides details about each manifest file, and lists the differences between the files.

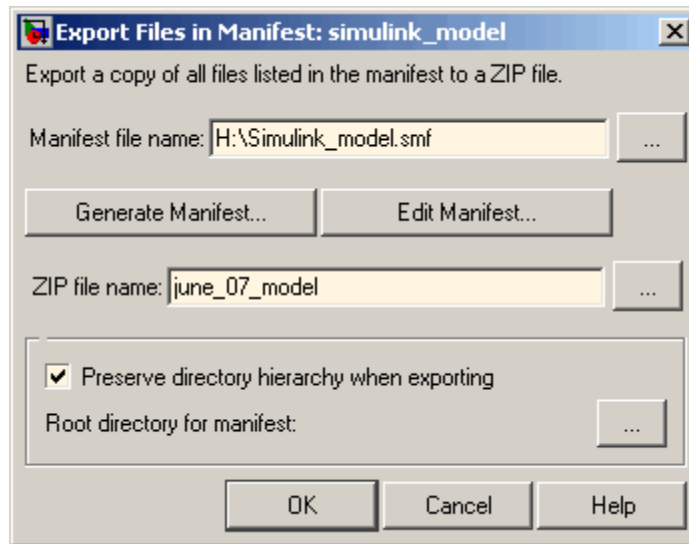
Exporting Files in a Manifest


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 **Tools > Model Dependencies > Export Files in Manifest**.

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 “Generating Manifests” on page 19-30).

- 2** If you want to view or edit the manifest before exporting it, select **Edit Manifest** to view or change the list of required files. See “Editing Manifests” on page 19-38. When you close the View and Edit Manifest dialog box, you return to the Export Files in Manifest dialog box.
- 3** Select **Preserve directory hierarchy** if you want to keep directory structure for your exported model and files. Then, select the root directory to use for this structure (usually the same as the **Project Root Directory** on the Generate Manifest dialog box).

Note You must select **Preserve directory hierarchy** if you are exporting a model that uses an M-file inside a MATLAB class (to maintain the directory structure of the class), or if the model refers to files in other directories (to ensure the exported files maintain the same relative paths).

4 Enter the zip file name to which you want to export the model.

5 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 “Updating a Block Diagram” on page 1-27).

You can specify the type of dependencies you want to detect when you generate the manifest. See “Manifest Analysis Scope Options” on page 19-33.

For more information on what the tool analyzes, refer to the following sections:

- “Analysis Limitations” on page 19-44
- “M-Code Analysis” on page 19-45
- “Special Cases” on page 19-46

Analysis Limitations

The analysis might not find all files required by your model (for examples, see “M-Code Analysis” on page 19-45).

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 Simulink Fixed Point) cannot be detected.

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 “Editing Manifests” on page 19-38).

M-Code Analysis

When the Simulink Manifest Tools encounter M-code, for example in a model or block callback, or in an M-file S-function, they attempt to identify the files it references. If those files contain M-code, *and* the analysis scope option **Analyze M-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`, `csvread`, `wk1read`, `textread` 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-file and P-file exist, then the manifest reports both, and, if the **Analyze M-files** option is selected, the M-file is analyzed.

Special Cases

The following list contains additional information about specific cases:

- If your model uses classes created using the Data Class Designer and references a class called `MyPackage.MyClass`, all files inside the directory `@MyPackage` and its subdirectories are added to the manifest.
- 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 directory `X`, any file inside a sub-directory of `X` is considered part of your toolbox.
 - If you have a `Contents.m` file in directory `X/X`, any file inside all sub-directories of the “outer” directory `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 create a UI using GUIDE and add this to a model callback, then the dependency analysis detects the M-file 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 Real-Time Workshop 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 “Editing Manifests” on page 19-38).
- 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
Signal Processing Blockset™	From Wave File block (Microsoft Windows operating system only) From Multimedia File block (Windows only)
Video and Image Processing Blockset	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 M-Code.

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 M-code analysis (see "M-Code Analysis" on page 19-45).

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 directories, do it by relative path, for example:

```
load mydata\mydatafile.mat
```

Select **Preserve directory hierarchy** when exporting, so that the exported files are in the same locations relative to each other. Also, choose a root directory so that all the files listed in the manifest are inside it. Otherwise, any files outside the root are copied into a new directory called **external** underneath the root, and relative paths to those files become invalid.

If you are exporting a model that uses an M-file inside a MATLAB class (in a directory called @myclass, for example), you must select the **Preserve directory hierarchy** check box when exporting, to maintain the directory 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 directories other than MathWorks toolboxes.

Using the Model Manifest Report

- “Required Toolboxes” on page 19-49
- “Example Model Manifest Report” on page 19-50

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
- **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 19-49.

- **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 “Generating Manifests” on page 19-30. 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.
- **Directories referenced by this model.**
- **Dependency analysis settings** — records the details of the analysis scope options.
- **Actions** — Provides links to conveniently regenerate, edit or compare the manifest, and export the files in the manifest to a zip file.

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 Simulink Fixed Point) cannot be detected.

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:
 - Real-Time Workshop software is always reported as required.
 - If you also have an .ert system target file selected then Real-Time Workshop Embedded Coder software is always reported as required.
 - If your model uses Stateflow software, then Stateflow® Coder™ is always reported as required.

- If you clear the **Find library links** option, then the analysis cannot find a dependence on, for example, *someBlockSet.mdl*, and so no dependence is reported upon the block set.
- If you clear the **Analyze M-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 is a portion of an sample report.

The screenshot shows a web browser window titled "Model Manifest Report: mpowertrain". The address bar shows the file path: file:///C:/Work/manifestanalysis/mpowertrain_manifest_report.html. The main content area has the following sections:

- Model Manifest Report: mpowertrain**
- Analysis performed: 19-Jun-2009 16:55:15
- Model Reference and Library Link hierarchy**
 - [mpowertrain](#)
 - [menginemodel](#)
 - [mtransmission](#)
 - [mtransmission_ratio](#)
 - [mpowertrainlib](#)
 - [mpowertrainlib](#)
- Files used by this model**
- Root directory for this manifest: C:\Work\manifestanalysis
- Click on a column header to sort the table

File Name	Size	Last Modified Date	Will be Exported
\$projectroot/LowPassFilter.c (open)	17 bytes	2008-11-27 16:45:10	true
\$projectroot/LowPassFilter.h (open)	20 bytes	2008-11-27 16:45:10	true
\$projectroot/menginemodel.mdl (open)	19260 bytes	2008-11-27 16:45:10	true
\$projectroot/mpowertrainlib.mdl (open)	34759 bytes	2008-11-27 16:45:10	true
\$projectroot/msfunctiondemo.m (open)	11549 bytes	2008-11-27 16:45:10	true
\$projectroot/mtransmission_ratio.mdl (open)	20390 bytes	2008-11-27 16:45:10	true
\$projectroot/powertrain_data.mat	1976 bytes	2008-11-27 16:45:10	true
\$projectroot/Utils.lib (open)	4 bytes	2008-11-27 16:45:10	true
\$projectroot/mtransmission.mdl (open)	35057 bytes	2008-11-27 16:47:26	true
\$projectroot/mpowertrain.mdl (open)	56931 bytes	2008-11-27 16:47:28	true

- Toolboxes required by this model**
 - MATLAB (7.9)
 - Real-Time Workshop (7.4)
 - Simulink (7.4)
 - Stateflow (7.4)
 - Stateflow Coder (7.4)

Model Manifest Report: mpowertrain

File Edit View Go Debug Desktop Window Help

Location: file:///C:/Work/manifestanalysis/mpowertrain_manifest_report.html

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
LibraryLink	mpowertrain/Library Shift Logic (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
ModelReference	mpowertrain/engine model !! (show)	\$projectroot/menginemodel.mdl (open)	(not in a toolbox)
ModelReference	mpowertrain/transmission (show)	\$projectroot/mtransmission.mdl (open)	(not in a toolbox)
ModelCallback,PreLoadFcn	mpowertrain (show)	\$projectroot/powertrain_data.mat	(not in a toolbox)
LibraryLink	mtransmission/Grouped Unit Delay (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
LibraryLink	mtransmission/Torque Converter/Torque Conversion (show)	\$projectroot/mpowertrainlib.mdl (open)	(not in a toolbox)
ModelReference	mtransmission/transmission ratio (show)	\$projectroot/mtransmission_ratio.mdl (open)	(not in a toolbox)
MSFunction	mtransmission_ratio/Level-2 M-file S-Function (show)	\$projectroot/msfunctiondemo.m (open)	(not in a toolbox)
RealTimeWorkshop,Configuration,CustomCode	mtransmission (show)	\$projectroot/LowPassFilter.c (open)	(not in a toolbox)
RealTimeWorkshop,Configuration,CustomCode	mtransmission (show)	\$projectroot/LowPassFilter.h (open)	(not in a toolbox)
RealTimeWorkshop,Configuration,CustomCode	mtransmission (show)	\$projectroot/Utils.lib (open)	(not in a toolbox)

Directories referenced by this model

(This model does not refer to any directories)

Dependency analysis settings:

- Find model references: **true**
- Find library links: **true**
- 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 M-files: **true**
- Reporting of file dependence locations: **user files only**

Actions

- [Re-generate](#) this manifest
- [Edit](#) this manifest
- [Compare](#) this manifest with another one
- [Export](#) the files in this manifest to a ZIP file

Using the Model Dependency Viewer

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.

See the following topics for information on using the viewer:

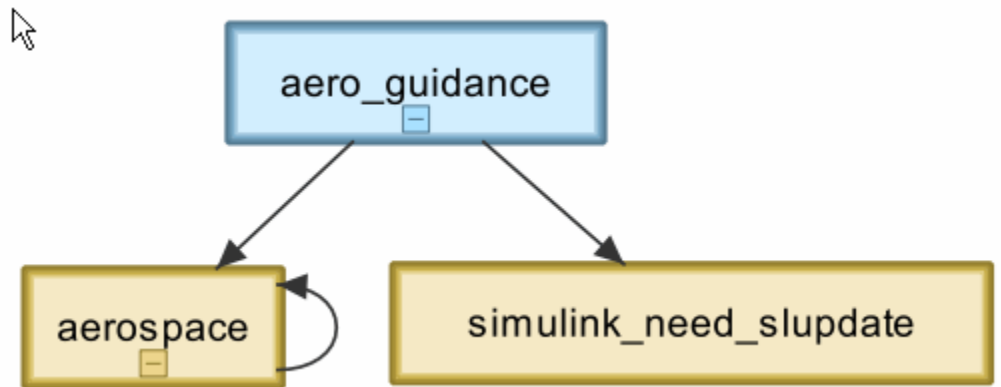
- “About Model Dependency Views” on page 19-53
- “Opening the Model Dependency Viewer” on page 19-60
- “Manipulating a Dependency View” on page 19-61
- “Browsing Dependencies” on page 19-66
- “Saving a Dependency View” on page 19-66
- “Printing a Dependency View” on page 19-67

About Model Dependency Views

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 19-53
- “Referenced Model Instances View” on page 19-55
- “Processor-in-the-Loop Mode Indicator” on page 19-57
- “Error and Warning Icons” on page 19-58

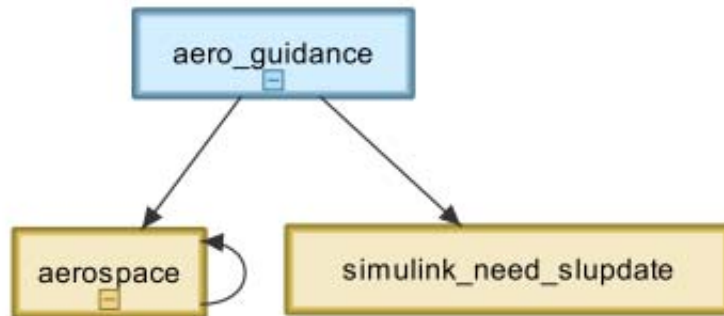
File Dependency View. A *file dependency* view shows the model and library files referenced by a top model. A referenced model or library file appears only once in the view even if it is referenced more than once in the model hierarchy displayed in the view. A file dependency view consists of a set of 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`.



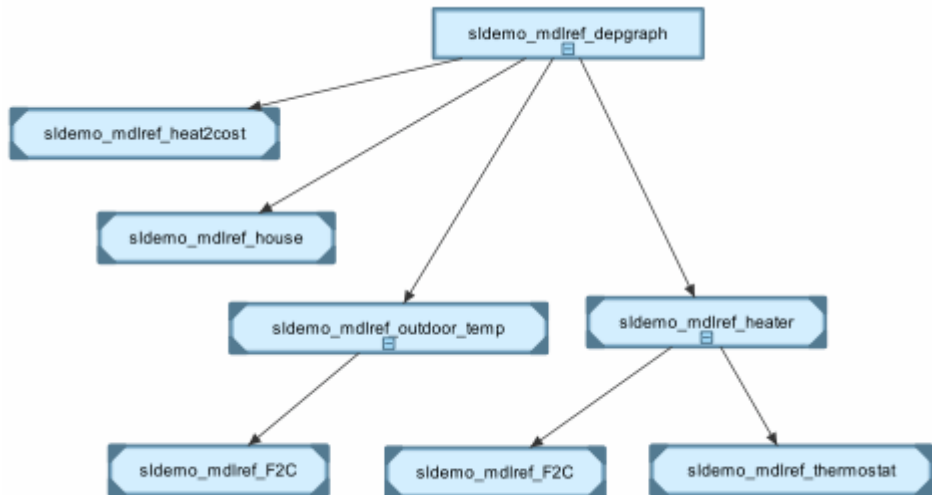
An arrow that points to the library from which it emerges indicates that the library references itself, i.e., blocks in the library reference other blocks in that same library. For example, the preceding view indicates that the aerospace library references itself.

A file dependency view optionally includes a legend that identifies the model in the view and the date and time the view was created.

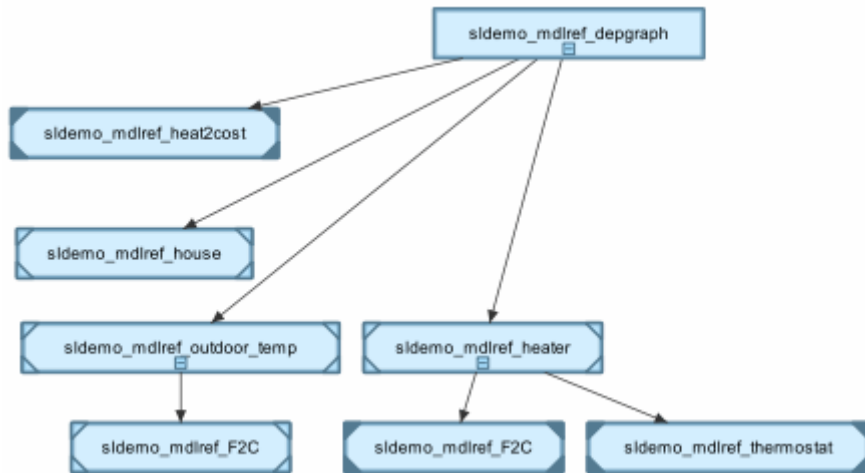
Dependency viewer: `aero_guidance`
 06-Jul-2007 16:17:10



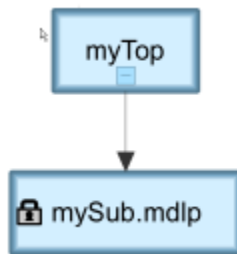
Referenced Model Instances View. A *referenced model instances* view shows every reference to a model in a model reference hierarchy (see Chapter 6, “Referencing a Model”) rooted at the top model targeted by the view. If a model hierarchy references the same model more than once, the referenced model 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`.



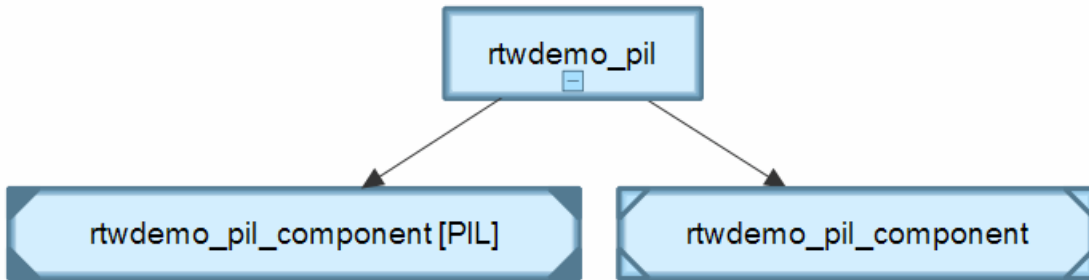
In an instance view, boxes represent a top model and model references. Boxes representing accelerated-mode instances (see “Referenced Model Simulation Modes” on page 6-12) have filled triangles in their corners; boxes representing normal-mode instances, have unfilled triangles in their corners. For example, 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 .mdlP extension. Protected referenced model boxes have no expand(+)/collapse(-) button.



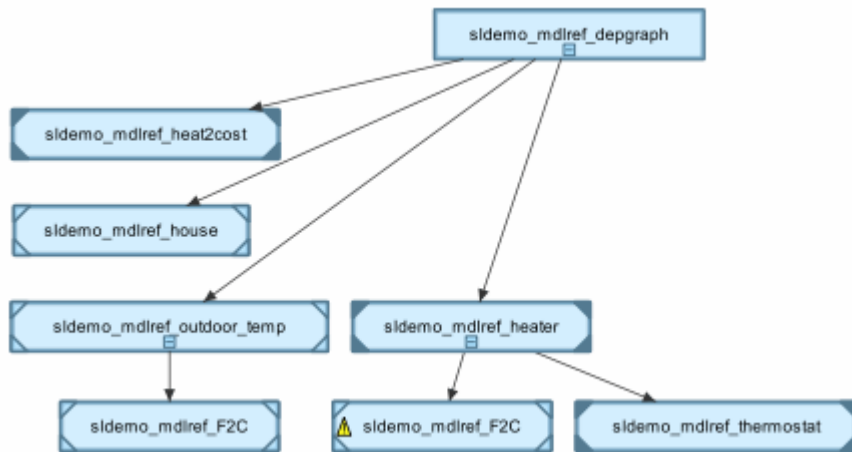
Processor-in-the-Loop Mode Indicator. An instance view appends PIL to the names of models that run in Processor-in-the-Loop mode (see “Specifying the Simulation Mode” on page 6-13). For example, the following dependency instance view indicates that the model named `rtwdemo_pil_component` runs in processor-in-the-loop mode.



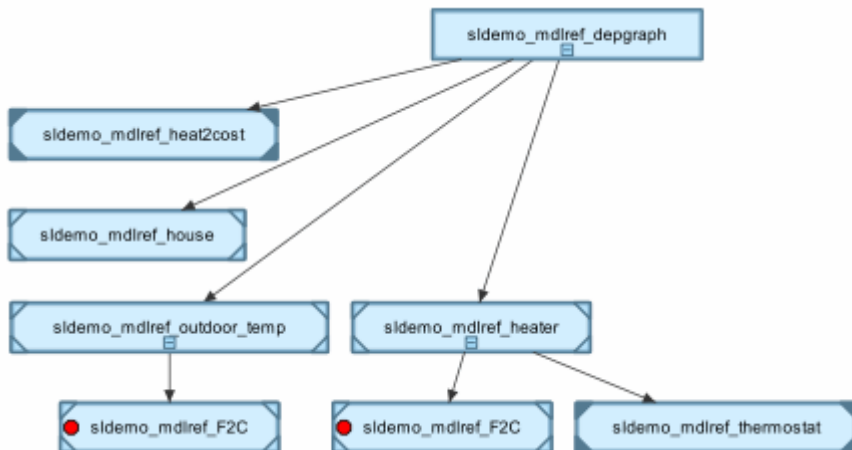
Error and Warning Icons. An instance view displays warning or error icons on instance boxes to indicate invalid normal-mode references (see “Mixing Simulation Modes” on page 6-14).

Icon	Indicates
	A reference 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.
	One of multiple normal-mode references to the same model in a model reference hierarchy. Such a reference is invalid because it violates the rule that only one normal-mode reference to a model can occur in a given model reference hierarchy.

The following instance view, for example, indicates that the accelerated-mode configuration of the reference to `sldemo_md1ref_heater` overrides the normal-mode configurations of its reference to `sldemo_md1ref_F2C`.



Changing the simulation mode of the reference to `sidemo_md1ref_heater` to normal creates two normal-mode references in the model reference hierarchy to the same model, i.e., `sidemo_md1ref_F2C`, which is invalid as indicated in the following instances view.



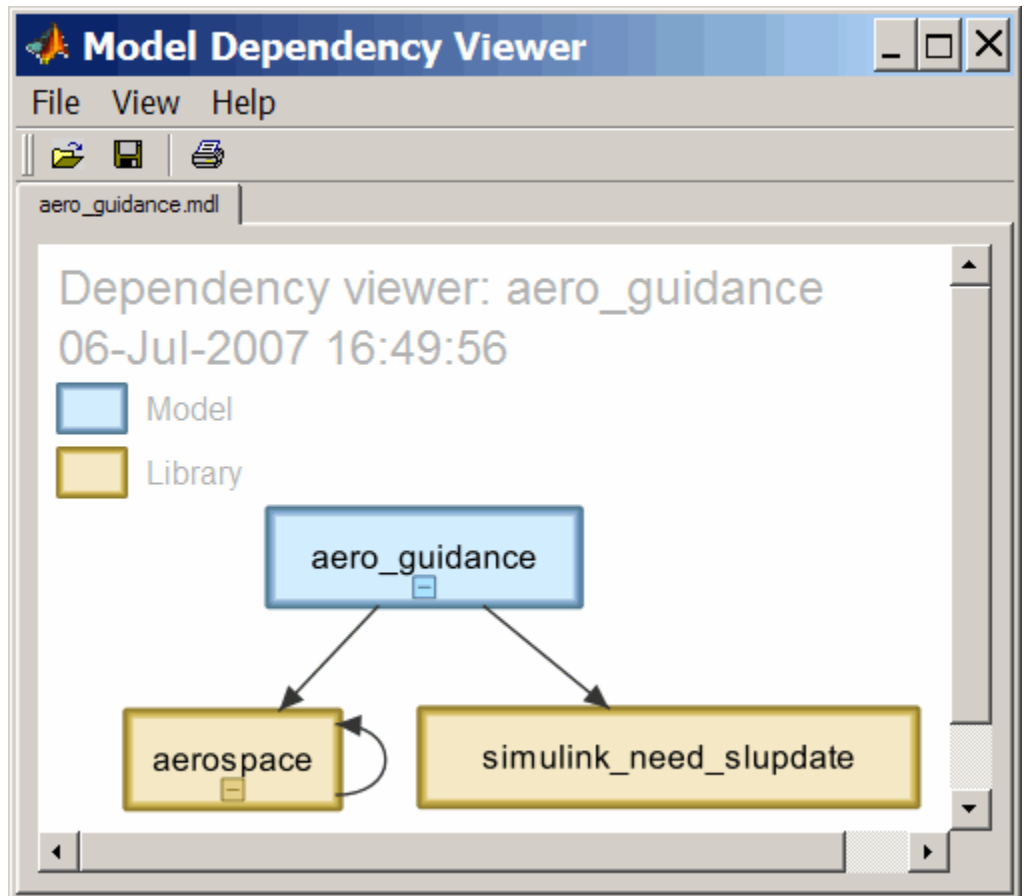
Opening the Model Dependency Viewer

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 **Tools > Model Dependencies > Model Dependency Viewer**, then select the type of view you want to see:
 - **.mdl File Dependencies Including Libraries**
 - **.mdl File Dependencies Excluding Libraries**
 - **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 19-62
- “Excluding Block Libraries from a File Dependency View” on page 19-62
- “Including Simulink Blocksets in a File Dependency View” on page 19-62
- “Changing View Orientation” on page 19-62

- “Expanding or Collapsing Views” on page 19-63
- “Zooming a Dependency View” on page 19-63
- “Moving a Dependency View” on page 19-64
- “Rearranging a Dependency View” on page 19-64
- “Displaying and Hiding a Dependency View’s Legend” on page 19-64
- “Displaying Full Paths of Referenced Model Instances” on page 19-65
- “Refreshing a Dependency View” on page 19-66

Changing Dependency View Type. You can change the type of dependency view displayed in the viewer.

To change the type of dependency view:

- Select **View > Dependency Type > .mdl File Dependencies** (see “File Dependency View” on page 19-53)
- or
- Select **View > Dependency Type > Referenced Model Instances** (see “Referenced Model Instances View” on page 19-55).

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 block libraries supplied by The MathWorks when **View > Include Libraries** is selected.

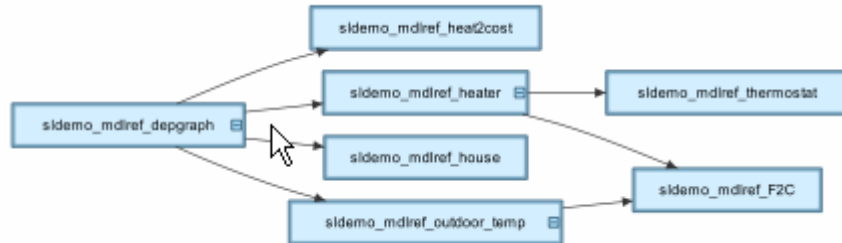
To include libraries supplied by The MathWorks:

- 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:

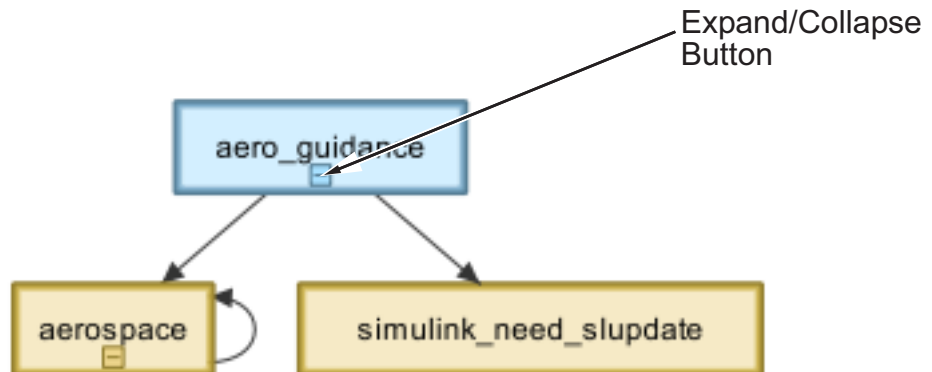
- Select **View > Orientation > Horizontal**.



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:

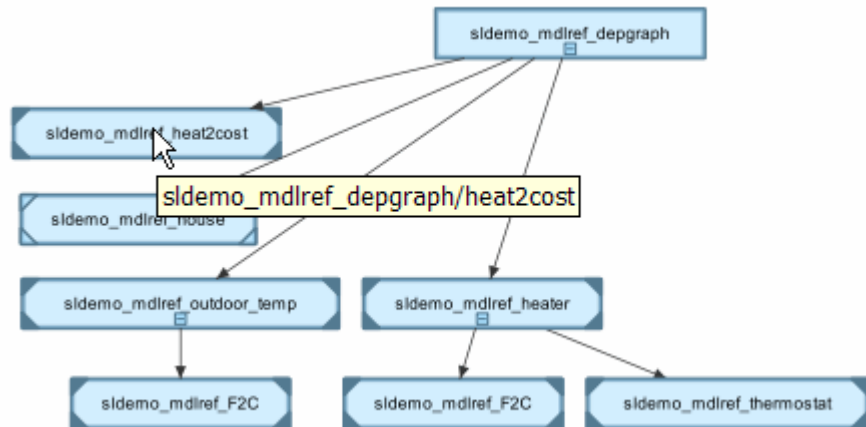
- Select **View > Show Legend** from the viewer's menu bar.

Displaying Full Paths of Referenced Model Instances. In an instance view (see “Referenced Model Instances View” on page 19-55) , 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.

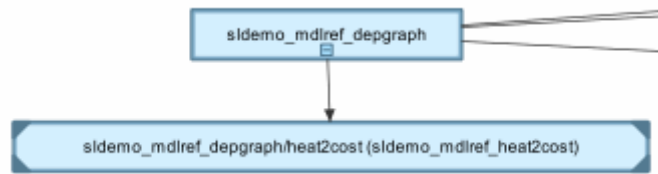
A tooltip appears, displaying the path displays the full path of the Model block corresponding to the instance.



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.

Managing Configuration Sets

- “Setting Up Configuration Sets” on page 20-2
- “Referencing Configuration Sets” on page 20-12

Setting Up Configuration Sets

In this section...

- “About Configuration Sets” on page 20-2
- “Configuration Set Components” on page 20-3
- “The Active Set” on page 20-3
- “Displaying Configuration Sets” on page 20-3
- “Activating a Configuration Set” on page 20-4
- “Opening Configuration Sets” on page 20-4
- “Copying, Deleting, and Moving Configuration Sets” on page 20-5
- “Copying Configuration Set Components” on page 20-6
- “Creating Configuration Sets” on page 20-6
- “Setting Values in Configuration Sets” on page 20-7
- “Configuration Set API” on page 20-7
- “Model Configuration Dialog Box” on page 20-10
- “Model Configuration Preferences Dialog Box” on page 20-11

About Configuration Sets

A *configuration set* is a named set of values for a model’s parameters, such as solver type and simulation start or stop time. Every new model is created with a default configuration set, called Configuration, that initially specifies default values for the model’s parameters. You can subsequently create and modify additional configuration sets and associate them with the model. The sets associated with a model can specify different values for any or all configuration parameters.

This section describes techniques for defining and using configuration sets that are stored in individual models. Such configuration sets are available only to the model that contains them. The next section, “Referencing Configuration Sets” on page 20-12, describes techniques for storing configuration sets in the base workspace, independently of any model. Such configuration sets can be shared by any number of models.

Configuration Set Components

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

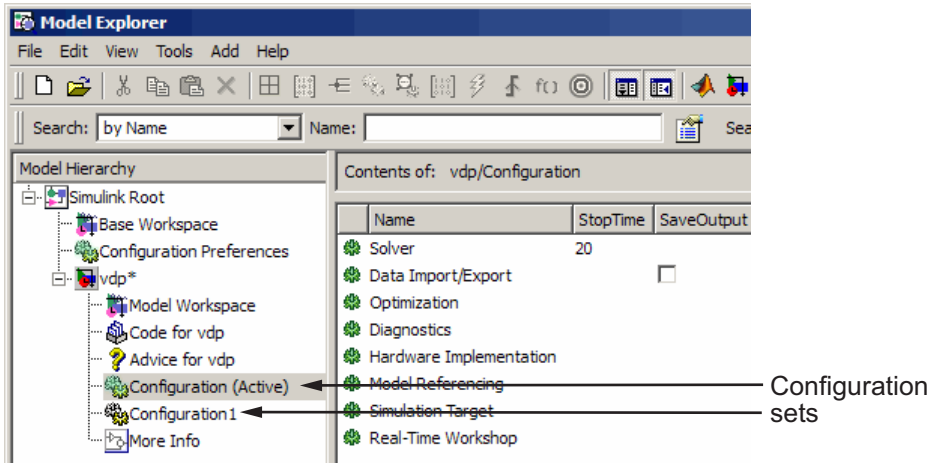
Some Simulink-based products, such as Real-Time Workshop, define additional components. If such a product is installed on your system, the configuration set also contains the components that it defines.

The Active Set

Only one of the configuration sets associated with a model is active at any given time. The active set determines the current values of the model's parameters. Changing the value of a parameter in the Model Explorer changes its value in the associated configuration set. 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, e.g., testing and production, or apply standard configuration settings to new models.

Displaying Configuration Sets

To display the configuration sets associated with a model, open the Model Explorer (see “The Model Explorer” on page 19-2). The configuration sets associated with the model appear as gear-shaped nodes in the Model Explorer's **Model Hierarchy** pane.



The Model Explorer’s **Contents** pane displays the components of the selected configuration set. The Model Explorer’s Dialog pane displays a dialog box for setting the parameters of the selected group (see “Configuration Parameters Dialog Box”).

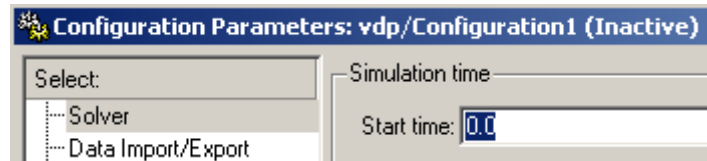
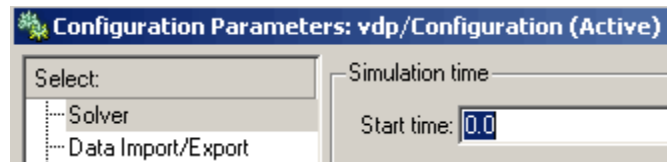
Activating a Configuration Set

To activate a configuration set, right-click the configuration set’s node to display the node’s context menu, then select **Activate** from the context menu.

Opening Configuration Sets

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 node’s context menu, then select **Open** from the context menu. You can open the Configuration Parameters dialog box for any configuration set, whether or not it is active. You might want to open a configuration set to inspect or edit the parameter settings.

The title bar of the dialog box indicates if 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 top-left corner label changes to reflect the state.

Copying, Deleting, and Moving Configuration Sets

You can use edit commands on the Model Explorer's **Edit** or context menus or object drag-and-drop operations to delete, copy, and move configuration sets among models displayed in the Model Explorer's **Model Hierarchy** pane.

For example, to copy a configuration set, using edit commands:

- 1** Select the model with a configuration set that you want to copy in the **Model Hierarchy** pane.
- 2** Select the configuration set that you want to copy in the **Contents** pane.
- 3** Select **Copy** from the Model Explorer's **Edit** menu or the configuration set's context menu.
- 4** Select the model in which you want to create the copy.

Note You can create a copy in the same model as the original.

- 5 Select **Paste** from the Model Explorer's **Edit** menu or from the model's context menu.

To copy the configuration set, using object drag-and-drop, hold the right mouse button down and drag the configuration set's 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's node to the node of the destination model.

Note You cannot move or delete a model's active configuration set.


Copying Configuration Set Components

To copy a configuration set component from one configuration set to another:

- 1 Select the component in the Model Explorer's **Contents** pane.
- 2 Select **Copy** from the Model Explorer's **Edit** menu or the component's context menu.
- 3 Select the configuration set into which you want to copy the component.
- 4 Select **Paste** from the Model Explorer's **Edit** menu or the component's context menu.

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 B's existing Solver component.

Creating Configuration Sets

To create a new configuration set, select **Configuration Set** from the Model Explorer's **Add** menu or press the **Add Configuration** button  in the Model Explorer's toolbar. You can also create a new configuration set by copying an existing configuration set.

Setting Values in Configuration Sets

To set the value of a parameter in a configuration set, select the configuration set in the Model Explorer and then edit the value of the parameter on the corresponding dialog box in the Model Explorer's dialog view.

Configuration Set API

An application program interface (API) lets you create and manipulate configuration sets from the command line or in a MAT-file or M-file. The API includes the `Simulink.ConfigSet` data object class and the following model construction commands:

- `attachConfigSet`
- `attachConfigSetCopy`
- `detachConfigSet`
- `getConfigSet`
- `getConfigSets`
- `setActiveConfigSet`
- `getActiveConfigSet`
- `openDialog`
- `closeDialog`

These commands, along with the methods and properties of `Simulink.ConfigSet` class, allow an M-file program to create and modify configuration sets, attach configuration sets to a model, set a model's active configuration set, open and close configuration sets, and detach configuration sets from a model. For example, to create a configuration set from scratch at the command line, enter

```
cfg_set = Simulink.ConfigSet
```

The default name of the new configuration set is `Configuration`. To change the name, execute

```
cfg_set.Name = 'name'
```

where *name* is the new name of the configuration set.

Use `get_param` and `set_param` to get and set the value of a parameter in a configuration set. For example, to specify the Simulink fixed-step discrete solver in the configuration set, execute

```
set_param(cfg_set, 'Solver', 'FixedStepDiscrete')
```

To save the configuration set in a MAT-file, execute

```
save mat_file cfg_set
```

where *mat_file* is the name of the MAT-file. To load the configuration set, execute

```
load mat_file
```

To prevent or allow a user to change the value of a parameter in a configuration set using either the Model Explorer or `set_param` command, execute

```
setPropEnabled(cfg_set, 'param', [0 | 1])
```

where *param* is the name of the parameter. To attach a configuration set to a model, execute

```
attachConfigSet(model, cfg_set)
```

where *model* is the model name (in quotes) or object. To get a model's active configuration set, execute

```
cfg_set = getActiveConfigSet(model)
```

To get a configuration set's full name (e.g., 'engine/Configuration'), execute

```
getFullName(cfg_set)
```

To set a model's active set, execute

```
setActiveConfigSet(model, 'cfg_set_name')
```

where *cfg_set_name* is the name of the configuration set.

Tip To find the name of a configuration set, execute

```
cfg_set_name = cfg_set.get_param('Name')
```

To open the Configuration Parameters dialog box for an active configuration set, execute

```
openDialog(cfg_set)
```

To open the Configuration Parameters dialog box for any configuration set, execute

```
cfg_set = getConfigSet(gcs, 'cfg_set_name')
openDialog(cfg_set);
```

where *cfg_set_name* is the name of the configuration set.

To close the Configuration Parameters dialog box for a configuration set, execute

```
closeDialog(cfg_set);
```

To rename the active configuration set of *modelA*, copy it, and attach a copy of that configuration to *modelB*, execute

```
activeConfigA = getActiveConfigSet('modelA');
activeConfigA.Name = 'myactiveConfigA';
newConfig = attachConfigSetCopy('modelB', activeConfigA);
```

where *activeConfigA* is the active configuration set of *modelA*. *modelA* is the model whose active configuration set you want to copy. *modelB* is the model with which you want to associate the copy of the configuration set. You can also use this command to make multiple copies of one configuration set. You can use this technique to programmatically assign copies of the same configuration set to multiple models.

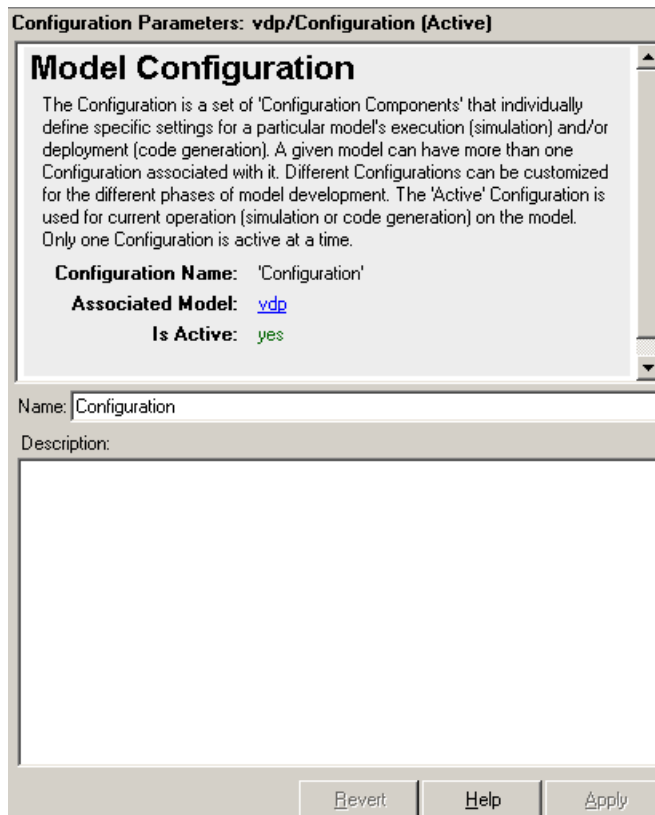
To detach a configuration set from a model, execute

```
detachConfigSet(model, cfg_set)
```

where *model* is the model name (in quotes) or object.

Model Configuration Dialog Box

The Model Configuration dialog box appears in the Model Explorer dialog pane when you select any model configuration.



You can edit the name and description of your configuration. See “Model Configuration Pane”.

Model Configuration Preferences Dialog Box

The Model Configuration Preferences dialog box appears in the Model Explorer dialog pane when you view the default model configuration.

- 1** Enable **View > Show Configuration Preferences** in the Model Explorer menu.
- 2** Select Configuration Preferences under the Simulink Root node in the Model Explorer **Model Hierarchy** pane.

You can also edit the configuration defaults in the Simulink Preferences window. See “Model Configuration Pane”.

Referencing Configuration Sets

In this section...

“Overview of Configuration References” on page 20-12

“Creating a Freestanding Configuration Set” on page 20-16

“Creating and Attaching a Configuration Reference” on page 20-17

“Obtaining a Configuration Reference Handle” on page 20-21

“Attaching a Configuration Reference to Other Models” on page 20-22

“Changing a Configuration Reference” on page 20-23

“Activating a Configuration Reference” on page 20-24

“Unresolved Configuration References” on page 20-24

“Getting Values from a Referenced Configuration Set” on page 20-25

“Changing Values in a Referenced Configuration Set” on page 20-25

“Replacing a Referenced Configuration Set” on page 20-27

“Building Models and Generating Code” on page 20-28

“Configuration Reference Limitations” on page 20-28

“Configuration References for Models with Older Simulation Target Settings” on page 20-29

Overview of Configuration References

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 you can use it with any models. 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 configuration set. The result is the same as if the referenced configuration set resides within the model.

Reasons to 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, and changing the value of any parameter in the set changes it for every model that uses the set. Use this feature to reconfigure many submodels quickly and ensure consistent configuration of parent models and referenced models.

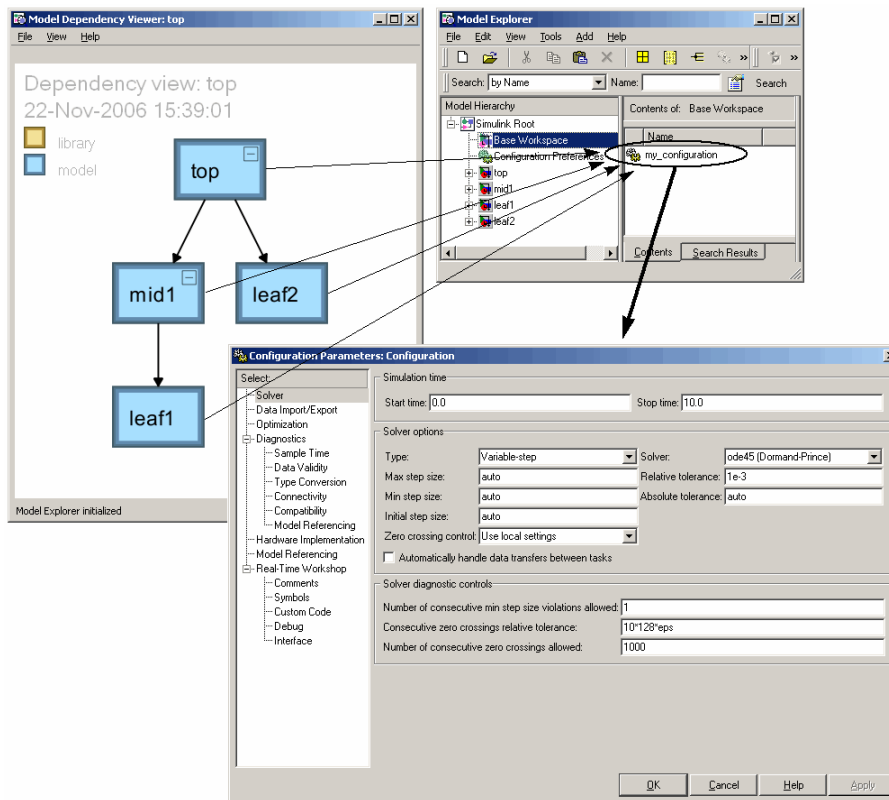
- **Replace the configuration sets of any number of models without changing the model files**

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 submodel that uses different configuration sets in different contexts contains a configuration reference that specifies the submodel configuration set as a variable. When you call an instance of the submodel, Simulink software assigns that variable a freestanding configuration set for the current context.

The next figure shows one way to use configuration references. Each model that appears in the Model Dependency Viewer specifies the configuration reference `my_reference` as its active configuration set, and `my_reference` points to the freestanding configuration set `Configuration`. The parameter values in `Configuration` therefore apply to all four models, and any parameter change in `Configuration` applies to all four models.



Basic Steps for Using a Configuration Reference

To use a configuration reference to link a freestanding configuration set to a model, follow these steps.

Step	Task	Reference
1	Create or obtain a configuration set and store it in the base workspace as the value of a MATLAB variable.	“Creating a Freestanding Configuration Set” on page 20-16

Step	Task	Reference
2	Create a configuration reference that specifies the relevant MATLAB variable.	“Creating and Attaching a Configuration Reference” on page 20-17
3	Attach the configuration reference to a model.	“Creating and Attaching a Configuration Reference” on page 20-17
4	Activate the configuration reference.	“Activating a Configuration Reference” on page 20-24
5	Access, set, and change configuration set parameters and the MATLAB variable as needed.	<ul style="list-style-type: none"> • “Getting Values from a Referenced Configuration Set” on page 20-25 • “Changing Values in a Referenced Configuration Set” on page 20-25 • “Replacing a Referenced Configuration Set” on page 20-27

Comparison of Configuration References and Configuration Sets

A configuration reference is an object of type `Simulink.ConfigSetRef`, and a configuration set is an object of type `Simulink.ConfigSet`. The two classes are similar in many ways. Wherever the same operation is applicable to both, the relevant functions and methods overload to work with either class. For example, you can **attach** or **activate** a configuration set or a configuration reference using the same GUI operations and API syntax.

You cannot nest configuration references, because only one level of indirection is available. You can obtain configuration parameter values by operating on a configuration reference as if it were the configuration set that it references. See “Getting Values from a Referenced Configuration Set” on page 20-25 for details. General information about configuration sets appears in “Setting Up Configuration Sets” on page 20-2.

Creating a Freestanding Configuration Set

All freestanding configuration sets reside in the base workspace as the values of MATLAB variables. 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. Trying to use it in a configuration reference causes an error. You can store a freestanding configuration set in the base workspace in these ways:

- Create and populate a new configuration set.
- Copy a configuration set that resides within a model.
- 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 variable. Use any technique to manipulate a freestanding configuration set and its parameter values that you use with a configuration set stored directly in a model.

Creating and Populating a New Configuration Set

You can create a configuration set in the base workspace using any of the techniques described in “Creating Configuration Sets” on page 20-6 or as follows:

```
cset = Simulink.ConfigSet
```

where *cset* is a new or existing base workspace variable. The new configuration set initially has default parameter values, copied from the default configuration set.

Copying a Configuration Set Stored in a Model

You can copy an existing configuration set to the base workspace using drag and drop operations described in “Copying, Deleting, and Moving Configuration Sets” on page 20-5, and assign the set to a MATLAB variable. For example:

```
cset = copy (getActiveConfigSet(md1))  
cset = copy (getConfigSet(md1, ConfigSetName))
```


where *mdl* is any open model, and *ConfigSetName* is the name of any configuration set attached to the model. The first example obtains the currently active configuration set. The second example obtains a configuration set by specifying the name under which it appears in the Model Explorer.

Be sure to copy any configuration set obtained from an existing model, as shown in the examples. Otherwise, *cset* refers to the existing configuration set stored in the model, rather than a new freestanding configuration set in the base workspace. In this case, any use of a configuration reference that links to *cset* causes an error.

Reading 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, as previously described, then save the variable to the MAT-file:

```
save (workfolder/ConfigSetName.mat, cset)
```

where *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

Any technique that executes the necessary code works.


Creating and Attaching a Configuration Reference

After you store a configuration set in the base workspace, you can link to that set from a configuration reference and attach it to a model. The model

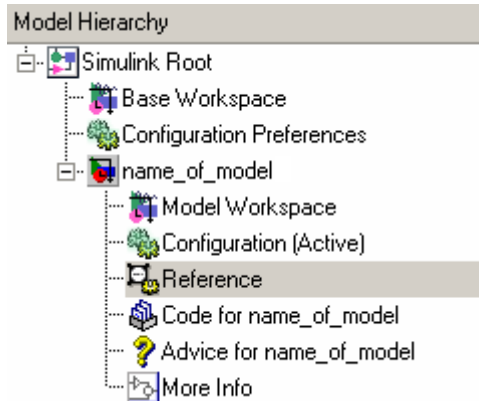
then has the same configuration parameters that it does as if the referenced configuration set resides directly in the model. You can attach any number of configuration references to a model. Each reference must have a unique name.

GUI Techniques

To create a configuration reference using the GUI:

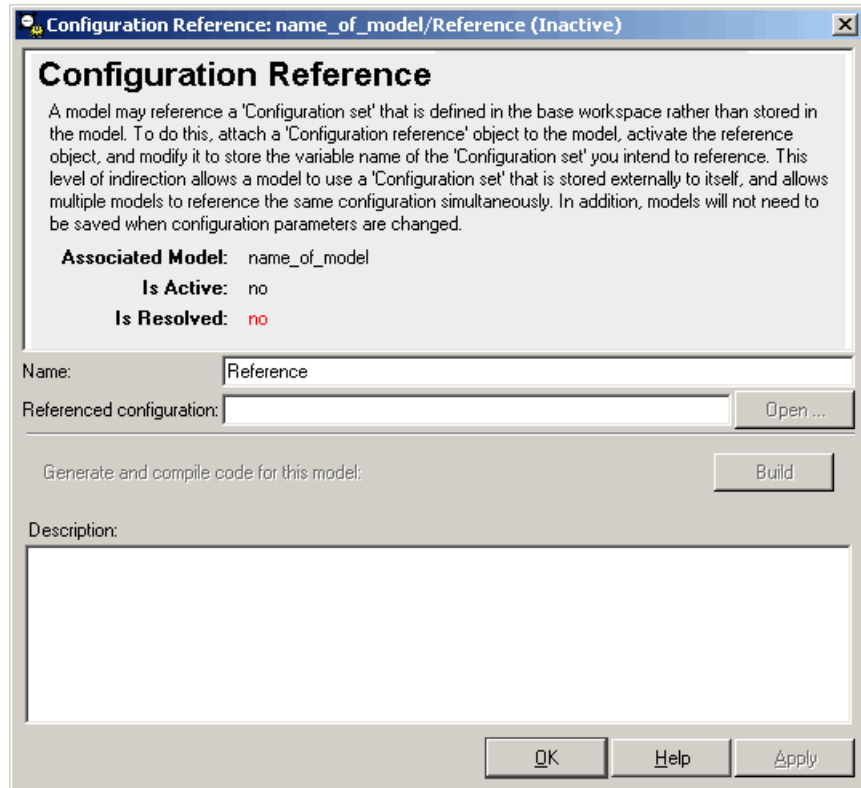
- 1 In the Model Explorer, select the model to which to attach the configuration reference.
- 2 Click the **Add Reference** tool  or select **Add > Configuration Reference**.

A new configuration reference attaches to the selected model. The default name of the new reference is **Reference**, with a digit appended if necessary to prevent name conflict. The name of the configuration reference appears in the Model Hierarchy pane under the Model Workspace icon, below the names of any configuration sets.



- 3 Select the new configuration reference in the Model Hierarchy pane, or right-click the configuration reference and select **Open** from the context menu.

The Configuration Reference dialog box appears in the Dialog pane or a separate window.

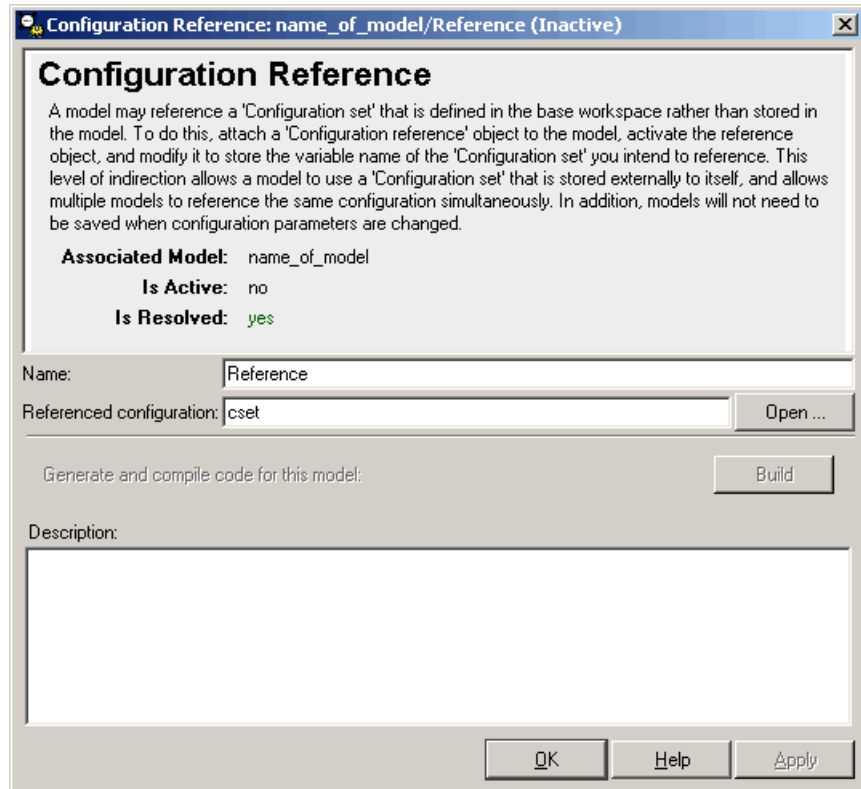


- 4 Change the default **Name** if desired. This name exists for human readability and does not affect the configuration reference functionally.
- 5 Specify the **Referenced configuration** set to be the base workspace variable whose value is the freestanding configuration set that you want to reference.

Tip Do not specify the name of a configuration reference. If you nest a configuration reference, an error occurs.

- 6 Click **OK** or **Apply**.

The **Is Resolved** field in the dialog box changes to **yes**.



If you do not specify a valid **Referenced configuration**, a warning appears. Using a configuration reference with an invalid **Referenced configuration** generates an error. The API equivalent of **Referenced configuration** is `WSVarName`. You can later use the GUI or API to correct the specification or provide a configuration set that has the correct name. See “Unresolved Configuration References” on page 20-24 for more information.

API Techniques

To create and populate a configuration reference 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(mdl, cref, true)
```

The third argument is optional and authorizes renaming to avoid a name conflict.

Using a configuration reference with an invalid `WSVarName` generates an error. The GUI equivalent of `WSVarName` is **Referenced configuration**. You can later use the API or GUI to correct the reference or provide a configuration set that has the correct name. See “Unresolved Configuration References” on page 20-24 for more information.

Obtaining a Configuration Reference Handle

Most functions and methods that operate on a configuration reference take a handle to the reference. If you create a configuration reference programmatically, with a statement like

```
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(mdl, 'ConfigSetRefName')
```

where *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 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.

Attaching a Configuration Reference to Other Models

After you create a configuration reference and attach it to a model, you can attach copies of the reference to any other models. To create and attach a copy of a configuration reference, you can use any technique you use to copy and attach a configuration set. See “Copying, Deleting, and Moving Configuration Sets” on page 20-5 and “Configuration Set API” on page 20-7.

Models do not share configuration reference objects. Each model has its own copy of any configuration reference attached to it, just as each has its own copy of any attached configuration set. Configuration references in different models establish configuration set sharing by specifying the same base workspace variable, which links different models to the same freestanding configuration set.

If you use the GUI, attaching an existing configuration reference to another model automatically attaches a copy, as distinct from a handle to the original. 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(mdl, ConfigSetRefName))  
attachConfigSet (mdl, 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 to 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 extra, you can remove it with `detachConfigSet`, as shown in the last line of the example.

```
% Get copy of original config set as
% a variable in the base workspace
open_system('mdl1')
% Get handle to local cset
cset = getConfigSet('mdl1', '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'
% Rename if name conflict occurs
attachConfigSet('mdl1', cref1, true)

% In a second model, create a configuration
% reference to the same cset
open_system('mdl2')
% Rename if name conflict occurs
attachConfigSetCopy('mdl2', cref1, true)
% Delete original cset from first model
detachConfigSet('mdl1', 'Configuration')
```

Changing a Configuration Reference

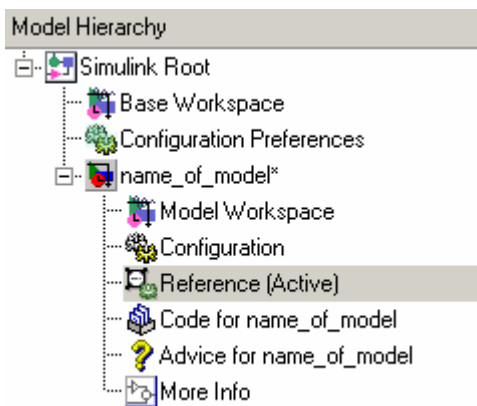
You can change an existing configuration reference by reopening its Configuration Reference dialog box and changing the **Name** or **Referenced configuration**. Similarly, you can use the API on an existing configuration reference to change the Name or WSVarName. If you refer to a configuration set that does not yet exist, no error occurs, but the configuration reference is unusable. The configuration reference becomes usable as soon as the configuration set exists.

Activating a Configuration Reference

After you create a configuration reference and attach it to a model, activate it using any technique that activates a configuration set in the model:

- From the GUI, select **Activate** from the context menu of the configuration reference.
- 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 is **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 that contains the reference.

Unresolved Configuration References

When a configuration reference does not specify 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.

“Creating and Attaching a Configuration Reference” on page 20-17 describes techniques for resolving configuration references.

Getting 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')
```

Changing Values in a Referenced Configuration Set

You cannot change a configuration set in any way, such as changing configuration parameter values, by operating on a configuration reference. With `cref` and `cset` as before, if you execute:

```
set_param (cset, 'StopTime', '300')
set_param (cref, 'StopTime', '300')           % ILLEGAL
```

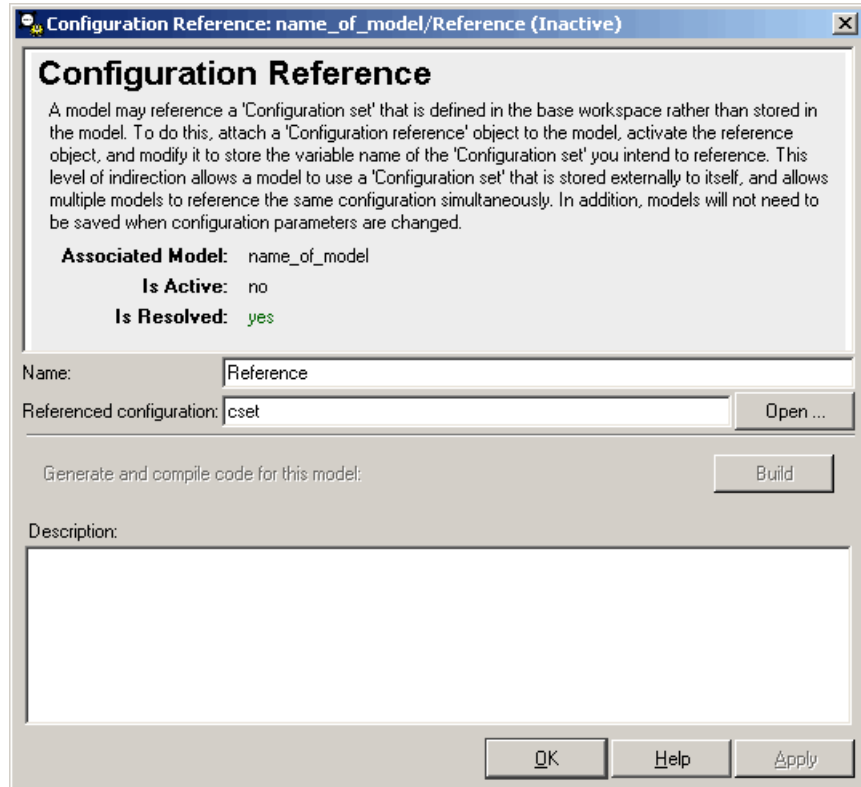
the first operation succeeds, but the second causes an error. Instead, you must obtain the configuration set itself and change it directly, using the GUI or the API.

GUI Techniques

To obtain a referenced configuration set using the GUI:

- 1 Select the configuration reference in the Model Hierarchy pane, or right-click the configuration reference and select **Open** from the context menu.

The Configuration Reference dialog box appears in the Dialog pane or a separate window.



- 2 Click **Open** to the right of the **Referenced configuration** field.

The Configuration Parameters dialog box opens on the configuration set specified by **Referenced configuration**. You can now change and apply or save parameter values as you would for any configuration set.

API Techniques

To obtain a referenced configuration set using the API:

- 1 Obtain a handle to the configuration reference, as described in “Obtaining a Configuration Reference Handle” on page 20-21.

- 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.

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 stale. To avoid this situation, execute:

```
cref.refresh
```

where *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 config reference
cref1 = Simulink.ConfigSetRef;
% Resolve the config ref to the set
cref1.WsVarName = 'cset1';
% Attach the config ref 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.

Building Models and Generating Code

The Real-Time Workshop 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. You can use 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 effect as its equivalent in the Configuration Parameters dialog box and operates on the model that contains the configuration reference.

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 “Replacing a Referenced Configuration Set” on page 20-27.
- 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 applies, even if an active configuration reference points to that target. For more information about `ActivateCallback` functions, see “rtwgensettings Structure” in the Real-Time Workshop Embedded Coder Developing Embedded Targets documentation.

Configuration References for Models with Older Simulation Target Settings

Suppose that you have a nonlibrary model that contains one of these blocks:

- Embedded MATLAB Function block
- Stateflow chart
- Truth Table block
- Attribute Function block

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 you use:

- 1** A copy of the configuration set in the base workspace attaches to the model.
- 2** The simulation target settings migrate to the corresponding parameters in this new configuration set.
- 3** The new configuration set becomes active.
- 4** 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.

Optional Migration Process That Restores the Configuration Reference

If you want your models to retain an active configuration reference, use the `updatecsref` utility. This utility can compare the simulation target settings among all specified nonlibrary models with the parameters of the configuration set in the base workspace.

- If all settings are consistent with parameters in the shared configuration set, the utility does not perform any updates on the shared set.
- If all settings are not consistent, the utility provides a list of differences in the Command Window.

The utility copies simulation target settings to the shared configuration set only if you approve the changes.

Suppose that you have models created in R2008a that contain:

- Stateflow charts
- Identical, non-default simulation target settings
- Active configuration references that point to the same configuration set in the base workspace

To run the `updatecsref` utility on these models, follow these steps:

1 At the command prompt, type:

```
updatecsref('modelfile1', 'modelfile2', ..., 'modelfileN')
```

2 For each model, the utility looks for the active configuration reference that points to a configuration set in the base workspace. Sample messages include:

```
Analyzing 'active_csref_8a_v1'
Found the following valid configuration reference(s) in model 'active_csref_8a_v1'
[1] Configuration reference 'Ref' points to the base workspace configuration set
object 'cset'
It will be used as the active configuration set. Do you want to continue? (Y/n)
```

```
Analyzing 'active_csref_8a_v2'
Found the following valid configuration reference(s) in model 'active_csref_8a_v2'
[1] Configuration reference 'Ref' points to the base workspace configuration set
object 'cset'
It will be used as the active configuration set. Do you want to continue? (Y/n)
```

If you approve this shared configuration set, choose **Y**. Otherwise, choose **n** to stop the migration process.

3 All differences between the simulation target settings and the shared configuration set appear. Sample messages include:

```
Parameter 'SFSimEcho' of 'cset' will be changed from 'on' to 'off'
Parameter 'SFSimOverflowDetection' of 'cset' will be changed from 'on' to 'off'
Do you want to proceed and change base workspace object 'cset'? (Y/n)
```

If you agree with all changes to the configuration set in the base workspace, choose **Y**. Otherwise, choose **n** to stop the migration process.

Note If any simulation target settings between nonlibrary models are inconsistent, these messages appear in the Command Window:

- A list of the differences
- A request to update your models to ensure consistent settings

In this case, the utility exits automatically. After you update your models as requested, you can start over from step 1.

- 4** The utility displays messages about updates to your nonlibrary models and the shared configuration set.

Your models continue to use an active configuration reference to point to the configuration set in the base workspace.

Note For more information about using the `updatecsref` utility, type `help updatecsref` at the command prompt.

Running Simulations

- “Simulation Basics” on page 21-2
- “Controlling Execution of a Simulation” on page 21-3
- “Specifying a Simulation Start and Stop Time” on page 21-8
- “Choosing a Solver” on page 21-9
- “Interacting with a Running Simulation” on page 21-19
- “Saving and Restoring the Simulation State as the SimState” on page 21-20
- “Diagnosing Simulation Errors” on page 21-27

Simulation Basics

You can simulate a model at any time simply by clicking the **Start** button on the Model Editor displaying the model (see “Starting a Simulation” on page 21-3). 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. Specifying simulation options is called “configuring the model”. With the 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 “Setting Up Configuration Sets” on page 20-2 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 “Pausing or Stopping a Simulation” on page 21-5), and launch simulations of other models. If an error occurs during a simulation, Simulink halts the simulation and a diagnostic viewer pops up that helps you to determine the cause of the error.

Controlling Execution of a Simulation

In this section...

“Starting a Simulation” on page 21-3

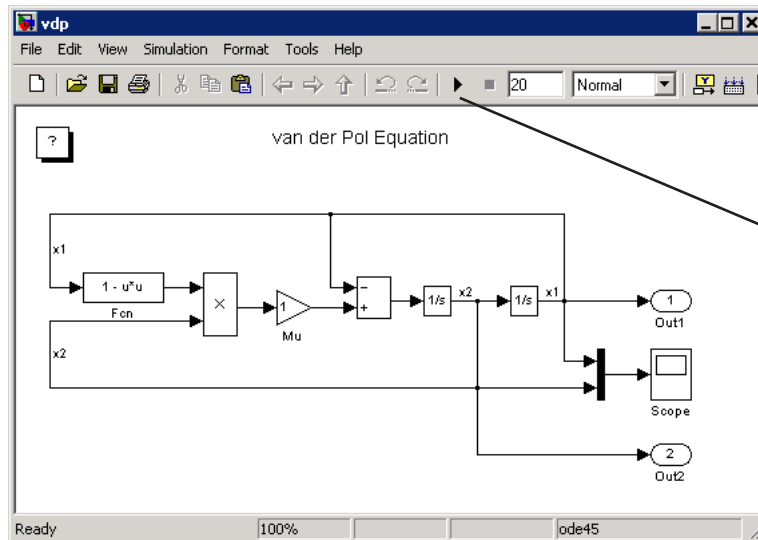
“Pausing or Stopping a Simulation” on page 21-5

“Using Blocks to Stop or Pause a Simulation” on page 21-5

Starting a Simulation

This sections explains how to run a simulation interactively. See Chapter 22, “Running a Simulation Programmatically” 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 **Start** or click the **Start** button on the model toolbar.

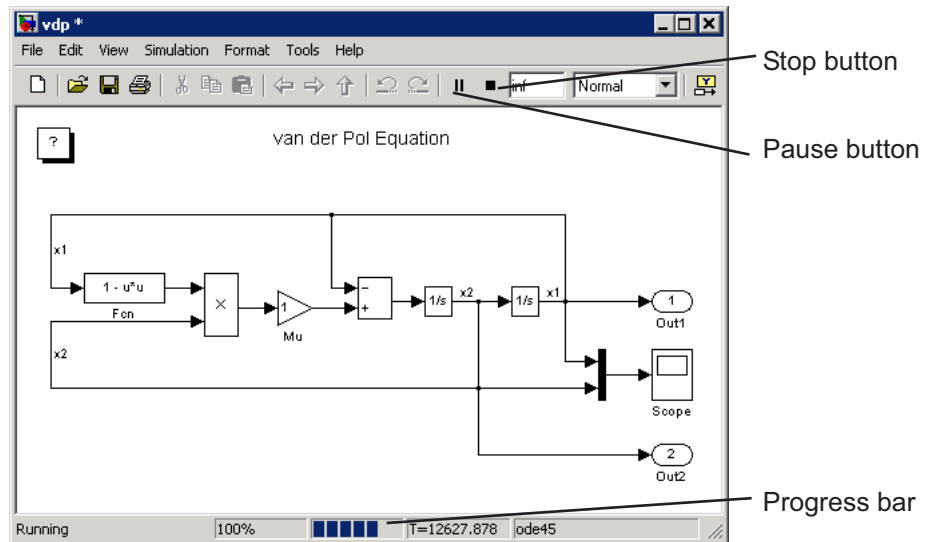


Start button

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 (see “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 **Stop** command replaces the **Start** command on the **Simulation** menu. A **Pause** command appears on the menu and replaces the **Start** button on the model toolbar.



Your computer beeps to signal the completion of the simulation.

Pausing or Stopping 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 **Start** 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 **Start** 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.

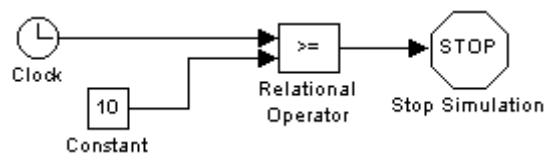
Using Blocks to Stop or Pause a Simulation

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:

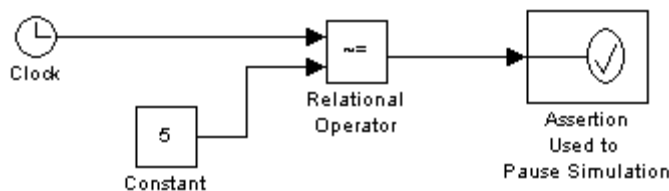
- 1 Drag a copy of the Assertion block from the Model Verification library and drop it into your model.
- 2 Connect the Assertion block to a signal whose value becomes zero at the desired pause time.
- 3 Open the Block Parameters dialog box of the Assertion block .
 - Enter the following commands into 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 Chapter 22, “Running a Simulation Programmatically” for more information on using the `set_param` command to control the execution of a Simulink model.

Specifying a 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.

Choosing a Solver

In this section...

“What Is a Solver?” on page 21-9

“Choosing a Solver Type” on page 21-9

“Choosing a Fixed-Step Solver” on page 21-11

“Choosing a Variable-Step Solver” on page 21-15

What Is a Solver?

A solver is a component of the Simulink software that determines the next simulation time step. In making this determination, the solver satisfies the target accuracy requirements that you specify. The Simulink product provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. The following sections explain how to choose the solver best suited to your application. For information on tailoring the selected solver to your model, see “Improving Simulation Accuracy” on page 23-10.

Choosing a Solver Type

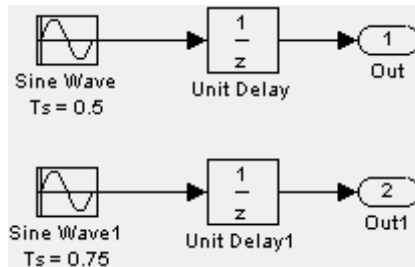
Solvers are divided into two types: fixed-step and variable-step. Both types of solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the step size. With a fixed-step solver, the step size remains constant throughout the simulation. In contrast, with a variable-step solver, the step size can vary from step to step, depending on the model dynamics. In particular, a variable-step solver reduces the step size when the model states are changing rapidly to maintain accuracy. Similarly, the variable-step solver increases the step size when the model states are changing slowly and thus avoids unnecessary steps. The **Type** control on the Simulink **Solver** configuration pane allows you to select either of these two types of solvers (see “Solver Pane”).

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. The rationale for this decision is that real-time computer systems operate at fixed-size signal sample rates. A variable-step solver may

cause the simulation to miss error conditions that can occur on a real-time computer system.

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 can shorten the time required to simulate your model significantly. A variable-step solver allows such savings because, for a given level of accuracy, the variable-step solver can dynamically adjust the step size as necessary and thus reduce the number of steps required. Whereas the fixed-step solver must use a single step size throughout the simulation based upon the accuracy requirements. In order to satisfy such requirements throughout the simulation, the solver might require a very small step.

The following model illustrates 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 ...]

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

[0.0 0.5 0.75 1.0 1.5 2.0 2.25 ...]

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

The variable-step discrete solver uses zero-crossing detection (see “Zero-Crossing Detection” on page 2-24) to handle continuous signals. Simulink uses this solver by default if you specify a continuous solver and if your model has no continuous states.

Choosing a Fixed-Step Solver

When you set the **Type** control of the **Solver** configuration pane to **fixed-step**, the adjacent **Solver** control allows you to choose one of the fixed-step solvers provided. The set of fixed-step solvers comprises two types of solvers: discrete and continuous. Both of these types rely upon the model blocks to compute the values of any discrete states. Blocks that define discrete states compute the values of those states at each time step. Unlike the discrete solvers, the continuous solvers compute the continuous states defined by blocks through numerical integration. Therefore, the very first step is to decide whether you need to use a discrete or a continuous solver.

If your model has no states or discrete states only, choose the fixed-step discrete solver. If your model has continuous states, you must choose either one of the explicit fixed-step continuous solvers (ode1, ode2, ode3, ode4, ode5, ode8), “Explicit Fixed-Step Continuous Solvers” on page 21-13 or the implicit fixed-step continuous solver (ode14x), “Implicit Fixed-Step Continuous Solvers” on page 21-14. Compared to the explicit fixed-step solvers, the ode14x implicit solver provides for high accuracy and stability but is computationally expensive.

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 size (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 attempt to use the fixed-step discrete solver to update or simulate a model that has continuous states, an error message appears. Thus, updating or simulating a model is a quick way to determine whether it 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 employ 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.

The two distinct types of fixed-step continuous solvers provided are: explicit and implicit solvers. Explicit solvers (see “Explicit Fixed-Step Continuous Solvers” on page 21-13) 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:

$$X(n+1) = X(n) + h * DX(n)$$

where X is the state, DX is the state derivative, and h is the step size, and n indicates the current time step. An implicit solver (see “Implicit Fixed-Step Continuous Solvers” on page 21-14) 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,

$$X(n+1) - X(n) - h*DX(n+1) = 0$$

This type of solver requires more computation per step than an explicit solver but is also more accurate for a given step size. This solver thus can be faster than explicit fixed-step solvers for certain types of stiff systems.

Explicit Fixed-Step Continuous Solvers. 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 of the available solvers and the integration technique it uses.

Solver	Integration Technique
ode1	Euler's Method
ode2	Heun's Method
ode3	Bogacki-Shampine Formula
ode4	Fourth-Order Runge-Kutta (RK4) Formula
ode5	Dormand-Prince (RK5) Formula
ode8	Dormand-Prince RK8(7) Formula

The integration techniques used by the fixed-step continuous solvers trade accuracy for computational effort. 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).

As with the fixed-step discrete solver, the accuracy and the duration of a simulation driven by a fixed-step continuous solver depends 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 is capable of handling 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 or that the model cannot be simulated in less time with a less complex solver. Depending on the dynamics of your model, you might need to choose another solver and/or a different sample time to achieve both acceptable accuracy and an acceptable simulation time.

Implicit Fixed-Step Continuous Solvers. This category provides one solver : `ode14x`. This solver uses a combination of Newton’s method and extrapolation from the current value to compute the value of a model 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)” “Fixed-step solver size (fundamental sample time)” in the online documentation). 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 thus generally requires experimentation.

Here is the most efficient way to choose the best fixed-step solver for your model experimentally. First, use one of the variable-step solvers to simulate your model to the level of accuracy that you desire. These results will give you a good approximation of the correct simulation results. Next, use `ode1` to simulate your model at the default step size for your model. Compare the results of simulating your model with `ode1` with the results of simulating with 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 the `ode1` is the simplest of the fixed-step solvers and hence yields the shortest simulation time for the current step size.

If `ode1` does not give accurate 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. First, try `ode3`. If it gives accurate results, try `ode2`. If `ode2` gives accurate results, it is the best solver for your model; otherwise, `ode3` is the best. 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`.

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 with the rate of change of the model states. The choice between the two types of 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. They reduce the step size to increase the accuracy when the model states are changing rapidly and increase the step size to avoid unnecessary steps when the model states are changing slowly. Computing the step size at each time step adds to the computational overhead but can reduce the total number of steps, and hence the simulation time required to maintain a specified level of accuracy. This capability is particularly important for models with rapidly changing or piecewise continuous states.

Following are the variable-step continuous solvers provided. They comprise both one-step and multistep solvers.

- `ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver; that is, in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best solver to apply as a first try for most problems. For this reason, `ode45` is the default solver used for models with continuous states.
- `ode23` is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It can be more efficient than `ode45` at crude tolerances and in the presence of mild stiffness. `ode23` is a one-step solver.
- `ode113` is a variable-order Adams-Bashforth-Moulton PECE solver. It can be more efficient than `ode45` at stringent tolerances. `ode113` is a *multistep* solver; that is, it normally needs the solutions at several preceding time steps to compute the current solution.
- `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), also known as Gear's method. The `ode15s` solver is a multistep solver that numerically generates the Jacobian matrices. If you suspect that a problem is stiff, or if `ode45` failed or was highly inefficient, try `ode15s`.
- `ode23s` is based on a modified Rosenbrock formula of order two. Because it is a one-step solver, it can be more efficient than `ode15s` at crude tolerances. Like `ode15s`, `ode23s` numerically generates the Jacobian for you. However, it can solve certain kinds of stiff problems for which `ode15s` is not effective.
- `ode23t` is an implementation of the trapezoidal rule using a “free” interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.
- `ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. 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 not designed for stiff problems are ineffective on intervals where the solution changes slowly because they 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.

Specifying Variable-Step Solver Error Tolerances

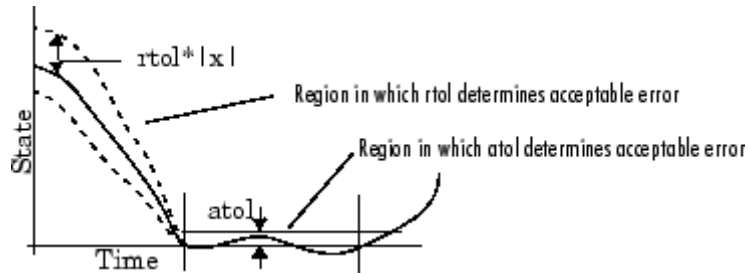
The solvers use standard local error control techniques to monitor the error at each time step. During each time step, the solvers compute the state values at the end of the step and determine the *local error*—the estimated error of these state values. They then compare the local error to the *acceptable 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* 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.



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 is reset to the maximum value that the state has assumed, thus far, times the relative tolerance for that state. Thus, if a state changes from 0 to 1 and the `reltol` is $1e-3$, then by the end of the simulation the `abstol` is set to $1e-3$ also. If a state goes from 0 to 1000, then the `abstol` is set 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.

The Integrator, Transfer Fcn, State-Space, and Zero-Pole blocks allow you to specify absolute tolerance values for solving the model states that they compute or that determine their output. 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 in this way. For example, if the global setting does not provide sufficient error control for all of your model states because they vary widely in magnitude, then you might want to set the value yourself.

Interacting 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 on a computer running the Microsoft Windows operating system (see “Block Data Tips” on page 7-2).

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.

Saving and Restoring the Simulation State as the SimState

In this section...

“Overview of the SimState” on page 21-20

“How to Save the SimState” on page 21-21

“How to Restore the SimState” on page 21-22

“How to Change the States of a Block within the SimState” on page 21-24

“Using the SimState Interface Checksum Diagnostic” on page 21-24

“Limitations of the SimState ” on page 21-25

“Using SimState within S-Functions” on page 21-26

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.

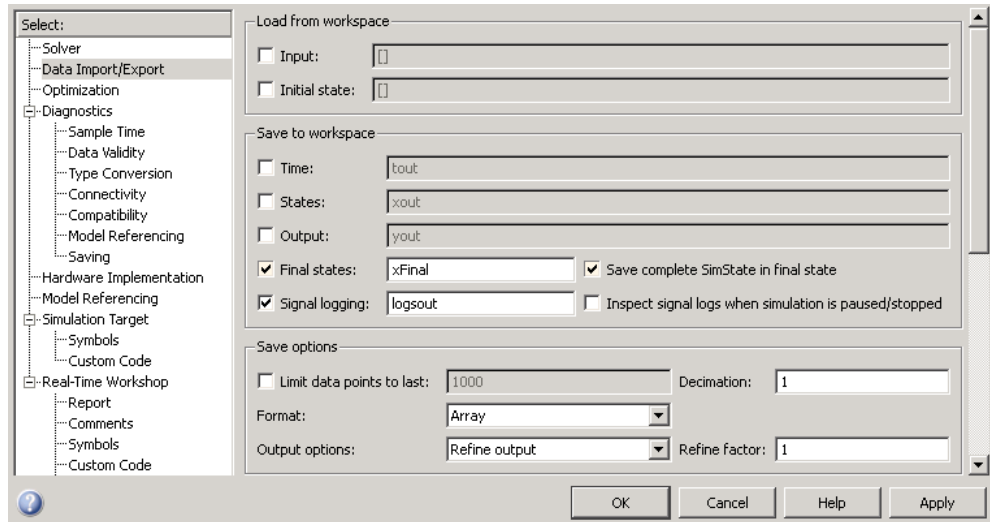
Note The **Final states** option of the **Data Import/Export** pane in Simulink saves only *logged* states—the continuous and discrete states of blocks—which are a subset of the complete simulation state of the model. Hence you cannot use the **Final states** to save and restore the complete simulation state as the initial state of a new simulation.

How to 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 > Start**.



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')
```

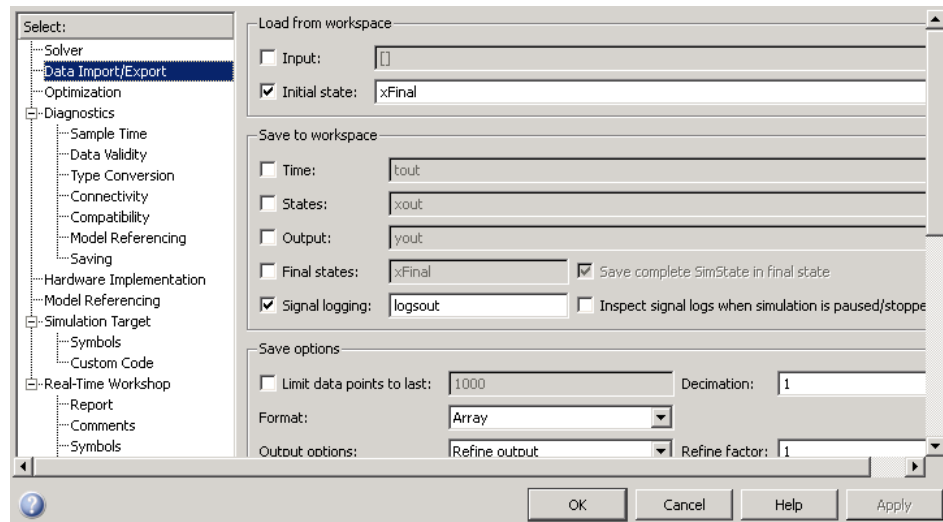
How to Restore the SimState

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`.

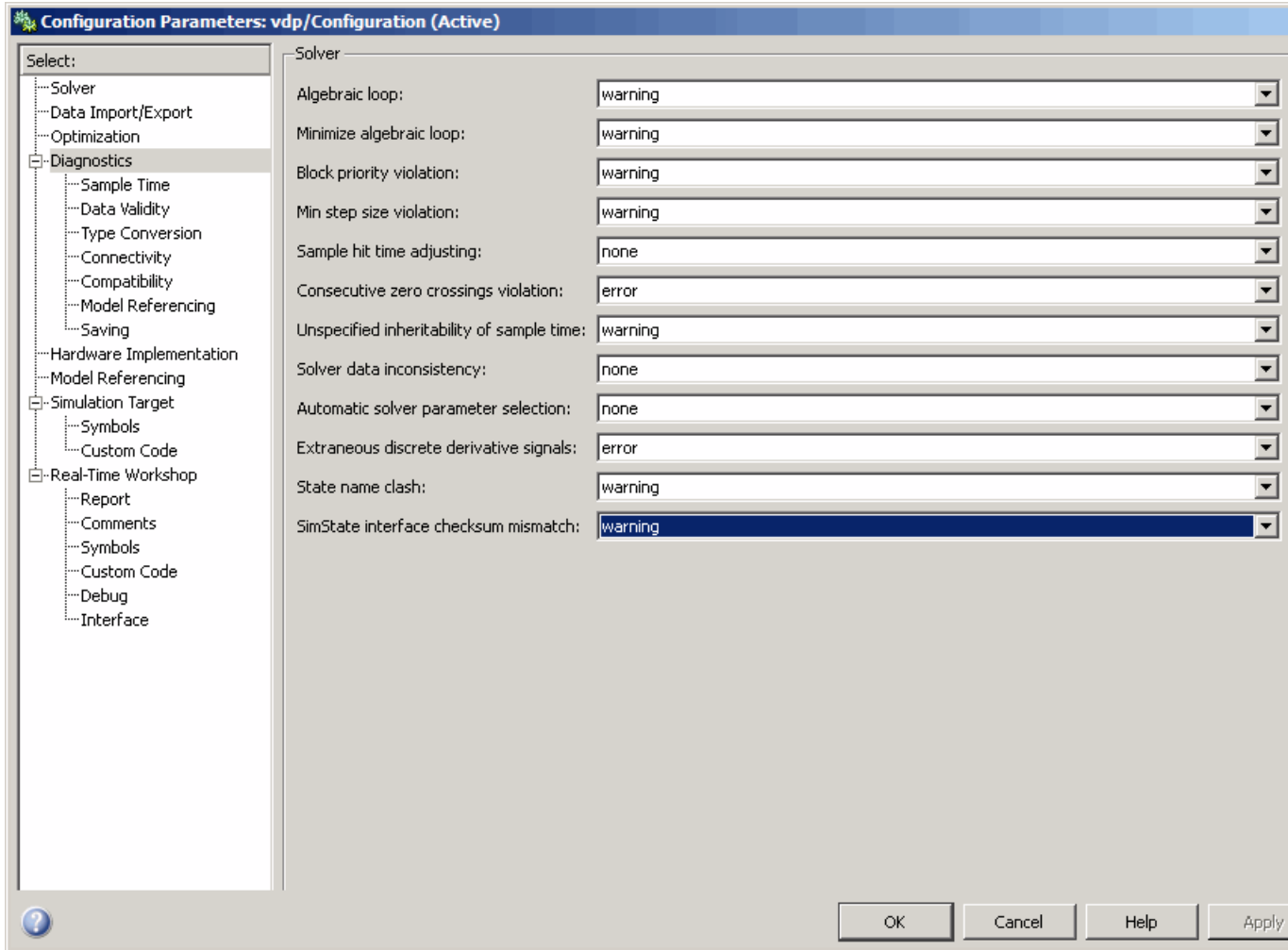
```
set_param mdl, 'LoadInitialState', 'on', 'InitialState',...  
[mdl 'SimState']  
simOut = sim(mdl, 'StopTime', tstop)
```

How to 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.

Using the 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.
- You cannot save the `SimState` in one version of Simulink and restore it to another version of Simulink.

Using `SimState` within S-Functions

Special APIs for C and Level-2 M 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`”.

Diagnosing Simulation Errors

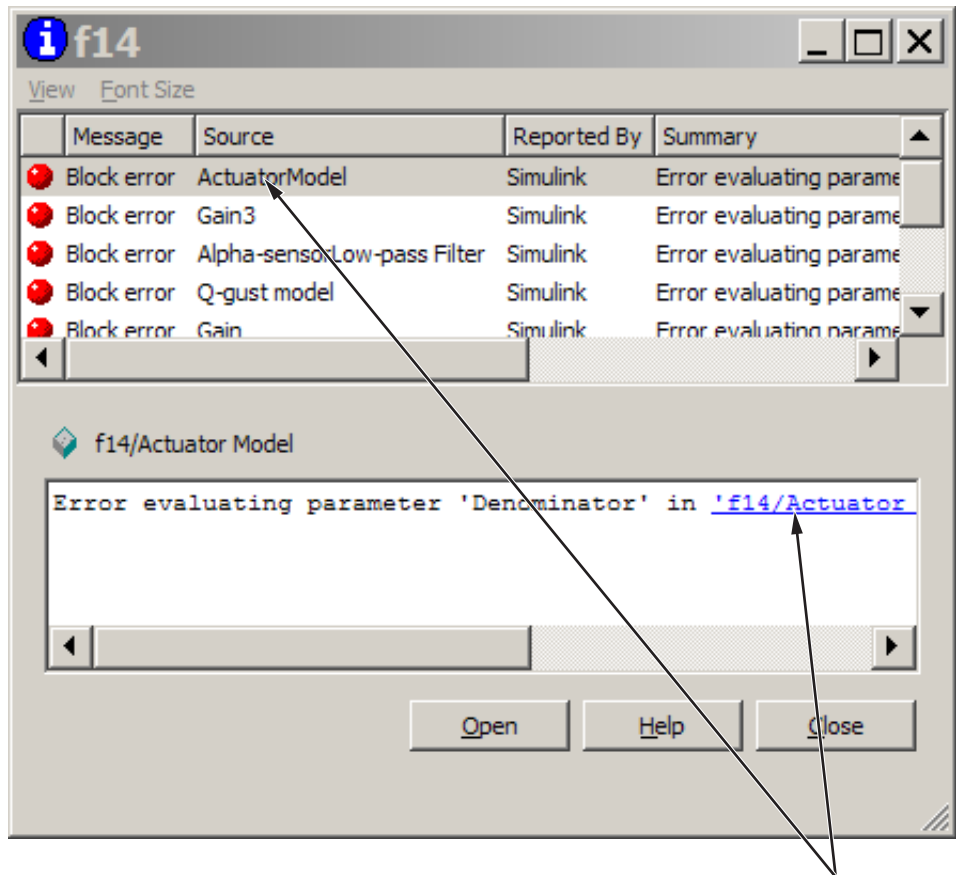
In this section...
“Response to Run-Time Errors” on page 21-27
“Simulation Diagnostics Viewer” on page 21-27
“Creating Custom Simulation Error Messages” on page 21-30

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 Real-Time Workshop product).

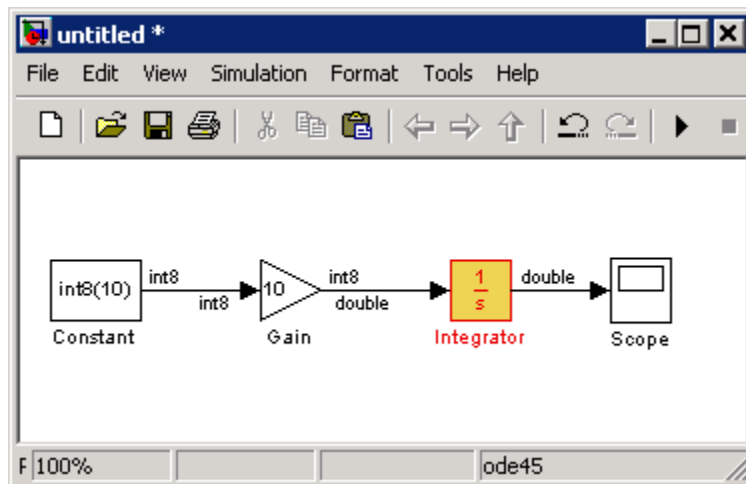
Summary. Error message, abbreviated to fit in the column

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.

Creating 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 Fcn block executes.

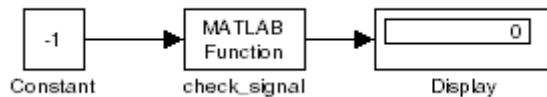
You can use the MATLAB error function in callbacks, S-functions or the MATLAB Fcn block to create custom error messages specific to your application. Following are the capabilities available to messages:

- 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`.

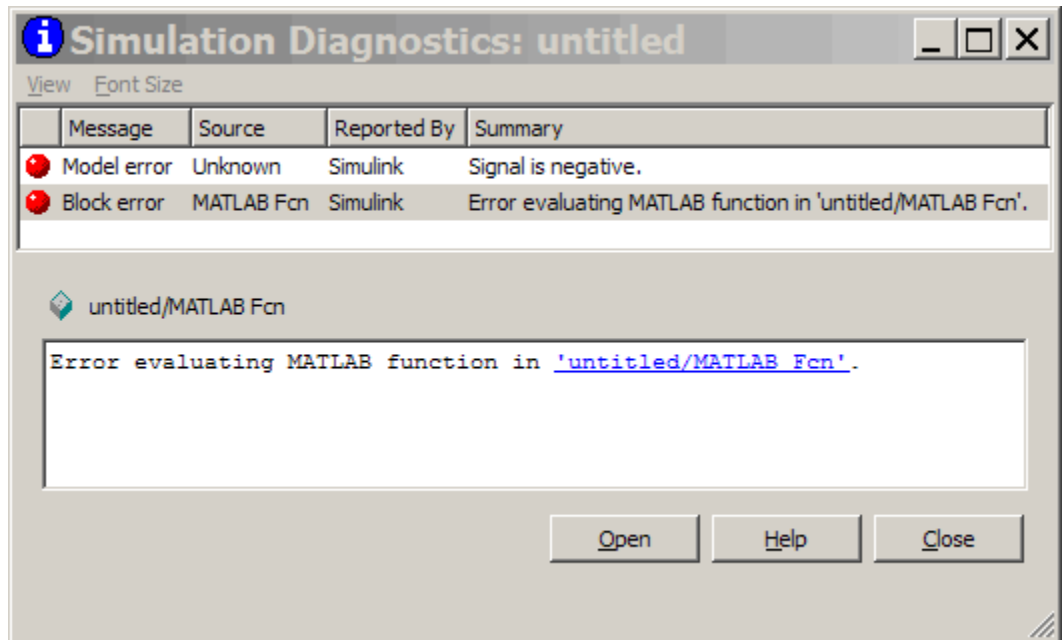


The MATLAB Fcn block invokes the following function:

```
function y=check_signal(x)
    if x<0
        error('Signal is negative');
    else
        y=x;
    end
end
```

end

Executing this model displays the error message in the Simulation Diagnostics Viewer.



Creating Hyperlinks to Files, Directories, or Blocks

To include a hyperlink to a block, file, or directory 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 directory "c:/work"')`

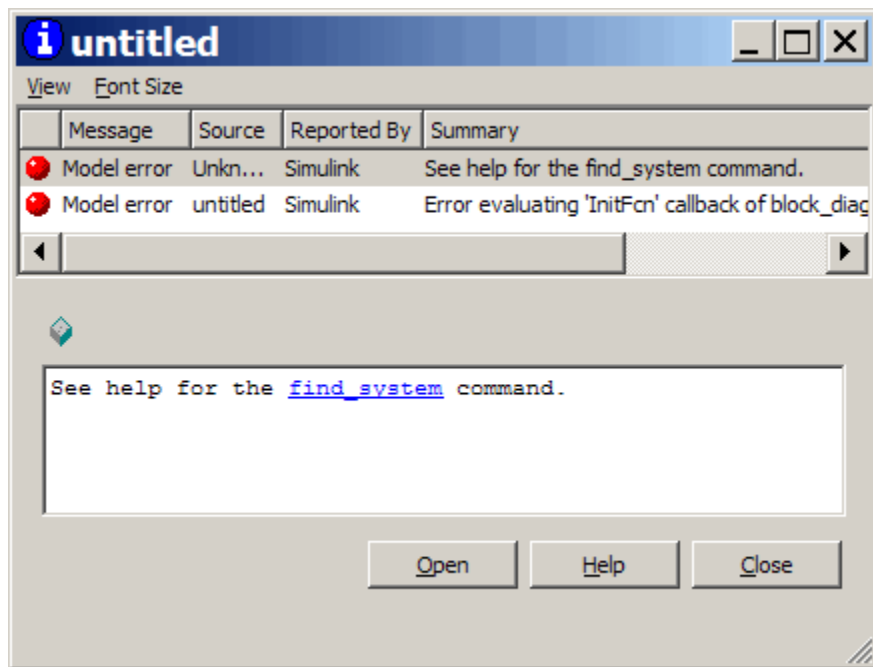
displays a text hyperlink to the `c:/work` directory. Clicking the link opens a system command window (shell) and sets its working directory to `c:/work`.

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 Viewer 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 22-2
- “Using the sim Command” on page 22-3
- “Using the set_param Command” on page 22-5
- “Running Parallel Simulations” on page 22-7
- “Error Handling in Simulink Using MSLErrorException” on page 22-14

About Programmatic Simulation

Entering simulation commands in the MATLAB Command Window or from an M-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.

Using the sim Command

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', 'ParameterName1', Value1,
'ParameterName2', Value2...);
SimOut = sim('model', ParameterStruct);
SimOut = sim('model', ConfigSet);
```

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 **Return 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 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.

For complete details of the `sim` command syntax, see the `sim` reference page. For information about the configuration parameters, see “Configuration Parameters Dialog Box”.

Examples of Implementing the `sim` Command

Following are examples that demonstrate 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.find('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.

```
mdl = 'vdp';
```

```

load_system mdl
simMode = get_param mdl, 'SimulationMode';
set_param mdl, 'SimulationMode', 'rapid'
cs = getActiveConfigSet mdl;
mdl_cs = cs.copy;
set_param mdl_cs, 'AbsTol', '1e-5', ...
    'SaveState', 'on', 'StateSaveName', 'xoutNew', ...
    'SaveOutput', 'on', 'OutputSaveName', 'youtNew'
simOut = sim mdl, mdl_cs;
set_param mdl, 'SimulationMode', simMode

```

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 “Running Parallel Simulations” on page 22-7.

Using the set_param Command

You can use the `set_param` command to start, stop, pause, or continue a simulation, to update a block diagram, or to write all data logging variables to the base workspace. The format of the `set_param` command is:

```
set_param('sys', 'SimulationCommand', 'cmd')
```

where `'sys'` is the name of the system and `'cmd'` is `'start'`, `'stop'`, `'pause'`, `'continue'`, `'update'`, or `'WriteDataLogs'`. See Chapter 15, “Importing and Exporting Data” for information about data logging.

Similarly, you can use the `get_param` command to check the status of a simulation. The format of the `get_param` function call for this use is

```
get_param('sys', 'SimulationStatus')
```

The Simulink software returns 'stopped', 'initializing', 'running', 'paused', 'updating', 'terminating', or 'external' (used with the Real-Time Workshop product).

Note If you use `matlab -nodisplay` to start a session, then you cannot use `set_param` to run your simulation session.

Running a 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.

Running Parallel Simulations

Overview of Calling `sim` from within `parfor`

The MATLAB `parfor` command allows you to run parallel Simulink simulations. 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 for details on selecting a simulation mode and “Designing Your Model for Effective Acceleration” on page 27-14 for optimizing simulation run times.) For other simulation modes, you need to address any workspace access issues and data concurrency issues in order 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 details about the `parfor` command, see `parfor`.

Note All simulation modes require a homogeneous file system. The workers and the callers must be on the same operating system and have the same type of file system (for example, NTFS or FAT).

`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 27-7.

As for the third condition, the following sample code demonstrates how to disable the up-to-date check using the `sim` command.

```
% Load the model and set parameters
mdl = 'vdp';
load_system(mdl)
set_param(mdl, 'SimulationMode', 'rapid')
% Build the Rapid Accelerator target
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl);
% Run parallel simulations
parfor i=1:4
    simOut{i} = sim(mdl,...
        'RapidAcceleratorUpToDateCheck', 'off',...
        'SaveTime', 'on',...
        'StopTime', num2str(10*i));
end
```

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 demonstration of this method of running parallel simulations, refer to the Rapid Accelerator Simulations Using PARFOR demo.

Workspace Access Issues

In order 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 “Using Model Workspaces” on page 4-66.

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
mdl = 'vdp';
paramName = 'StopTime';
paramValue = {'10', '20', '30', '40'};
parfor i=1:4
    simOut{i} = sim(mdl, ...
                   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(mdl, 'ExternalInput', 'extInp');
end
```

For further details, see the Rapid Accelerator Simulations Using PARFOR demo.

Data Concurrency Issues

Data concurrency issues 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 or Embedded MATLAB block. The cause, in this case, is that Simulink tries to concurrently access target data from multiple worker sessions. Similarly, ToFile 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 EDA Simulator Link™ for use with the Mentor Graphics® ModelSim® HDL simulator.

Resolving Data Concurrency Issues

The core requirement of `parfor` is the independence of the different iterations of the `parfor` body. This restriction is not compatible with the core requirement of simulation via incremental code generation, for which the simulation target from a prior simulation is reused or updated for the current simulation. Hence parallel simulation of models that involve code generation (such as Accelerator mode simulation) result in data concurrency issues. However, you can avoid such issues by adding several lines of MATLAB code. You must first create a temporary folder within the `parfor` loop and then add 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.

Following are examples that use this method to resolve common concurrency issues.

A Model with Stateflow, Embedded MATLAB, or Model Blocks

In this example, either the model (mdl) is configured to simulate in Accelerator mode or it contains a Stateflow®, an Embedded MATLAB™, or a Simulink 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(mdl)
    % set the block parameters like the filename of the ToFile block
    set_param(someBlkInMdl, blkParamName, blkParamValue{i})
    % set the model parameters by passing them to the sim command
    out{i} = sim(mdl, 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 ToFile Blocks

If you simulate a model with ToFile 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 ToFile blocks. By providing a unique suffix for each iteration of the `parfor` body, you can avoid the concurrency issue.

```
parfor i=1:4
    sim mdl, ...
        '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 “Error Handling”. 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 demonstrates 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:SL_BlzParamUndefined'
    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 *vdz*, MATLAB would respond to the `getfullname` command with:

```
ans =
```

vdp/Mu

Improving Simulation Performance and Accuracy

- “About Improving Performance and Accuracy” on page 23-2
- “Speeding Up the Simulation” on page 23-2
- “Comparing Performance” on page 23-4
- “Improving Acceleration Mode Performance” on page 23-8
- “Improving Simulation Accuracy” on page 23-10

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.

Speeding Up the Simulation

Slow simulation speed can have many causes. Here are a few:

- Your model includes a MATLAB Fcn block. When a model includes a MATLAB Fcn block, the MATLAB interpreter is called at each time step, drastically slowing down the simulation. Use the built-in Fcn block or Math Function block whenever possible.
- Your model includes an M-file S-function. M-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” in the online documentation.
- 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 2-35.
- 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 “How Scope Viewer Parameter Settings Can Affect Performance” on page 24-10.

Comparing Performance

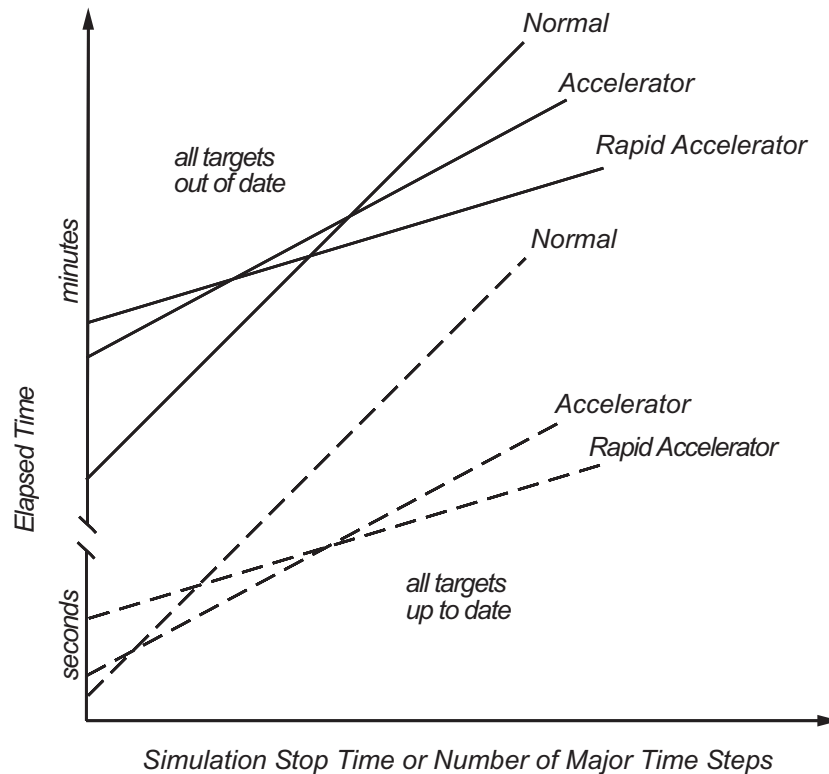
In this section...
“Performance of the Simulation Modes” on page 23-4
“Measuring Performance” on page 23-6

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 Embedded MATLAB 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 “Selecting Blocks for Accelerator Mode” on page 27-14) might not run much faster than they would in Normal mode unless the simulation stop time is very large. Similarly, models with a large Stateflow or Embedded MATLAB content 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 or Embedded MATLAB 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.

Measuring Performance

You can use the `tic`, `toc`, and `sim` commands to compare Accelerator mode or Rapid Accelerator mode execution times to Normal mode.

- 1 Use `load_system` to load your model into memory without opening a window.
- 2 From the **Simulation** 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** menu, and build an executable for the model by clicking the **Start** 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 27-7 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.

Improving Acceleration Mode Performance

In this section...
“Techniques” on page 23-8
“C Compilers” on page 23-9

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 Real-Time Workshop software to create C code for the S-function). See “Controlling S-Function Execution” on page 27-15 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 “Integrating External Code With Generated C and C++ 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. See “Customizing the Build Process” on page 27-20 for details.

C Compilers

On computers running the Microsoft Windows operating system, the Accelerator and Rapid Accelerator modes use the default 32-bit C compiler supplied by The 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, or if you wish to use a 64-bit compiler.

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/

Improving 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 3-31).
- A Derivative block found in an algebraic loop can result in a loss in solver accuracy.

Visualizing Simulation Results

- “About Scope Blocks, Viewers, Signal Logging, and Test Points” on page 24-2
- “Methods for Attaching a Generator or Viewer” on page 24-6
- “Displaying a Scope Viewer” on page 24-7
- “Things to Know When Using Viewers” on page 24-9
- “Changing Viewer Characteristics” on page 24-12
- “Scope Viewer Context Menu” on page 24-17
- “Performing Common Viewer Tasks” on page 24-18
- “Performing Common Generator Tasks” on page 24-24

About Scope Blocks, Viewers, Signal Logging, and Test Points

In this section...

“What Are Scope Blocks, Signal Viewers, Test Points and Data Logging?” on page 24-2

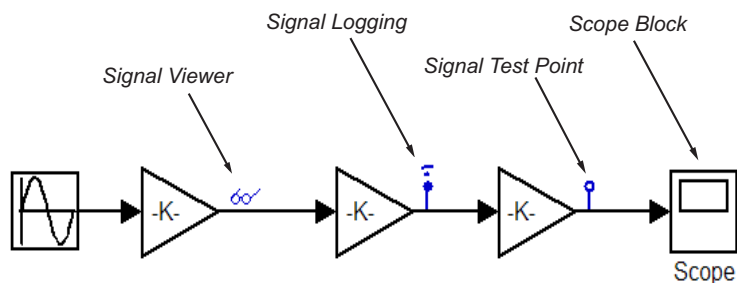
“How Scope Blocks and Signal Viewers Differ” on page 24-3

“Why Use Generators and Signal Viewers Instead of Source and Scope Blocks?” on page 24-4

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 learn how to quickly perform basic signal viewer tasks, see “Performing Common Viewer Tasks” on page 24-18.
- For detailed information on signal viewers, see “Introducing the Signal and Scope Manager” on page 10-37.
- To learn how to add and change signal viewers, see “Using the Signal and Scope Manager” on page 10-43.
- For more information on signal logging, see “Logging Signals” on page 15-3.

- For more information on Signal Test Points, see “Working with Test Points” on page 10-61.
- For more information on Scope Blocks, see “Sinks”.

How Scope Blocks and Signal Viewers Differ

You use Scope Blocks and signal viewers to display simulation results, but as shown in this table, their characteristics differ:

Characteristic	Signal Viewer	Scope Block
Interface	Attach to signal using Signal Selector or context menu See “The Signal Selector” on page 10-48 See “Scope Viewer Context Menu” on page 24-17	Drag from Library Browser
Scope of Control	All viewers centrally managed from Signal and Scope Manager See “Introducing the Signal and Scope Manager” on page 10-37	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 “Displaying Multiple Axes” on page 24-20	Multiple
Data handling	Save data to a signal logging object	Save variable data to workspace as structures or arrays

Characteristic	Signal Viewer	Scope Block
Data logging	Log data to model-wide data object See Simulink.ModelDataLogs	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.	The scope displays minor steps if the Refine parameter is greater than 1. The value of the Refine parameter indicates the number of intermediate steps displayed.

Why Use Generators and Signal Viewers Instead of Source and Scope Blocks?

You should use signal generators and viewers instead of Source and Scope blocks when:

- You want to navigate to and attach generators or viewers deep within a model hierarchy.
- You want to centrally manage all generators and viewers present in your model.

- You want to use the display features provided by signal viewers that are not available in Scope blocks.
- You want to reduce clutter in your block diagram. Because signal viewers attach directly to signals, it is not necessary to route them to a Scope block. This results in fewer signal routes in your block diagram.
- You want to easily view data from referenced models. See “Working with Test Points” on page 10-61 for more information.

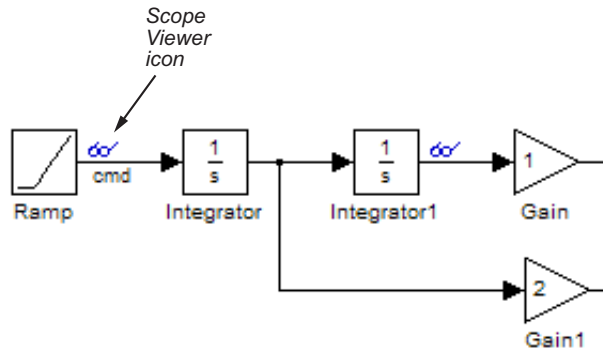
Methods for Attaching a Generator or Viewer

Generators and viewers attach to signals in your model in one of two ways:

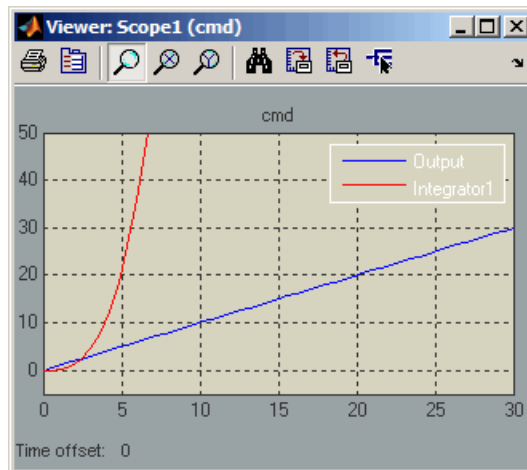
When...	Use...
You want to review all of the scopes and viewers, and the signals connected to them	Signal and Scope Manager
You want to quickly connect and disconnect a viewer or generator	Signal Context Menu

Displaying a Scope Viewer

Click the Scope Viewer icon to display a particular Scope Viewer:



Two plots are displayed in this viewer example. Each is identified with a unique color, and the graph has a legend.



- To learn how to add a legend and to zoom into regions in your graph, see “Performing Common Viewer Tasks” on page 24-18.
- To learn how a viewer can display multiple signals, see “Adding Multiple Signals to a Scope Viewer” on page 24-19.

- To learn about the Scope Viewer controls, see “Changing Viewer Characteristics” on page 24-12.
- To learn how to attach a Scope Viewer to a signal, see “Attaching a Scope Viewer” on page 24-18.

Tip You must first attach a viewer for the Scope Viewer icon to be visible.

Things to Know When Using Viewers

In this section...

“About Viewers” on page 24-9

“How the Viewer Determines Trace Color Coding and Line Styles” on page 24-9

“How Scope Viewer Parameter Settings Can Affect Performance” on page 24-10

About Viewers

- The Scope Viewer is not the same as the Scope block. For an explanation of the differences, see “How Scope Blocks and Signal Viewers Differ” on page 24-3.
- The Scope Viewer does not show the signal label on the axis.
- The Scope Viewer does not work with the Report Generator.
- The Scope Viewer does not display the simulation minor time step values.
- Not all of the Scope Viewer features are supported when you simulate your model in Rapid Accelerator mode. For more information, see “Using Scopes and Viewers with Rapid Accelerator Mode” on page 27-16.
- The **Help** button for the Scope Viewer is located in the Scope Viewer’s Parameters dialog box. For more information, see “Scope Viewer Parameters Dialog Box” on page 24-13.

How the Viewer Determines Trace Color Coding and Line Styles

The Scope Viewer displays each signal as a separate, color-coded trace, in the following order:

- 1 Blue
- 2 Red
- 3 Magenta

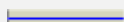
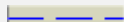


4 Cyan

5 Yellow

6 Green

The viewer cycles through the colors if the axis is displaying more than six signals.

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	

How Scope Viewer Parameter Settings Can Affect Performance

In some cases, when a Scope Viewer needs to display a large number of data points, the simulation slows. 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. See “**Scroll**” on page 24-14.
- Reduce the time range. See “**Time Range**” on page 24-14.

- Use decimation to reduce the number of data points. See “**Decimation**” on page 24-15.
- Increase the refresh period to decrease the refresh rate. See “**Refresh Period**” on page 24-16.
- Limit the number of data points that the viewer saves to the workspace. See “**Limit data points to last**” on page 24-15.

Changing Viewer Characteristics







In this section...





“The Scope Viewer Toolbar” on page 24-12

“Scope Viewer Parameters Dialog Box” on page 24-13


The Scope Viewer Toolbar

The Scope Viewer toolbar is attached to each Scope 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 Viewer Parameters Dialog Box” on page 24-13.
	Simultaneously zooms in on the x and y axes. The zoom feature is not active while the simulation is running. For more information, see “Zooming In On Graph Regions” on page 24-19.
	Use this button to zoom in on the x axis only. The zoom feature is not active while the simulation is running. For more information, see “Zooming In On Graph Regions” on page 24-19.
	Use this button to zoom in on the y axis only. The zoom feature is not active while the simulation is running. For more information, see “Zooming In On Graph Regions” on page 24-19.
	Automatically scales the axis to fully display all signals.

Icon	Function
	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.
	Activates the Signal Selector. For more information, see “The Signal Selector” on page 10-48.
	Docks and undocks the Scope Viewer. When you dock the Scope Viewer, it is placed within the MATLAB Command Window and automatically resized.

Scope Viewer Parameters Dialog Box

Open the Scope Parameters dialog box by clicking  on the scope toolbar, or by selecting **Scope parameters** from the context menu. There are three tabs:

- **General**, where you set the axis characteristics and the sampling decimation value (see “General Tab” on page 24-13).
- **History**, where you control the amount of stored and displayed data (see “History Tab” on page 24-15).
- **Performance**, where you control the scope refresh rate (see “Performance Tab” on page 24-16).

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 Multiple Signals to a Scope Viewer” on page 24-19.

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 “How Scope Viewer Parameter Settings Can Affect Performance” on page 24-10.

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 your model uses a fixed-step solver with a step size 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 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 “Logging Signals” on page 15-3 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’s **Configuration Parameters** dialog box.

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 Viewer is refreshed. Reducing the refresh rate can speed up the simulation in some cases.

Note For information about additional Scope Viewer parameters that can affect performance, see “How Scope Viewer Parameter Settings Can Affect Performance” on page 24-10.

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.

Scope Viewer Context Menu

The Scope Viewer context menu is a convenient way to make simple changes to a Scope Viewer without navigating to the Scope Parameters dialog box.

Right-click within a Scope Viewer to display the context menu. It contains the following controls:

Control	Function
Legends	Adds a legend to your viewer.
Autoscale	Autoscales the viewer axis.
Signal selection	Displays the Signal Selector dialog. For information, see “The Signal Selector” on page 10-48.
Axes properties	Displays the Axis Properties dialog. You can manually set the minimum and maximum range for the <i>y</i> axis here.
Scope parameters	Displays the Scope parameters dialog. For information, see “Scope Viewer Parameters Dialog Box” on page 24-13.
Tick labels	Displays the Tick Labels dialog. From here you can turn on and off various tick options.

Performing Common Viewer Tasks

In this section...

- “Viewing Scope Viewer Help” on page 24-18
- “Attaching a Scope Viewer” on page 24-18
- “Adding Multiple Signals to a Scope Viewer” on page 24-19
- “Adding a Legend” on page 24-19
- “Zooming In On Graph Regions” on page 24-19
- “Displaying Multiple Axes” on page 24-20
- “How to Save Data to MATLAB Workspace” on page 24-22
- “Saving the Viewer Data to a File” on page 24-22
- “Plotting the Viewer Data” on page 24-23

Viewing Scope Viewer Help

The Scope Viewer **Help** button is in the Scope Viewer’s Parameters dialog.

Access the dialog by clicking  on the Scope Viewer toolbar.

Attaching a Scope Viewer

- 1 In your block diagram, right-click a signal, and from the context menu, select **Create & Connect Viewer**.
- 2 From the list that appears, select the type of scope to be attached.

An empty Scope Viewer will be displayed. You must run the simulation after the viewer has been attached for the information to be plotted.

Tip You can also attach signal viewers with the Signal and Scope Manager. To learn how to do this, see “Using the Signal and Scope Manager” on page 10-43

Adding Multiple Signals to a Scope Viewer

To add additional traces to an existing Scope Viewer:

- 1 In your block diagram, right-click the signal to be added.
- 2 From the signal context menu, select **Connect to existing viewer**.
- 3 A list of Signal Viewers that have already been created is displayed. From the list, select the scope to which the new signal will be added.

You can also add signals to Signal Viewers with the Signal and Scope Manager. To learn how to do this, see “Using the Signal and Scope Manager” on page 10-43.

Note The simulation must be run again for the new trace to be displayed in the viewer.

Adding a Legend

To add a legend (a box identifying each signal in a graph):

- 1 Right-click in the viewer to display the context menu.
- 2 Click on **Legends**.

Tip Left-click and drag the legend box to reposition it.

The color and line styles used in the display are described in “How the Viewer Determines Trace Color Coding and Line Styles” on page 24-9.

Zooming In On Graph Regions


- To zoom in on a region (simultaneously zooming in on the x and y




directions), select from the Scope Viewer toolbar, and left-click


within the graph. While holding down the mouse button, define the region of interest.

To zoom, click and hold the left mouse button, and drag the mouse to define the zoom region bounding box. Release the mouse button to effect the change.

- Select  and use the mouse to zoom only in the x direction.

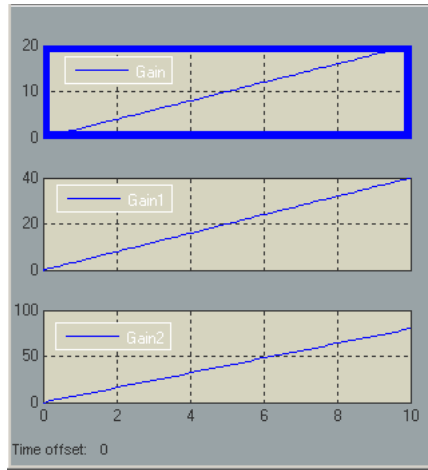
Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the left mouse button, then moving the pointer to the other end of the region. Release the mouse button to effect the change. To zoom, click and hold the left mouse button, and drag the mouse to define the zoom region bounding box. Release the mouse button to effect the change.

- Select  and use the mouse to zoom only the y direction.

Tip Use Autoscale () to restore the display if you mistakenly zoom in too much.

Displaying Multiple Axes

You can add multiple plots (called *axes*) to a Scope Viewer. Each axes can have different *y*-axis settings. This Scope Viewer has three axes, each displaying a separate signal with their own *y* axis settings.



To add axes to an existing Scope Viewer:

- 1 Open the Scope Viewer Signal Properties dialog.

Access this dialog from the Scope Viewer toolbar or by right-clicking within a viewer.

For information on the Scope Viewer toolbar, see “The Scope Viewer Toolbar” on page 24-12.

- 2 In the **Number of axes** field, enter the total number of axes for the graph.
- 3 Click OK to accept the change and dismiss the dialog.
- 4 In your block diagram, right-click on the signal to be added.

The signal context menu appears.

- 5 Select **Connect to existing viewer**.

A list of signal viewers is displayed.


- 6 From the list, select the scope to which the new signal will be added.
- 7 From the list of Axes, select the pane to which the signal will be plotted. The panes are numbered from top to bottom.

Repeat this step until signals for each of the axes have been assigned.

Note Run the simulation again to display the new traces.

How to Save Data to MATLAB Workspace

To save the data displayed on the viewer to the MATLAB workspace, perform the following steps:

- 1 Click on the  in the Viewer toolbar.
- 2 Click on the **History** tab.
- 3 Select **Save to model signal logging object (logout)**.
- 4 Enter the logging variable name in the **Logging name** field.
- 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.

Saving the Viewer Data to a File

You can quickly save the viewer data to a file using the Time Series Tools.

- 1 First select the model **Tools** option **Inspect Logged Signals**. The *Time Series Tools* opens and the file tree contains your data on the lowest-level nodes under “logout”.

- 2 Export the data to a file by following the directions in “Importing and Exporting Data”.

Plotting the Viewer Data

You can plot the data interactively or programmatically.

- **Time Series Tools:** plot the data interactively using the *Time Series Tools* as explained in “Plotting Time Series”.
- **Simplot:** plot the data from the command line using the `simplot` command (see `simplot`). The resulting figure looks identical to the viewer window and can be annotated using the Plot Editing Tools.

Performing Common Generator Tasks

In this section...
“Attaching a Generator” on page 24-24
“Removing a Generator” on page 24-24

Attaching a Generator

- 1 Right-click the input to a block (such as a gain block), and from the signal context menu select **Create & Connect Signal Generator**.
- 2 From the list that appears, select the type of scope to be attached.

The generator will appear in the block diagram as a small rectangle that lists the generator type you have chosen. Double-click this region to display a dialog from where you can change the generators properties.

You can also attach generators with the Signal and Scope Manager. To learn how to do this, see “Using the Signal and Scope Manager” on page 10-43.

Removing a Generator

To remove the generator from the block diagram:

- 1 Right-click a generator.
- 2 From the context menu, select **Disconnect Generator**.

Analyzing Simulation Results

- “Viewing Output Trajectories” on page 25-2
- “Linearizing Models” on page 25-4
- “Finding Steady-State Points” on page 25-10

Viewing Output Trajectories

In this section...

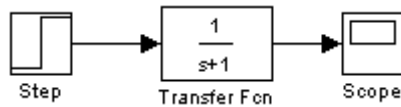
“Using the Scope Block” on page 25-2

“Using Return Variables” on page 25-2

“Using the To Workspace Block” on page 25-3

Using the Scope Block

You can display output trajectories on a Scope block during simulation as illustrated by the following model.

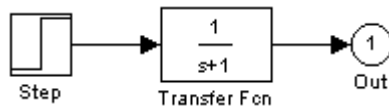


The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

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.



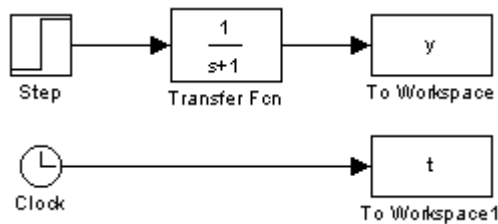
The block labeled Out is an Outport block from the Ports & Subsystems library. The output trajectory, `yout`, is returned by the integration solver. For more information, see “Data Import/Export Pane”.

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.

Linearizing Models

In this section...

“About Linearizing Models” on page 25-4

“Linearization with Referenced Models” on page 25-6

“Linearization Using the ‘v5’ Algorithm” on page 25-8

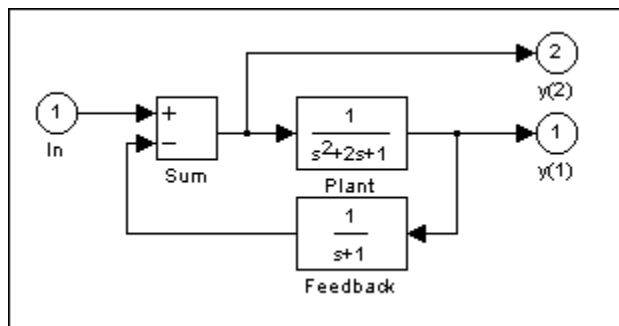
About Linearizing Models

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)
```

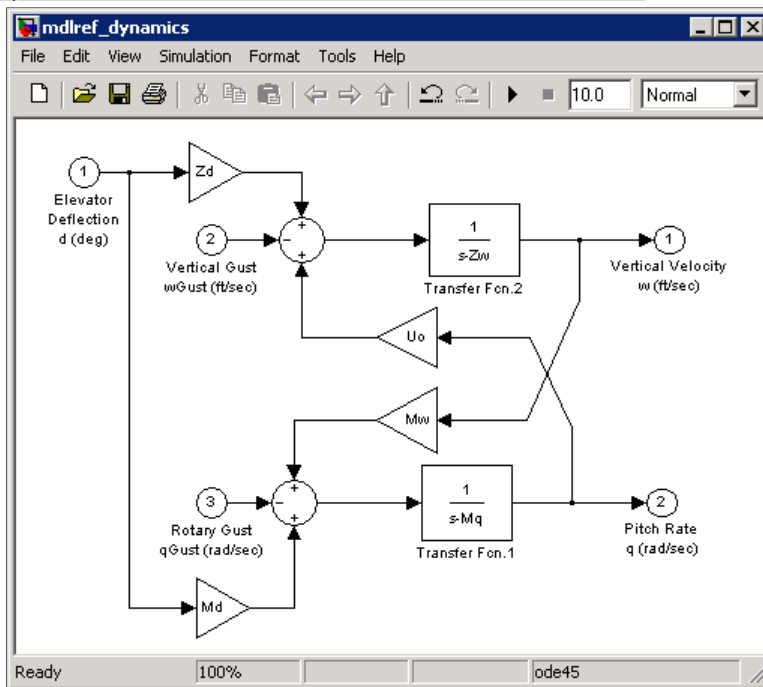
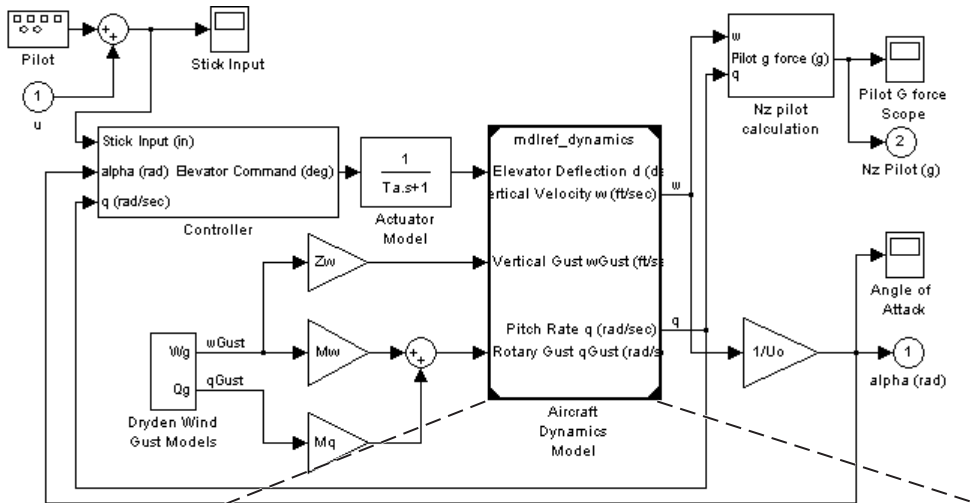
For discrete systems or mixed continuous and discrete systems, use the function `dlinmod` for linearization. This function has the same calling syntax as `linmod` except that the second right-hand argument must contain a sample time at which to perform the linearization.

Linearization with Referenced Models

You can use `linmod` to extract a linear model from a Simulink environment that contains Model blocks.

Note In normal simulation mode, the `linmod` command applies the block-by-block linearization algorithm on blocks inside the referenced model. If the model block is in accelerated 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. See the Control Design documentation for an explanation of the block-by-block linearization algorithm.

For example, consider the f14 model `mdlref_f14.mdl`. The Aircraft Dynamics Model block refers to the model `mdlref_dynamics.mdl`.



To linearize the md1ref_f14 model, call the linmod command on the top md1ref_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.

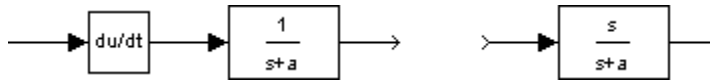
You access the Extras library by opening the Blocksets & Toolboxes icon:

- 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

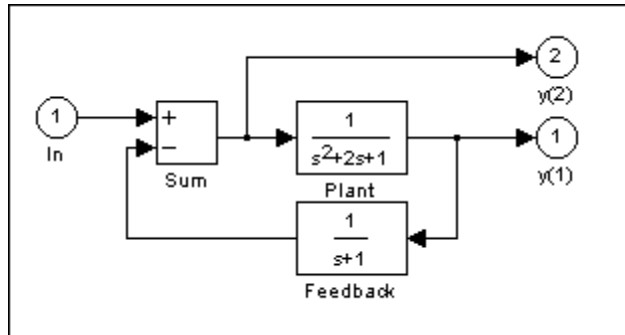
$$\frac{s}{s+a}$$

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 `lmod`.



You can use the `trim` function to find the values of the input and the states that set both outputs to 1. First, make initial guesses for the state variables (x) and input values (u), then set the desired value for the output (y).

```
x = [0; 0; 0];
u = 0;
y = [1; 1];
```

Use index variables to indicate which variables are fixed and which can vary.

```
ix = [];      % Don't fix any of the states
iu = [];      % Don't fix the input
iy = [1;2];   % Fix both output 1 and output 2
```

Invoking `trim` returns the solution. Your results might differ because of roundoff error.

```
[x,u,y,dx] = trim('lmod',x,u,y,ix,iu,iy)
```

```
x =  
    0.0000  
    1.0000  
    1.0000  
u =  
     2  
y =  
    1.0000  
    1.0000  
dx =  
    1.0e-015 *  
   -0.2220  
   -0.0227  
    0.3331
```

Note that there might be no solution to equilibrium point problems. If that is the case, `trim` returns a solution that minimizes the maximum deviation from the desired result after first trying to set the derivatives to zero. For a description of the `trim` syntax, see `trim` in the *Simulink Reference*.

Simulink Debugger

- “Introduction to the Debugger” on page 26-2
- “Using the Debugger’s Graphical User Interface” on page 26-3
- “Using the Debugger’s Command-Line Interface” on page 26-11
- “Getting Online Help” on page 26-13
- “Starting the Debugger” on page 26-14
- “Starting a Simulation” on page 26-15
- “Running a Simulation Step by Step” on page 26-19
- “Setting Breakpoints” on page 26-27
- “Displaying Information About the Simulation” on page 26-33
- “Displaying Information About the Model” on page 26-38

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.

Using the Debugger's Graphical User Interface

In this section...

“Displaying the Graphical Interface” on page 26-3

“Toolbar” on page 26-4

“Breakpoints Pane” on page 26-6

“Simulation Loop Pane” on page 26-7

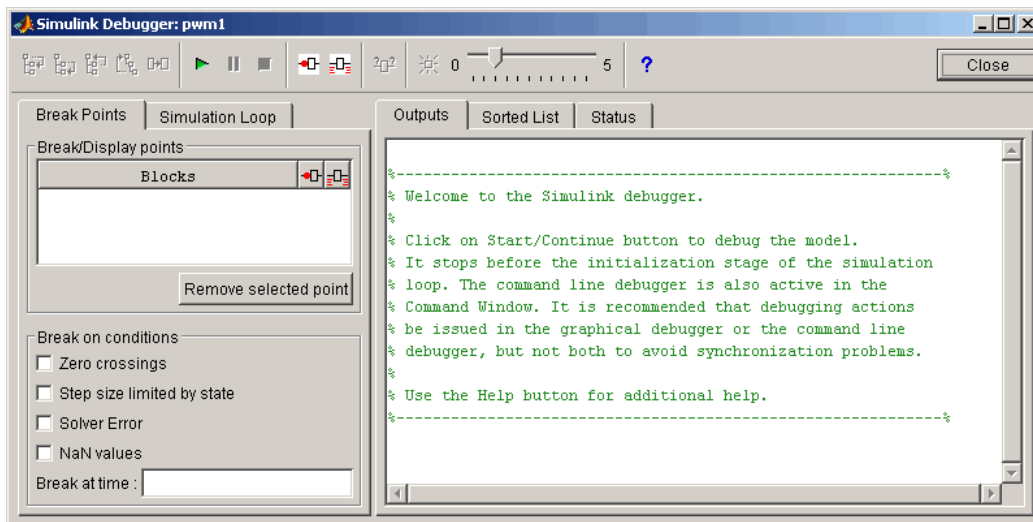
“Outputs Pane” on page 26-8

“Sorted List Pane” on page 26-9

“Status Pane” on page 26-10

Displaying the Graphical Interface

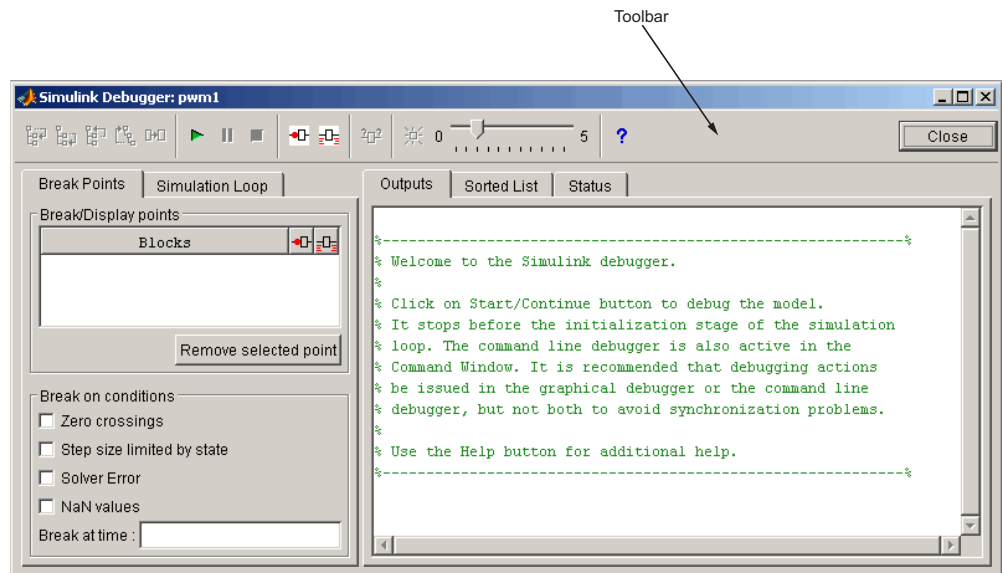
Select **Simulink Debugger** from a model window's **Tools** menu to display the debugger graphical interface.





Note The debugger graphical user interface does not display state or solver information. The command line interface does provide this information. See “Displaying System States” on page 26-36 and “Displaying Solver Information” on page 26-36.












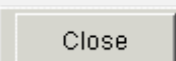
Toolbar

The debugger toolbar appears at the top of the debugger window.



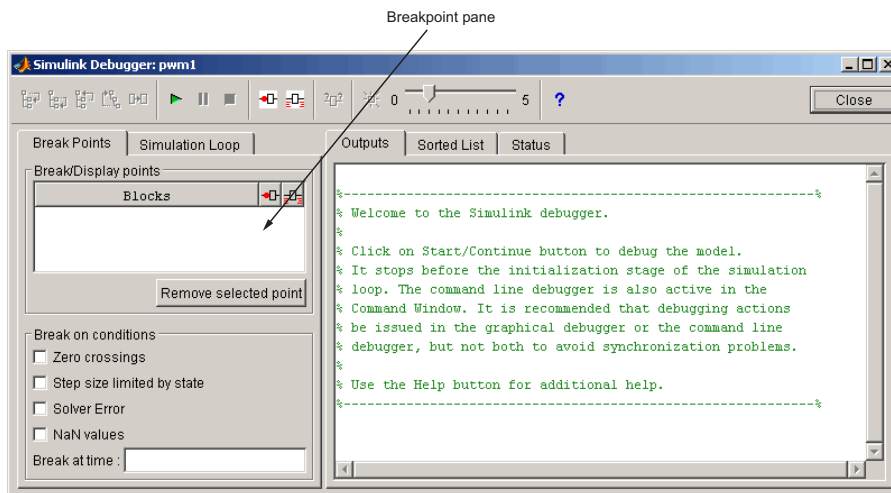
From left to right, the toolbar contains the following command buttons:

Button	Purpose
	Step into the next method (see “Stepping Commands” on page 26-21 for more information on this command, and the following stepping commands).
	Step over the next method.

Button	Purpose
	Step out of the current method.
	Step to the first method at the start of next time step.
	Step to the next block method.
	Start or continue the simulation.
	Pause the simulation.
	Stop the simulation.
	Break before the selected block.
	Display inputs and outputs of the selected block when executed (same as trace gcb).
	Display the current inputs and outputs of selected block (same as probe gcb).
	Toggle animation mode on or off (see “Animation Mode” on page 26-23). The slider next to this button controls the animation rate.
	Display help for the debugger.
	Close the debugger.

Breakpoints Pane

To display the **Breakpoints** pane, select the **Break Points** tab on the debugger window.



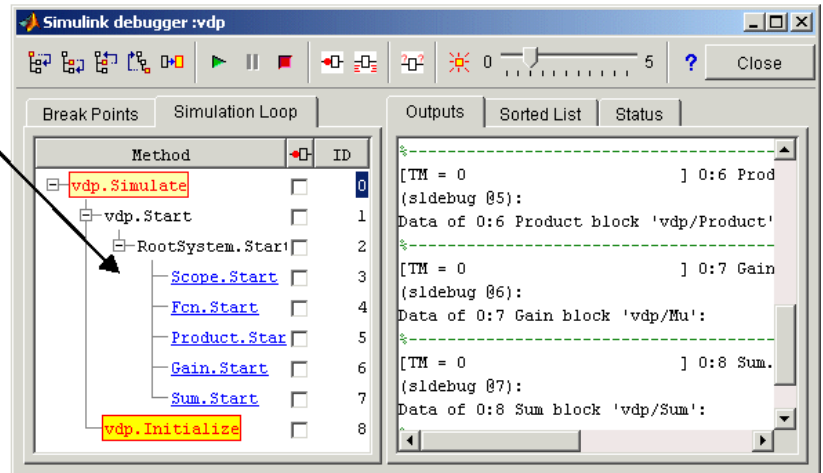
The **Breakpoints** pane allows you to specify block methods or conditions at which to stop a simulation. See “Setting Breakpoints” on page 26-27 for more information.

Note The debugger grays out and disables the **Breakpoints** pane when you select animation mode. (see “Animation Mode” on page 26-23). This prevents you from setting breakpoints and indicates that animation mode ignores existing breakpoints.

Simulation Loop Pane

To display the **Simulation Loop** pane, select the **Simulation Loop** tab on the debugger window.

Simulation Loop pane



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 26-29 for more information.

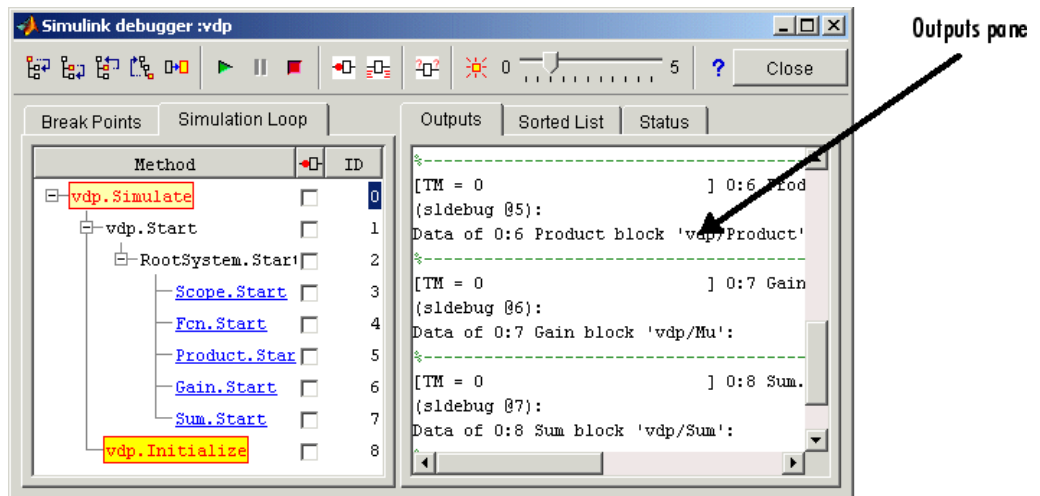
Note This column is disabled when its animation mode is selected (see “Animation Mode” on page 26-23), because in animation mode you can’t set breakpoints.

ID Column

The ID column lists the IDs of the methods listed in the **Methods** column. See “Method ID” on page 26-11 for more information.

Outputs Pane

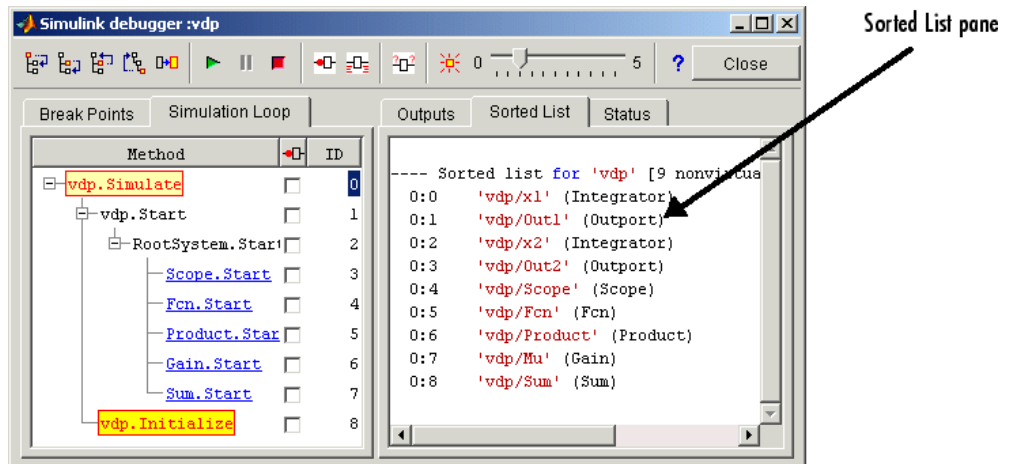
To display the **Outputs** pane, select the **Outputs** tab on the debugger window.



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 26-20). 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 26-11).

Sorted List Pane

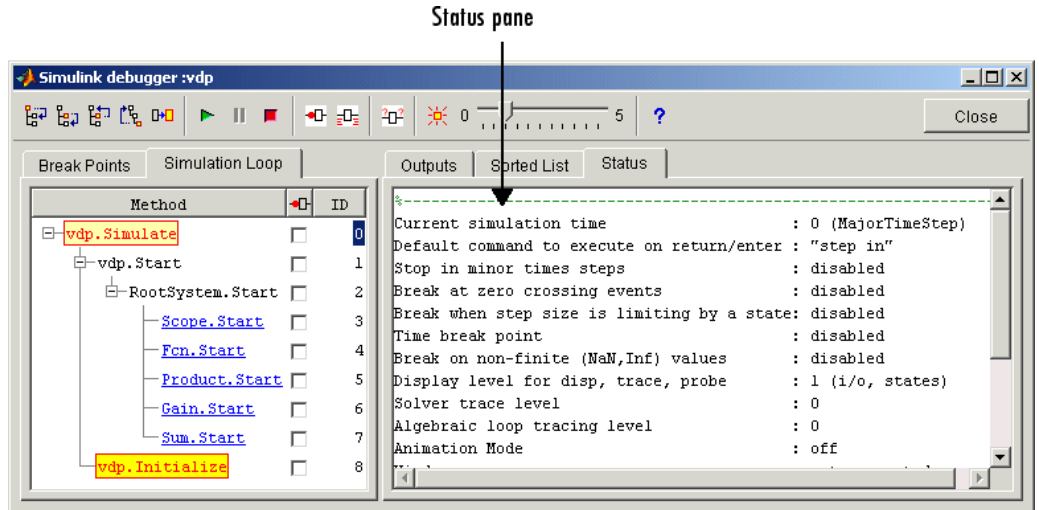
To display the **Sorted List** pane, select the **Sorted List** tab on the debugger window.



The **Sorted List** pane displays the sorted lists for the model being debugged. See “Displaying a Models Sorted Lists” on page 26-38 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.

Using the Debugger's Command-Line Interface

In this section...

“Controlling the Debugger” on page 26-11

“Method ID” on page 26-11

“Block ID” on page 26-11

“Accessing the MATLAB Workspace” on page 26-12

Controlling the Debugger

In command-line mode, you control the debugger by entering commands at the debugger command line in the MATLAB Command Window. The debugger accepts abbreviations for debugger commands. See “Simulink Debugger Commands” for a list of command abbreviations and repeatable commands.

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 `sid:bid`, where `sid` is an integer identifying the system that contains the block (either the root system or a nonvirtual subsystem) and `bid` is the position of the block in the system's sorted list. For example, the block index `0:1` refers to the first block in the model's root system. The `slist` command shows the block ID for each debugged block in the model.

Accessing the MATLAB Workspace

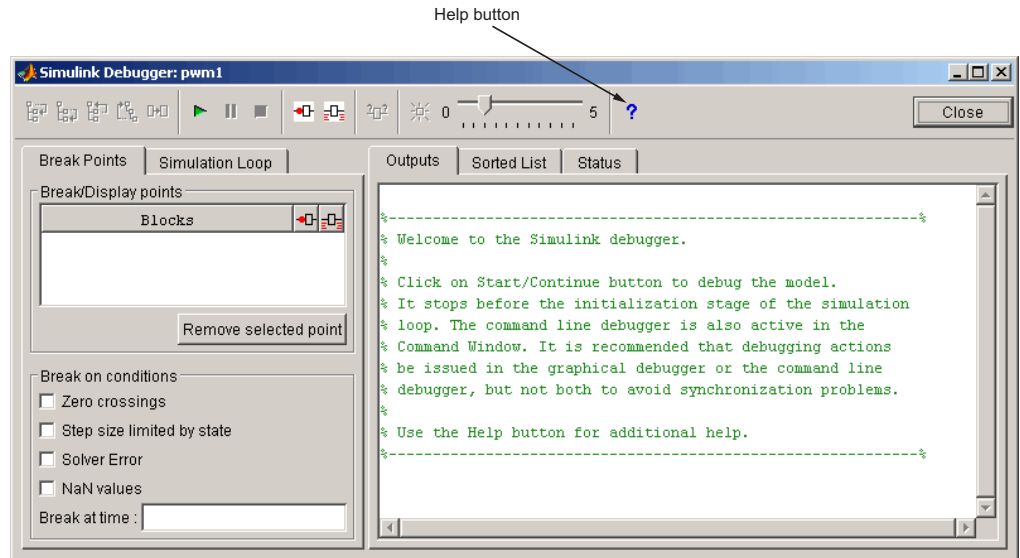
You can enter any MATLAB expression at the `sldebug` prompt. For example, suppose you are at a breakpoint and you are logging time and output of your model as `tout` and `yout`. The following command creates a plot.

```
(sldebug ...) plot(tout, yout)
```

You cannot display the value of a workspace variable whose name is partially or entirely the same as that of a debugger command by entering it at the debugger command prompt. You can, however, use the `eval` command to work around this problem. For example, use `eval('s')` to determine the value of `s` rather than `s(tep)` the simulation.

Getting 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.

Starting the Debugger

You can start the debugger either from a model window or from the command line. To start the debugger from a model window, select **Simulink Debugger** from the model window **Tools** menu. The debugger graphical user interface appears (see “Using the Debugger’s Graphical User Interface” on page 26-3).

To start the debugger from the MATLAB Command Window, enter either a `sim` command or the `sldebug` command. For example the following commands load the demo model `vdp` into memory, starts the simulation, and stops the simulation at the first block in the model’s execution list

```
sim('vdp',[0,10],simset('debug','on'))
```

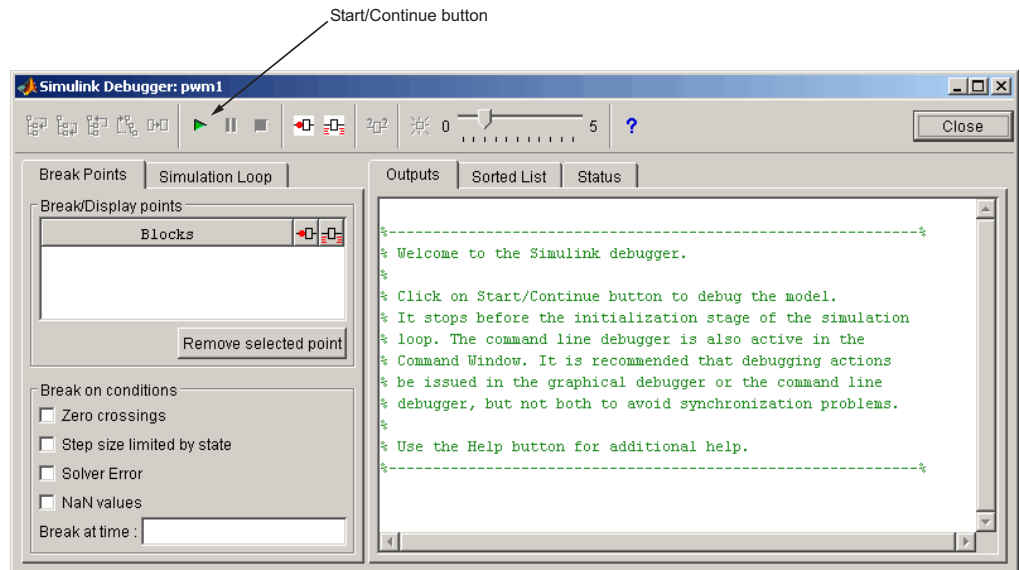
or

```
sldebug 'vdp'
```

Note When running the debugger in graphical user interface (GUI) mode, you must explicitly start the simulation. See “Starting a Simulation” on page 26-15 for more information.

Starting a Simulation

To start the simulation, click the **Start/Continue** button on the debugger toolbar.

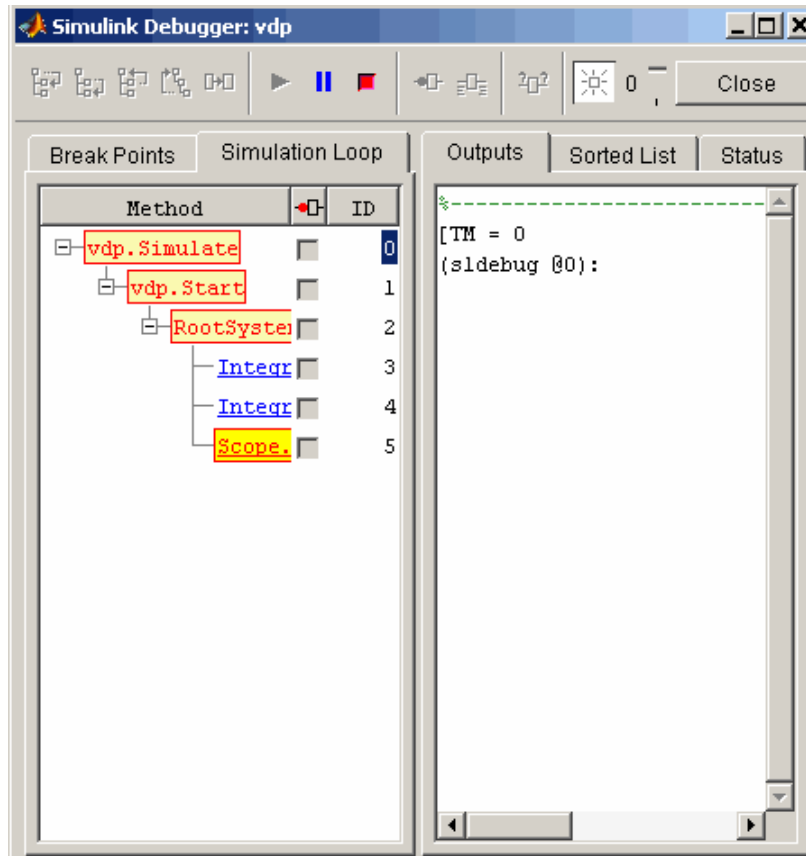


The simulation starts and stops at the first simulation method that is to be executed. It displays the name of the method in its **Simulation Loop** pane and in the debug pointer on the block diagram. The debug pointer indicates on the block diagram which block method is being executed at each step. At this point, you can

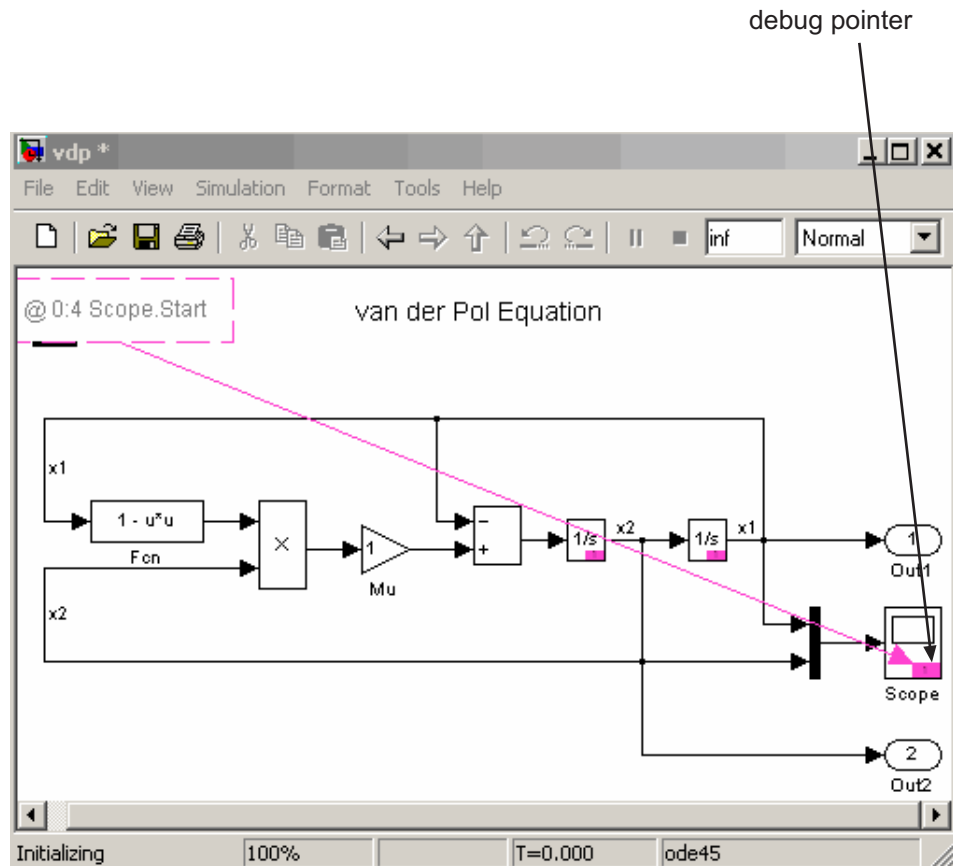
- Set breakpoints.
- Run the simulation step by step.
- Continue the simulation to the next breakpoint or end.
- Examine data.
- Perform other debugging tasks.

As the simulation progresses, the block diagram updates with debug pointers.

The debugger displays the name of the method in the Simulation Loop pane, as shown in the following figure:



The debugger also displays a graphical debug pointer (see “Debug Pointer” on page 26-25) in the block diagram of the model that you are debugging. The debug pointer points to the first block method to be executed.



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 Command Window of the MATLAB product. However, to prevent synchronization errors between the graphical and command-line interfaces you should avoid using the command-line interface.

Running a Simulation Step by Step

In this section...

“Introduction” on page 26-19
“Block Data Output” on page 26-20
“Stepping Commands” on page 26-21
“Continuing a Simulation” on page 26-22
“Running a Simulation Nonstop” on page 26-24
“Debug Pointer” on page 26-25

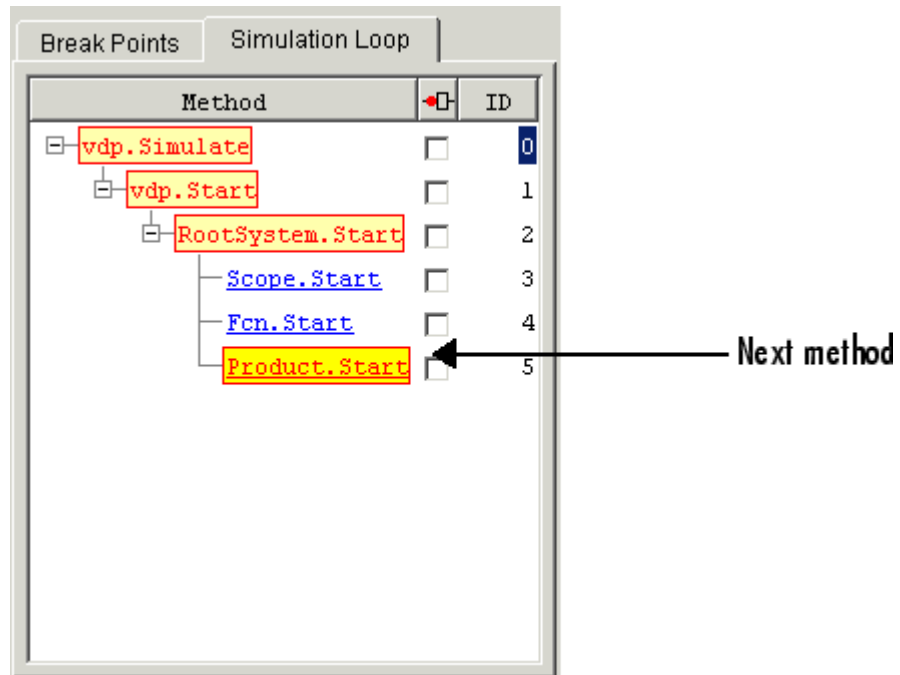
Introduction

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 26-21). 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. If the next method is a block method, the debugger points the debug pointer at the block corresponding to the method (see “Debug Pointer” on page 26-25 for more information). If the block of the next method to be executed resides in a subsystem, the debugger opens the subsystem and points to the block in the subsystem block diagram.

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

where v is the value of the block's discrete state vector.

The debugger also displays the current time, the ID and name of the next method to be executed, and the name of the block to which the method applies in the MATLAB Command Window. The following example illustrates typical debugger outputs after a step command.

```

          Current time      Next method
          ↙                ↘
%-----%
[Tm = 2.009509145207664e-005 ] 0:2 Integrator.Outputs 'vdp/x2'
(sldebug @44):
Data of 0:2 Integrator block 'vdp/x2':
U1      = [-2]
Y1      = [-4.0190182904153282e-005]
CSTATE  = [-4.0190182904153282e-005]
%-----%
[Tm = 2.009509145207664e-005 ] 0:3 Outputport.Outputs 'vdp/Out2'
```

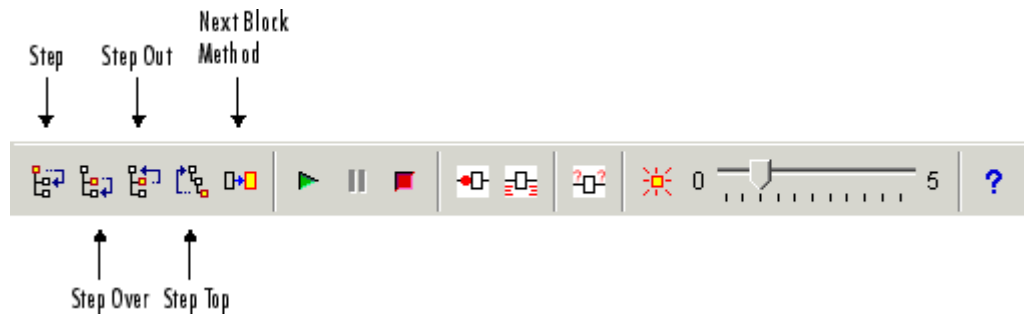
Stepping Commands

Command-line mode provides the following commands for advancing a simulation incrementally:

This command...	Advances the simulation...
step [in into]	Into the next method, stopping at the first method in the next method or, if the next method does not contain any methods, at the end of the next method
step over	To the method that follows the next method, executing all methods invoked directly or indirectly by the next method

This command...	Advances the simulation...
step out	To the end of the current method, executing any remaining methods invoked by the current method
step top	To the first method of the next time step (i.e., the top of the simulation loop)
step blockmth	To the next block method to be executed, executing all intervening model- and system-level methods
next	Same as step over

Buttons in the debugger toolbar allow you to access these commands in GUI mode.



Clicking a button has the same effect as entering the corresponding command at the debugger command line.

Continuing a Simulation

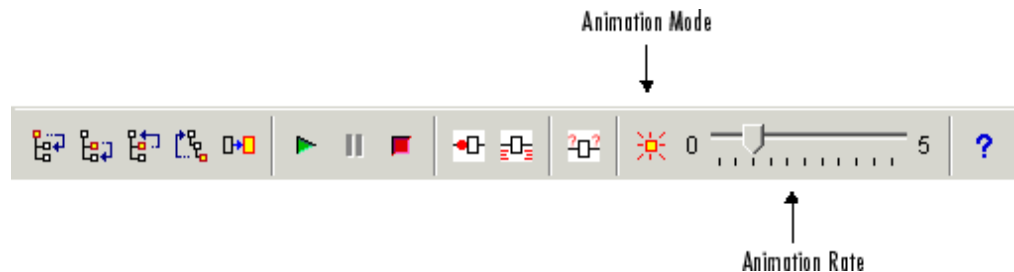
In GUI mode, the **Stop** button turns red when the debugger suspends the simulation for any reason. To continue the simulation, click the **Start/Continue** button. In command-line mode, enter `continue` to continue the simulation. By default, the debugger runs the simulation to the next breakpoint (see “Setting Breakpoints” on page 26-27) or to the end of the simulation, whichever comes first.

Animation Mode

In *animation mode*, the **Start/Continue** button or the `continue` command advances the simulation method by method, pausing after each method, to the first method of the next major time step. While running the simulation in animation mode, the debugger uses its debug pointer (see “Debug Pointer” on page 26-25) to indicate on the block diagram which block method is being executed at each step. The moving pointer shows the simulation progress.

Note In animation mode, the debugger does not allow you to set breakpoints and ignores any breakpoints that you set when animating the simulation.

To enable animation when running the debugger in GUI mode, click the **Animation Mode** button on the debugger toolbar.



Use the slider on the debugger toolbar to increase or decrease the delay between method invocations, and so slow down or speed up the animation rate. To disable animation mode when running the debugger in GUI mode, toggle the **Animation Mode** button on the toolbar.

To enable animation when running the debugger in command-line mode, enter the `animate` command at the command line. The `animate` command has an optional delay parameter for you to specify the length of the pause between method invocations (1 second by default), and thereby accelerate or slow down the animation. For example, the command

```
animate 0.5
```

causes the animation to run at twice its default rate. To disable animation mode when running the debugger in command-line mode, at the command line enter:

```
animate stop
```

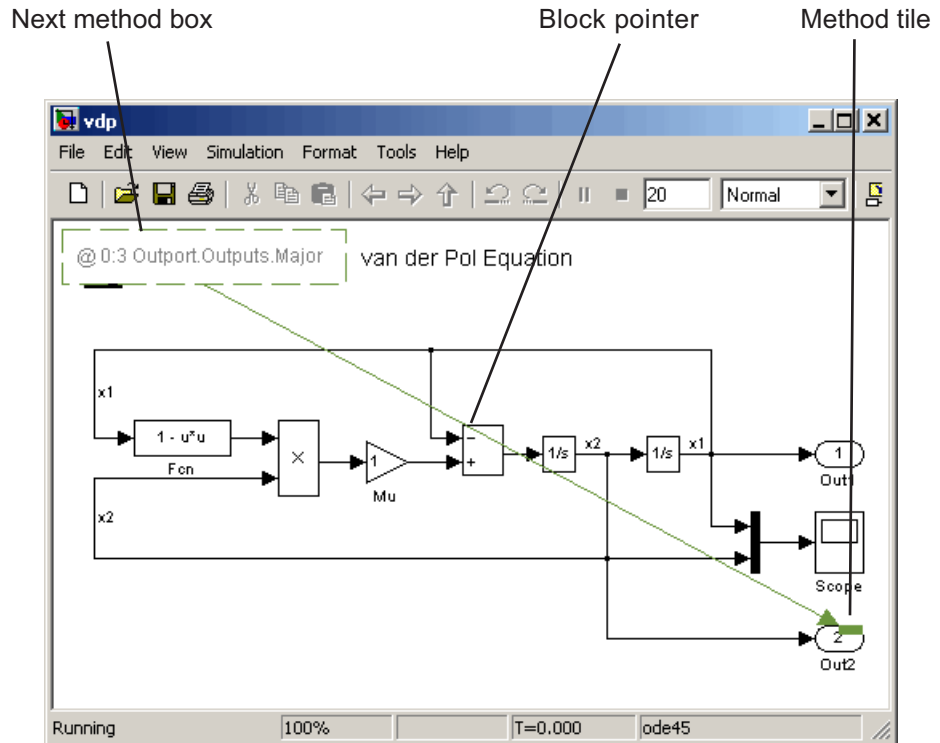
Running a Simulation Nonstop

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.

Debug Pointer

The debugger displays a debug pointer on the block diagram whenever it stops the simulation at a method.



The debug pointer is an annotation that indicates the next method to be executed when simulation resumes. It consists of the following elements:

- Next method box
- Block pointer
- Method tile

Next Method Box

The next method box appears in the upper-left corner of the block diagram. It specifies the name and ID of the next method to be executed.

Block Pointer

The block pointer appears when the next method is a block method. It indicates the block on which the next method operates.

Method Tile

The method tile is a rectangular patch of color that appears when the next method is a block method. The tile overlays a portion of the block on which the next method executes. The color and position of the tile on the block indicate the type of the next block method as follows.

Update (red)	Outputs Major Time Step (dark green)
Derivatives (orange)	Outputs Minor Time Step (green)
Zero Crossings (light blue)	Start (magenta) Initialize (blue) etc.

In animation mode, the tiles persist for the length of the current major time step and a number appears in each tile. The number specifies the number of times that the corresponding method has been invoked for the block thus far in the time step.

Setting Breakpoints

In this section...

“About Breakpoints” on page 26-27

“Setting Unconditional Breakpoints” on page 26-27

“Setting Conditional Breakpoints” on page 26-30

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.

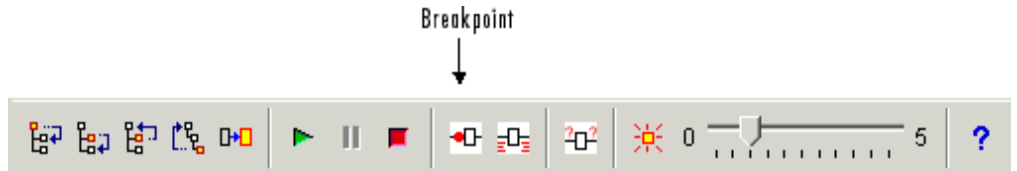
Setting Unconditional Breakpoints

You can set unconditional breakpoints from the:

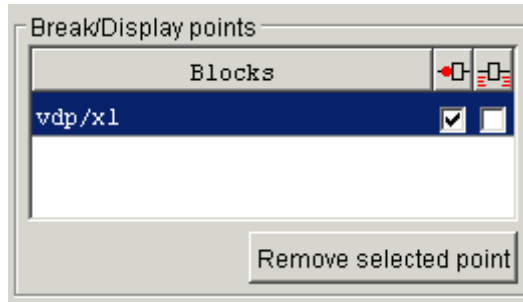
- Debugger toolbar
- **Simulation Loop** pane
- MATLAB product Command Window (command-line mode only)

Setting Breakpoints from the Debugger Toolbar

To set a breakpoint on a block’s methods, select the block and then click the **Breakpoint** button on the debugger toolbar. If you set a break point on a block, the debugger stops at any method that the execution reaches in the block.



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.

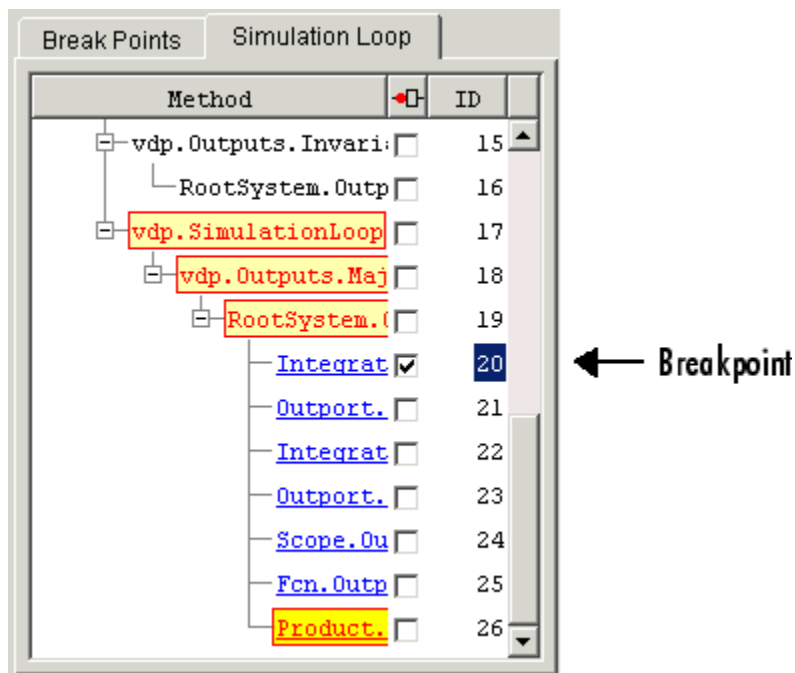
You can temporarily disable the breakpoints on a block by deselecting the check box in the breakpoints column of the panel. To clear the breakpoints on a block and remove its entry from the panel,

- 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 26-40).

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.



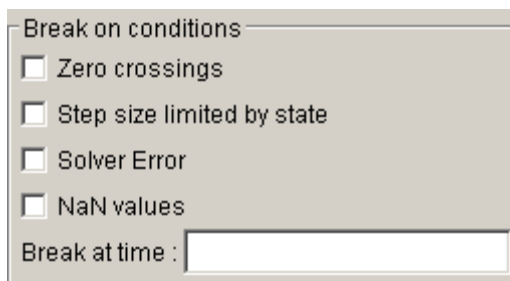
To clear the breakpoint, deselect the check box.

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 = 2.078784598291364      ] vdp.Outputs.Major
(sldebug @18):
```

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 26-11).

For example, setting a zero-crossing break at the start of execution of the zeroxing demo 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

```
2 Zero crossings detected at the following locations
 6 0:5:1 Saturate 'zeroxing/Saturation'
 7 0:5:2 Saturate 'zeroxing/Saturation'
ZeroCrossing Events detected. Interrupting model execution
%-----%
[TM = 0.4                               ] zeroxing.zc.SearchLoop
(sldebug @55): >>
```

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.

Displaying Information About the Simulation

In this section...

“Displaying Block I/O” on page 26-33

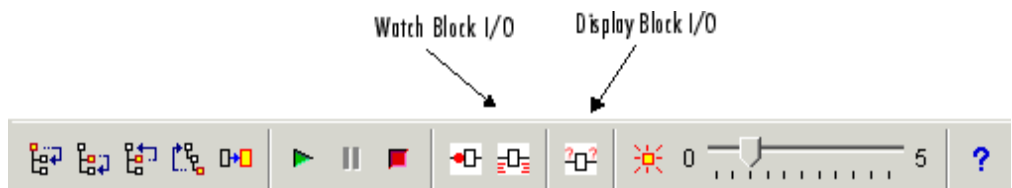
“Displaying Algebraic Loop Information” on page 26-35

“Displaying System States” on page 26-36

“Displaying Solver Information” on page 26-36

Displaying 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. In probe mode, the debugger displays the current inputs and outputs of any block that you select in the model's block diagram. 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 either by entering its block index or by selecting it in the block diagram 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, either select the block in the model diagram and enter undisp gcb or simply 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.

Displaying Algebraic Loop Information

The `atrace` command causes the debugger to display information about a model's algebraic loops (see "Algebraic Loops" on page 2-35) 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

This command...	Displays for each algebraic loop...
atrace 1	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
atrace 2	Same as level 1
atrace 3	Level 2 plus the Jacobian matrix used to solve the loop
atrace 4	Level 3 plus intermediate solutions of the loop variable

Displaying System States

The states debug command lists the current values of the system's states in the MATLAB product Command Window. For example, the following sequence of commands shows the states of the bouncing ball demo (sldemo_bounce) after its first and second time steps.

```
sdebug sldemo_bounce
[Tm=0                ] **Start** of system 'sldemo_bounce' outputs
(sdebug @0:0 'sldemo_bounce/Position'): states
Continuous state vector (value,index,name):
    10                0 (0:0 'sldemo_bounce/Position')
    15                1 (0:5 'sldemo_bounce/Velocity')
(sdebug @0:0 'sldemo_bounce/Position'): next
[Tm=0.01            ] **Start** of system 'bounce' outputs
(sdebug @0:0 'sldemo_bounce/Position'): states
Continuous state vector (value,index,name):
 10.1495095          0 (0:0 'sldemo_bounce/Position')
 14.9019             1 (0:5 'sldemo_bounce/Velocity')
```

Displaying 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.

Displaying Information About the Model

In this section...

“Displaying a Models Sorted Lists” on page 26-38

“Displaying a Block” on page 26-39

Displaying a Models Sorted Lists

In GUI mode, the debugger **Sorted List** pane displays lists of blocks for a models root system and each nonvirtual subsystem. Each list lists the blocks that the subsystems contains sorted according to their computational dependencies, alphabetical order, and other block sorting rules. In command-line mode, you can use the `slist` command to display a model's sorted lists.

```

---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
0:0   'vdp/Integrator1' (Integrator)
0:1   'vdp/Out1' (Outputport)
0:2   'vdp/Integrator2' (Integrator)
0:3   'vdp/Out2' (Outputport)
0:4   'vdp/Fcn' (Fcn)
0:5   'vdp/Product' (Product)
0:6   'vdp/Mu' (Gain)
0:7   'vdp/Scope' (Scope)
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 models blocks. Some debugger commands accept block IDs as arguments.

Identifying Blocks in Algebraic Loops

If a block belongs to an algebraic list, the `slist` command displays an algebraic loop identifier in the entry for the block in the sorted list. The identifier has the form

```
algId=s#n
```

where *s* is the index of the subsystem containing the algebraic loop and *n* is the index of the algebraic loop in the subsystem. For example, the following entry for an Integrator block indicates that it participates in the first algebraic loop at the root level of the model.

```
0:1 'test/ss/I1' (Integrator, tid=0) [algId=0#1, discontinuity]
```

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 26-41 for more information.

Displaying 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 Models Nonvirtual Systems

The `systems` command displays a list of the nonvirtual systems in the model that you are debugging. For example, the clutch demo (`clutch`) contains the following systems:

```
sdebug clutch
[Tm=0                ] **Start** of system 'clutch' outputs
(sdebug @0:0 'clutch/Clutch Pedal'): systems
0  'clutch'
1  'clutch/Locked'
2  '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) demo model.

```
sldebug vdp
[Tm=0                               ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): slist
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
0:0   'vdp/Integrator1' (Integrator)
0:1   'vdp/Out1' (Outputport)
0:2   'vdp/Integrator2' (Integrator)
0:3   'vdp/Out2' (Outputport)
0:4   'vdp/Fcn' (Fcn)
0:5   'vdp/Product' (Product)
0:6   'vdp/Mu' (Gain)
0:7   'vdp/Scope' (Scope)
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 @0:0 'clutch/Clutch Pedal'): zclist
2:3   'clutch/Unlocked/Sign' (Signum)
0:4   'clutch/Lockup Detection/Velocities Match' (HitCross)
0:10  'clutch/Lockup Detection/Required Friction
      for Lockup/Abs' (Abs)
0:11  'clutch/Lockup Detection/Required Friction for
      Lockup/ Relational Operator' (RelationalOperator)
```

```

0:18 'clutch/Break Apart Detection/Abs' (Abs)
0:20 'clutch/Break Apart Detection/Relational Operator'
      (RelationalOperator)
0:24 'clutch/Unlocked' (SubSystem)
0:27 'clutch/Locked' (SubSystem)

```

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 26-38) 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',[0,10],simset('debug','on'))
[Tm=0 ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): status
  Current simulation time: 0 (MajorTimeStep)
  Last command: ""
  Stop in minor times steps is disabled.
  Break at zero crossing events is disabled.
  Break when step size is limiting by a state is disabled.
  Break on non-finite (NaN,Inf) values is disabled.
  Display of integration information is disabled.
  Algebraic loop tracing level is at 0.

```


Accelerating Models

- “What Is Acceleration?” on page 27-2
- “How the Acceleration Modes Work” on page 27-3
- “Code Regeneration in Accelerated Models” on page 27-7
- “Choosing a Simulation Mode” on page 27-10
- “Designing Your Model for Effective Acceleration” on page 27-14
- “Performing Acceleration” on page 27-20
- “Interacting with the Acceleration Modes Programmatically” on page 27-24
- “Using the Accelerator Mode with the Simulink Debugger” on page 27-27
- “Capturing Performance Data” on page 27-29

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 that does not include “Algebraic Loops” on page 2-35, 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 “Using the Accelerator Mode with the Simulink Debugger” on page 27-27 and “Capturing Performance Data” on page 27-29.

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 23-4.

How the Acceleration Modes Work

In this section...
“Overview” on page 27-3
“Normal Mode” on page 27-3
“Accelerator Mode” on page 27-4
“Rapid Accelerator Mode” on page 27-5

Overview

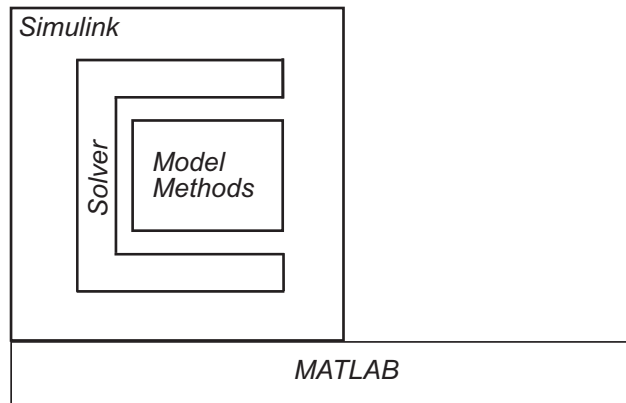
The Accelerator and Rapid Accelerator modes use portions of the Real-Time Workshop product to create an executable. These modes replace the interpreted code normally used in Simulink simulations, shortening model run time.

Although the acceleration modes use some Real-Time Workshop code generation technology, you do not need the Real-Time Workshop 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 Real-Time Workshop 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.



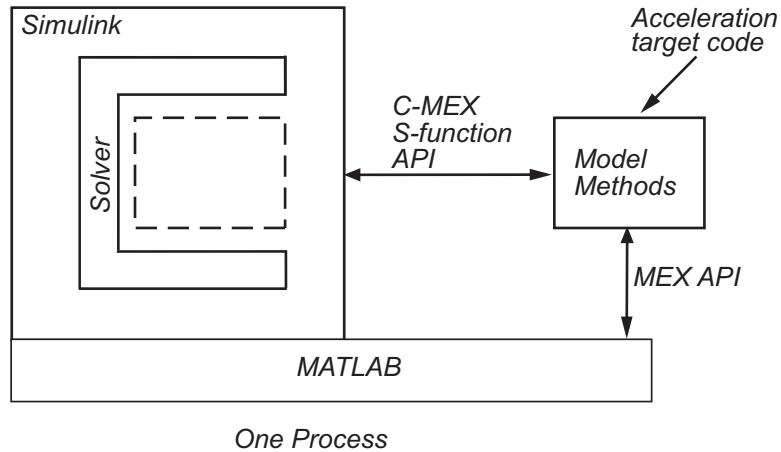
One Process

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.

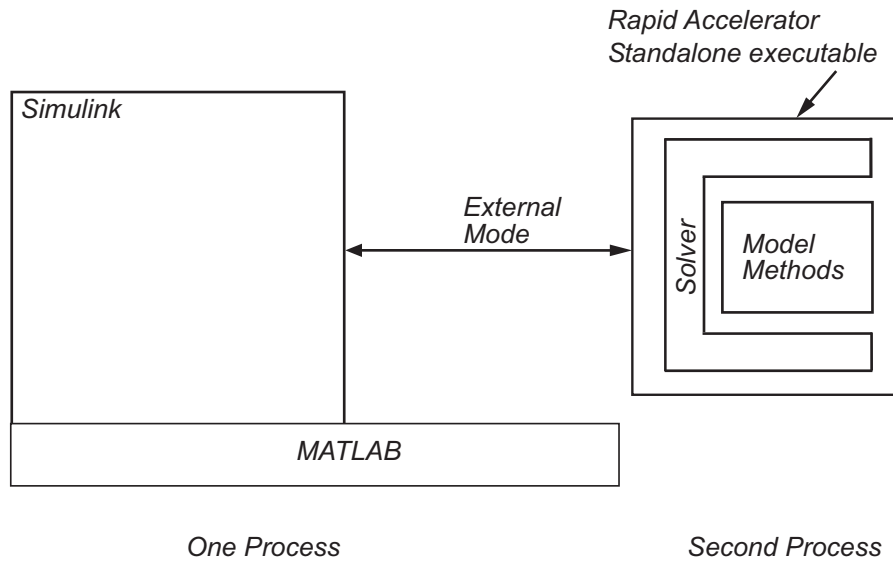
Simulink checks that the acceleration target code is up to date before reusing it. As explained in “Code Regeneration in Accelerated Models” on page 27-7, the target code regenerates if it is not up to date.

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 “Communicating With Code Executing on a Target System Using Simulink External Mode”) 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 27-7

“Determining If the Simulation Will Rebuild” on page 27-7

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 the Rapid Accelerator mode
- Changing solver parameters such as `stop time` or `rel tol` when using the 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 demo model `slAcceIDemoWhyRebuild`.

Parameter Handling in Rapid Accelerator Mode

In terms of model rebuilds, Block Diagram and Run-time parameters are handled differently than other parameters in rapid accelerator mode.

Some Block Diagram parameters can be changed 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	Decimation
ConsecutiveZCsStepRelTol	FinalStateName
ExtrapolationOrder	InitialState
InitialStep	LimitDataPoints
MaxConsecutiveMinStep	LoadExternalInput
MaxConsecutiveZCs	LoadInitialState
MaxNumMinSteps	MaxDataPoints
MaxOrder	OutputOption
MaxStep	OutputSaveName

BLOCK DIAGRAM PARAMETERS THAT DO NOT REQUIRE RAPID ACCELERATOR REBUILD	
MinStep	SaveFinalState
NumberNwtonIterations	SaveFormat
OutputTimes	SaveOutput
Refine	SaveState
RelTol	SaveTime
SolverName	SignalLogging
StartTime	SignalLoggingName
StopTime	StateSaveName
ZCDetectionTol	TimeSaveName

Run-time parameters are collected within an *rtp* structure either by user specification via a *simset* option or through automatic collection during the compile phase. Changes to the Run-time parameters are supported if they are passed directly to *sim*. However, if they are passed graphically via the block diagram or programmatically via the *set_param* command, then a rebuild may be necessary. All other parameter changes may necessitate a rebuild.

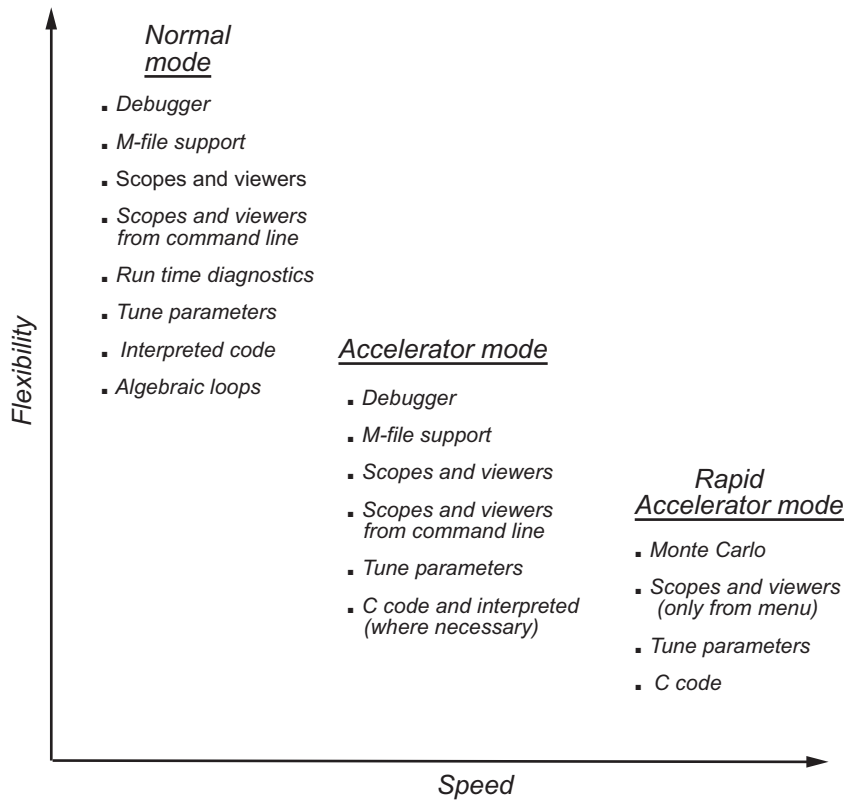
Parameter Changes:	Passed Directly to <i>sim</i> command	Passed Graphically via Block Diagram or via <i>set_param</i> command
Run-time	Rebuild <i>Not</i> Required	Rebuild May Be Required
Block Diagram (Solver and Logging Parameters)	Rebuild <i>Not</i> Required	Rebuild <i>Not</i> Required
Other	Rebuild May Be Required	Rebuild May Be Required

Choosing a Simulation Mode

In this section...
“Tradeoffs” on page 27-10
“Comparing Modes” on page 27-11
“Decision Tree” on page 27-12

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. 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 mdl; % 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.

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 type, stop time without rebuilding	✓	✓	
Change block tunable parameters such as gain	✓	✓	✓
Model Requirement			

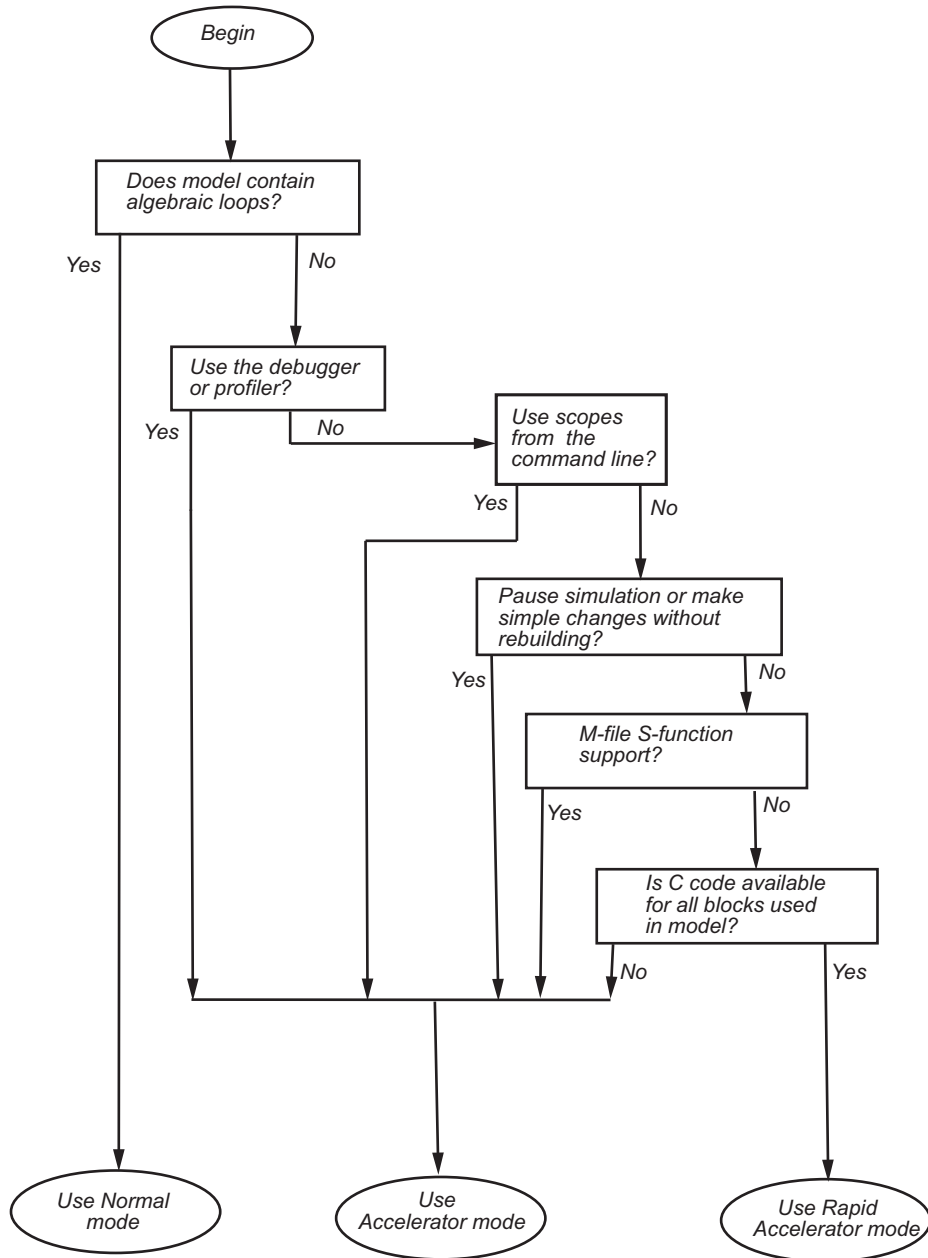
If you want to...	Then use this mode...		
	Normal	Accelerator	Rapid Accelerator
Accelerate your model even if C code is not used for all blocks		✓	
Support M-file S-function blocks	✓	✓	
Permit algebraic loops in your model	✓		
Have your model work with the debugger or profiler	✓	✓	
Have your model include C++ code	✓	✓	
Data Display			
Use scopes and signal viewers	✓	✓	See “Using Scopes and Viewers with Rapid Accelerator Mode” on page 27-16
Use scopes and signal viewers when running your model from the command line	✓	✓	

Note Scopes and viewers do not update if you run your model from the command line in Rapid Accelerator mode.

Decision Tree

The following decision tree can help you select between Normal mode, Accelerator mode, or Rapid Accelerator mode.

See “Comparing Performance” on page 23-4 to understand how effective the accelerator modes will be in improving the performance of your model.



Designing Your Model for Effective Acceleration

In this section...
“Selecting Blocks for Accelerator Mode” on page 27-14
“Selecting Blocks for Rapid Accelerator Mode” on page 27-15
“Controlling S-Function Execution” on page 27-15
“Accelerator and Rapid Accelerator Mode Data Type Considerations” on page 27-16
“Using Scopes and Viewers with Rapid Accelerator Mode” on page 27-16
“Factors Inhibiting Acceleration” on page 27-17

Selecting Blocks for Accelerator Mode

For some blocks, the Accelerator mode uses normal interpreted code rather than C code. The Accelerator mode runs even if these blocks are in your model, but with degraded performance.

These Simulink blocks use interpreted code:

- Display
- Embedded MATLAB Function
- From File
- From Workspace
- Inport (root level only)
- MATLAB Fcn
- 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.

Selecting 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:

- MATLAB Fcn
- 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.

Controlling 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 128 bits
- Rapid Accelerator mode supports root inputs of Enumerated data type
- Rapid Accelerator mode supports bus objects as parameters.
- The Accelerator mode and Rapid Accelerator mode store integers as compactly as possible.
- Simulink Fixed Point does not collect min, max, or overflow data in the Accelerator or Rapid Accelerator modes.

Using Scopes and Viewers with Rapid Accelerator Mode

Running the simulation from the command line or the menu determines 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	Not supported	Not supported
Multirate signal viewers	Not supported	Not supported

Rapid Accelerator mode does not support multirate signal viewers such as the Signal Processing Blockset spectrum scope or the Communications Blockset™ 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. “Running Acceleration Mode from the User Interface” on page 27-21 shows how to run Rapid Accelerator mode from the menu. “Interacting with the Acceleration Modes Programmatically” on page 27-24 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 M-file S-functions that are not numeric, logical, or character arrays, are sparse arrays, or that have more than two dimensions
- Contains algebraic loops
- UsesFcn blocks containing trigonometric functions having complex inputs

Rapid Accelerator mode does not support targets written in C++.

For Rapid Accelerator mode, 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`

Reserved Keywords

Certain words are reserved for use by the Real-Time Workshop 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 Real-Time Workshop product are listed in “Configuring 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`

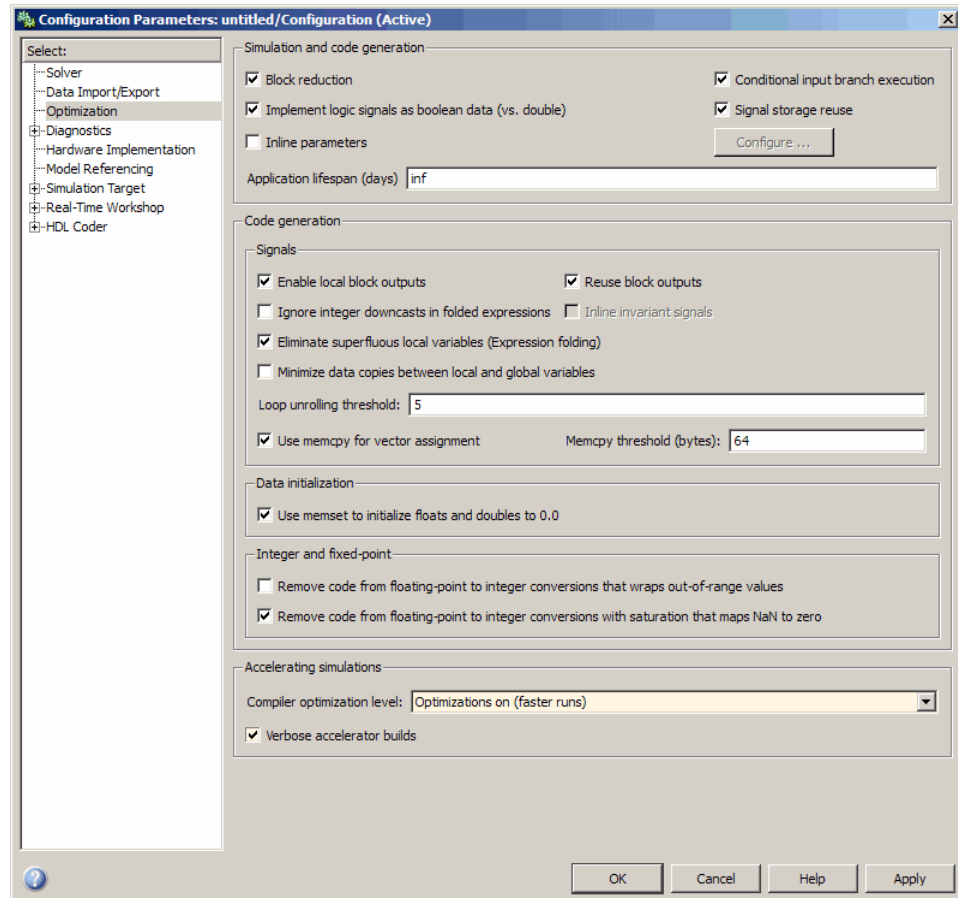
`muDoubleScalarTanh`

Performing Acceleration

In this section...
“Customizing the Build Process” on page 27-20
“Running Acceleration Mode from the User Interface” on page 27-21
“Making Run-Time Changes” on page 27-23

Customizing the Build Process

Compiler optimizations are off by default. This results in faster build times. To optimize acceleration of your model, set the compiler optimization level from the Optimization pane in the Configuration Parameters dialog box.



Select **Optimizations on (faster runs)** when you want to create optimized code. Code generation takes longer with this option, but the model runs faster.

Select **Verbose accelerator builds** to display progress information using code generation, and to see the compiler options in use.

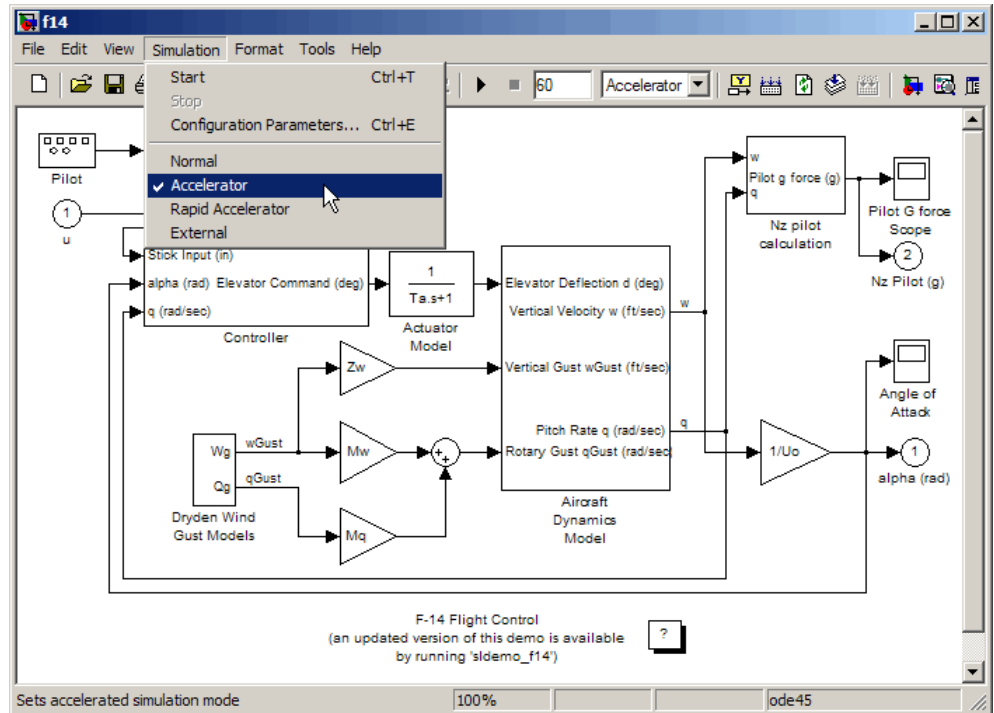
Running Acceleration Mode from the User Interface

To accelerate a model, first open it, and then from the **Simulation** 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** menu, select **Accelerator**.

Alternatively, you can select Accelerator from the model editor's toolbar.



2 From the **Simulation** menu, select **Start**.

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 modes run the model. If code was not previously built, they first generate and compile the C code, and then run the model.

For reasons why these modes rebuild your model, see “Code Regeneration in Accelerated Models” on page 27-7.

The Accelerator mode places the generated code in a subdirectory called *modelname_accel_rtw* (for example, *f14_accel_rtw*), and places a compiled MEX-file in the current working directory.

The Rapid Accelerator mode places the generated code in a subdirectory called *modelname_raccel_rtw* (for example, *f14_raccel_rtw*).

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 modes.

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.

Interacting with the Acceleration Modes Programmatically

In this section...

“Why Interact Programmatically?” on page 27-24

“Building Accelerator Mode MEX-files” on page 27-24

“Controlling Simulation” on page 27-24

“Simulating Your Model” on page 27-25

“Customizing the Acceleration Build Process” on page 27-26

Why Interact Programmatically?

You can build an accelerated model, select the simulation mode, and run the simulation from the command prompt or from M-files. 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.

Building 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
```

Controlling 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 `gcs` (“get current system”) 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(gcs,'SimulationMode','rapid');
```

Simulating Your Model

You can use `set_param` to configure the model parameters (such as the simulation mode and the stop time), and use the 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 27-7 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.

Customizing 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 “Customizing the Build Process” on page 27-20 for details on why doing so might be advantageous.

Controlling the Build Process

Use the `SimCompilerOptimization` parameter to control the acceleration build process. 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')
```

Using the Accelerator Mode with the Simulink Debugger

In this section...

“Advantages of Using Accelerator Mode with the Debugger” on page 27-27

“How to Run the Debugger” on page 27-27

“When to Switch Back to Normal Mode” on page 27-28

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 on the debugger, see Chapter 26, “Simulink Debugger”.

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** 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

Capturing Performance Data

In this section...

“What Is the Profiler?” on page 27-29

“How the Profiler Works” on page 27-29

“Enabling the Profiler” on page 27-31

“How to Save Simulink Profiler Results” on page 27-34

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 **Profiler** from the **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.

The screenshot shows the 'Simulink Profiler Report' window. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Go', 'Debug', 'Desktop', 'Window', and 'Help'. Below the menu bar are navigation icons. A navigation bar contains links: [Summary](#), [Function Details](#), [Simulink Profiler Help](#), and [Clear Highlighted Blocks](#).

Simulink Profile Report: Summary

Report generated 12-Jan-2009 12:24:50

Total recorded time: 0.06 s
 Number of Block Methods: 13
 Number of Internal Methods: 9
 Number of Nonvirtual Subsystem 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	0.06250000	100.0%	1	0.06250000000	0.00000000	0.0%	vdp
ModelExecute	0.03125000	50.0%	1	0.03125000000	0.01562500	25.0%	vdp
ModelInitialize	0.03125000	50.0%	1	0.03125000000	0.03125000	50.0%	vdp
vdp (MinorDeriv)	0.01562500	25.0%	331	0.00004720544	0.01562500	25.0%	vdp
MinorDeriv	0.01562500	25.0%	331	0.00004720544	0.00000000	0.0%	vdp
Integrate	0.01562500	25.0%	55	0.00028409091	0.00000000	0.0%	vdp
ModelTerminate	0.00000000	0.0%	1	0.00000000000	0.00000000	0.0%	vdp
vdp (MinorOutput)	0.00000000	0.0%	330	0.00000000000	0.00000000	0.0%	vdp
MinorOutputs	0.00000000	0.0%	330	0.00000000000	0.00000000	0.0%	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 submodel. Profiling occurs only if the submodel executes in Normal mode. See Chapter 6, “Referencing a Model” 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:

```
slprofreport(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  
slprofreport(profile1);
```


Customizing the Simulink User Interface

- “Adding Items to Model Editor Menus” on page 28-2
- “Disabling and Hiding Model Editor Menu Items” on page 28-13
- “Disabling and Hiding Dialog Box Controls” on page 28-15
- “Customizing the Library Browser” on page 28-21
- “Registering Customizations” on page 28-24

Adding Items to Model Editor Menus

In this section...

“About Adding Items to the Model Editor Menus” on page 28-2

“Code Example” on page 28-2

“Defining Menu Items” on page 28-4

“Registering Menu Customizations” on page 28-9

“Callback Info Object” on page 28-10

“Debugging Custom Menu Callbacks” on page 28-10

“About Menu Tags” on page 28-10

About Adding Items to the Model Editor Menus

You can add commands and submenus to the end of the Simulink model editor menus. Adding an item to the end of a Model Editor menu entails performing the following tasks:

- For each item, create a function, called a *schema function*, that defines the item (see “Defining Menu Items” on page 28-4).
- Register the menu customizations with the Simulink customization manager at startup, e.g., in an `sl_customization.m` file on the MATLAB path (see “Registering Menu Customizations” on page 28-9).
- Create callback functions that implement the commands triggered by the items that you add to the menus.

Note You can use the procedures described in the following sections to customize the Stateflow Chart Editor’s menu.

Code Example

The following `sl_customization.m` file adds four items to the 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

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
    end
    schema.callback = @myToggleCallback;
end
```

Defining 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 28-4
- “Defining Custom Submenus” on page 28-7

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 28-10) and create and return an action schema object (see “Action Schema Object” on page 28-5) 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';

    %% 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**

String 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.

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`

This property must be set to one of the following values:

`'checked'` Menu item displays a check mark.

`'unchecked'` Menu item does not display a check mark.

- `state`

String that specifies the state of this toggle 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 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.

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 28-7) 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 28-4.

```
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.
- `userdata`
Data that you specify. May be of any type.
- `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 “Defining Menu Items” on page 28-4)
 - 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 28-10) passed to the schema function
 - 'separator', which causes a separator to appear between the item defined by the preceding entry in the cell array and the item defined in the following entry. This case is ignored for this entry, e.g., 'SEPARATOR' and 'Separator' are valid entries. A separator is also suppressed if it would appear at the beginning or end of the submenu and combines separators that would appear successively (e.g., as a result of an item being hidden) into a single separator.

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.

Registering 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 28-24) 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 28-24). Each invocation should pass the tag of the menu (see “About Menu Tags” on page 28-10) to be customized and a custom menu function that specifies the items to be added to the menu (see “Creating the Custom Menu Function” on page 28-9). For example, the following `sl_customization` function adds custom items to the Simulink Tools menu.

```
function sl_customization(cm)
    %% Register custom menu function.
    cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyItems);
```

Creating the Custom Menu Function

The custom menu function returns a list of schema functions that define custom items that you want to appear on the model editor menus (see “Defining Menu Items” on page 28-4).

Your custom menu function should accept a callback info object (see “Callback Info Object” on page 28-10) and return a cell array that lists the schema functions. Each element of the cell array can be either a handle to a schema function or a two-element cell array whose first element is a handle to a

schema function and whose second element is user-defined data to be passed to the schema function. For example, the following custom menu function returns a cell array that lists three schema functions.

```
function schemas = getMyItems(callbackInfo)
    schemas = {@getItem1, ...
              @getItem2, ...
              {@getItem3,3} }; % Pass 3 as userdata to getItem3.
end
```

Callback Info Object

Instances of these objects are passed to menu customization functions. 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 M-file debugger's keyboard commands to continue execution of the callback.

About Menu Tags

A menu tag is a string that identifies a Simulink Model Editor or Stateflow Chart Editor menu bar or menu. You need to know a menu's tag to add

custom items to it (see “Registering Menu Customizations” on page 28-9). You can configure the editor to display all (see “Displaying Menu Tags” on page 28-11) but the following tags:

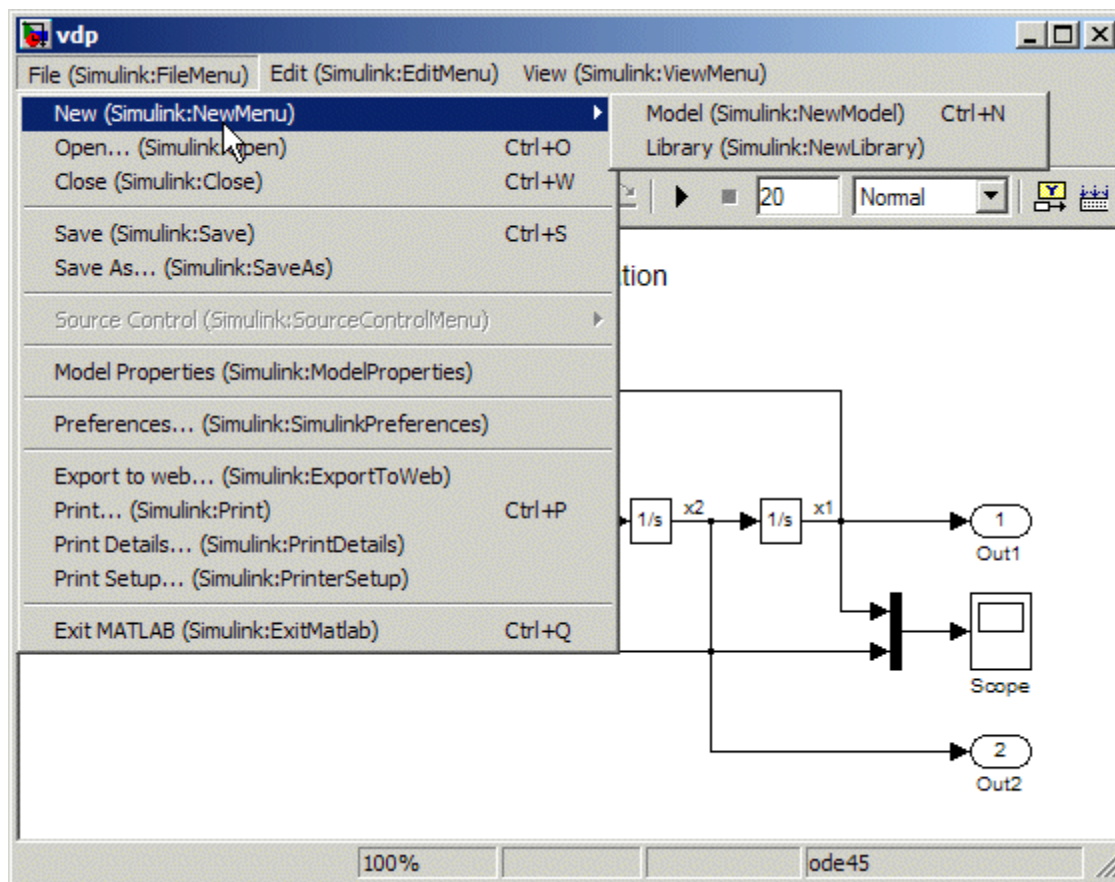
Tag	Usage
Simulink:MenuBar	Add menus to Model Editor’s menu bar.
Simulink:ContextMenu	Add items to the end of Model Editor’s context menu.
Simulink:PreContextMenu	Add items to the beginning of Model Editor’s context menu.
Stateflow:MenuBar	Add menus to Chart Editor’s menu bar.
Stateflow:ContextMenu	Add items to the end of Chart Editor’s context menu.
Stateflow:PreContextMenu	Add items to the beginning of Chart Editor’s context menu.

Displaying Menu Tags

You can configure the Simulink software (and the Stateflow product) 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. To configure the editor to display menu tags, set the customization manager’s `showWidgetIdAsToolTip` property to `true`, e.g., by entering the following commands at the command line:

```
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.

Disabling and Hiding Model Editor Menu Items

In this section...

“About Disabling and Hiding Model Editor Menu Items” on page 28-13

“Example: Disabling the New Model Command on the Simulink Editor’s File Menu” on page 28-13

“Creating a Filter Function” on page 28-13

“Registering a Filter Function” on page 28-14

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 28-13).
- Register the filter function with the customization manager (see “Registering a Filter Function” on page 28-14).

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 28-11) 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
```

Disabling and Hiding Dialog Box Controls

In this section...

“About Disabling and Hiding Controls” on page 28-15

“Example: Disabling a Button on a Simulink Dialog Box” on page 28-16

“Writing Control Customization Callback Functions” on page 28-17

“Dialog Box Methods” on page 28-17

“Dialog Box and Widget IDs” on page 28-18

“Registering Control Customization Callback Functions” on page 28-19

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 28-18), 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 M-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 “Writing Control Customization Callback Functions” on page 28-17) and registering the callback functions via an object called the customization manager (see “Registering Control Customization Callback Functions” on page 28-19). 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:

Example: Disabling a Button on a Simulink Dialog Box

The following `sl_customization.m` file disables the **Build** button on the **Real-Time Workshop** 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 28-24).
- 3 Open the `sldemo_engine` demo model, for example, by entering the command `sldemo_engine` at the command line.

Writing 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 **Real-Time Workshop** 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)`

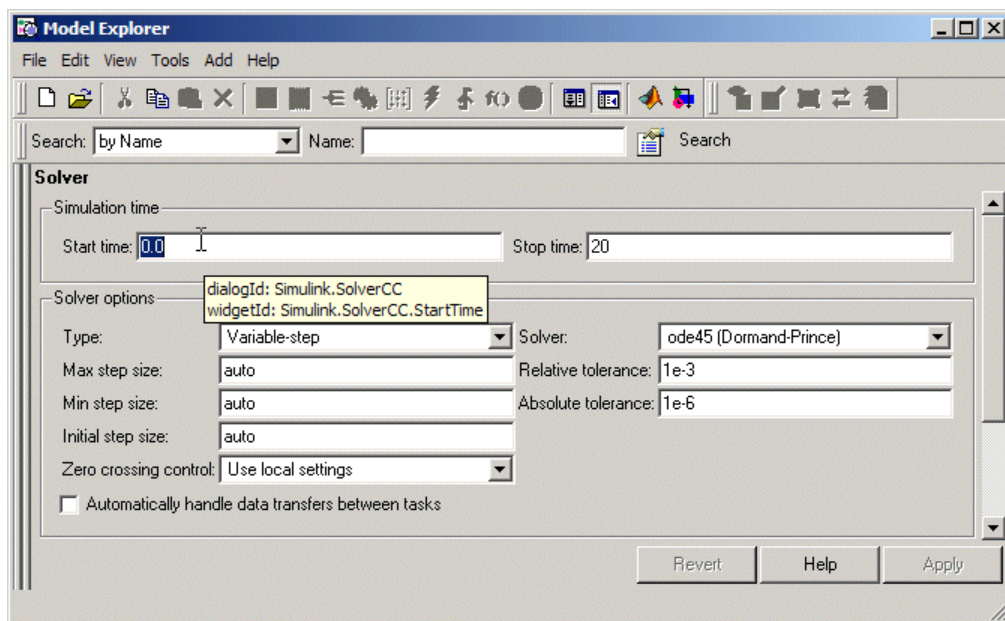
where `widgetIDs` is a cell array of widget identifiers (see “Dialog Box and Widget IDs” on page 28-18) that specify the widgets to be disabled or hidden.

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.

Registering 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 28-24) 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 28-18) 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 **Real-Time Workshop** pane of the Configuration Parameters dialog box (see “Writing Control Customization Callback Functions” on page 28-17).

```
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.

Customizing the Library Browser

In this section...

“Reordering Libraries” on page 28-21

“Disabling and Hiding Libraries” on page 28-21

“Customizing the Library Browser’s Menu” on page 28-22

Reordering 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 28-24) on the MATLAB path:

```
cm.LibraryBrowserCustomizer.applyOrder( {'LIBNAME1', PRIORITY1, ...
                                         'LIBNAME2', 'PRIORITY2, ...
                                         .
                                         .
                                         'LIBNAMEN', PRIORITYN} );
```

where `LIBNAMEN` is the name of the library or its `.mdl` 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} );
```

Disabling and Hiding 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 28-24) 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 'STATEN' 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'});
```

Customizing the Library Browser's Menu

You can perform the same kinds of customizations to the Library Browser's 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 28-24

“Customization Manager” on page 28-24

About Registering User Interface Customizations

You must register your user-interface customizations using an M-file 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 28-24). 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.

- “Adding Items to Model Editor Menus” on page 28-2
- “Disabling and Hiding Model Editor Menu Items” on page 28-13
- “Disabling and Hiding Dialog Box Controls” on page 28-15

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 28-11 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 “Adding Items to Model Editor Menus” on page 28-2 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 28-11 for more information). The `filterFcn` argument is a pointer to a function that hides or disables the specified menu item. See “Disabling and Hiding Model Editor Menu Items” on page 28-13 for more information.

Creating Custom Blocks

- “When to Create Custom Blocks” on page 29-2
- “Types of Custom Blocks” on page 29-3
- “Comparison of Custom Block Functionality” on page 29-7
- “Expanding Custom Block Functionality” on page 29-17
- “Tutorial: Creating a Custom Block” on page 29-18
- “Custom Block Examples” on page 29-44

When to Create Custom Blocks

Custom blocks allow you to expand the modeling functionality provided by the Simulink product. By creating a custom block, you can:

- Model behaviors for which the Simulink product does not provide a built-in solution.
- Build more advanced systems.
- Encapsulate systems into a library block that can be copied into multiple models.
- Provide custom graphical user interfaces or analysis routines.

Types of Custom Blocks

In this section...
“MATLAB Function Blocks” on page 29-3
“Subsystem Blocks” on page 29-4
“S-Function Blocks” on page 29-4

MATLAB Function Blocks

MATLAB function blocks allow you to use functions to define custom functionality. These blocks serve as 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 via a MATLAB function than via a block diagram.
- The custom functionality does not include continuous or discrete dynamic states.

You can create a custom block from an M-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 MATLAB Fcn block allows you to use a MATLAB function to define a SISO block.
- The Embedded MATLAB Function block allows you to define a custom block with multiple inputs and outputs that can be deployed to embedded processors.

Each of these blocks has advantages in particular modeling applications. For example, you can generate code from models containing Embedded MATLAB Function blocks while you cannot generate code for models containing a 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 via a graphical representation than via hand-written code.
- The custom functionality is a function of continuous or discrete system states.
- The custom functionality can be modeled using existing Simulink blocks.

Once you have a Simulink subsystem that models the desired behavior, you can convert it into a custom block by:

- 1 Masking the block to hide the block's 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.

See Chapter 8, “Working with Block Libraries” and Chapter 9, “Working with Block Masks” in Using Simulink for more information.

S-Function Blocks

S-function blocks allow you to write M, 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 M, 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.
- The custom functionality cannot be modeled 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 M-File S-Function block allows you to write your S-function in M. (See “Writing S-Functions in M”.) An M-file S-function can be debugged during 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 via a C MEX wrapper. (See “Writing S-Functions in C”.)
- 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 “Building S-Functions Automatically”.)
- The Legacy Code Tool transforms existing C or C++ functions into C MEX S-functions. (See “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool”.)

The S-function target in the Real-Time Workshop 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 “Creating Component Object Libraries and Enhancing Simulation Performance” in the Real-Time Workshop User’s Guide for details and limitations on using the S-function target.

Comparing M-File S-Functions to Embedded MATLAB Functions

M-file S-functions and Embedded MATLAB functions have some fundamental differences.

- The Real-Time Workshop product can generate code for both M-file S-functions and Embedded MATLAB functions. However, M-file S-functions require a Target Language Compiler (TLC) file for code generation. Embedded MATLAB functions do not require a TLC-file.
- M-file S-functions can use any MATLAB function. Embedded MATLAB functions support only a subset of the MATLAB functions. See “Embedded MATLAB Function Library Reference” in the Embedded MATLAB documentation for a list of supported functions.
- M-file S-functions can model discrete and continuous state dynamics. Embedded MATLAB functions 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.

- An M-file 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 “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool” for more information.

See “Integration Options” in the Real-Time Workshop User’s Guide for more information.

See “Example Using S-Functions to Incorporate Legacy C Code” in “Writing S-Functions” for an example.

Comparison of Custom Block Functionality

In this section...

“Custom Block Considerations” on page 29-7

“Modeling Requirements” on page 29-10

“Speed and Code Generation Requirements” on page 29-13

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’s inputs and outputs have various data attributes, such as data types or complexity?
- How important is the affect 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.

Modeling Requirements (Continued)

Custom Block Type	Supports Multiple Inputs and Outputs	Models State Dynamics	Supports Various Data Attributes
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.
MATLAB Fcn	No. Must have a single vector input and output.	No.	Supports only n-D, real, or complex signals with a data type of double.
Embedded 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 M-file 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.
MATLAB Fcn	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.
Embedded 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 29-16 for more information.
Level-2 M-file S-function	Can be slower if the S-function overrides methods executed when updating the diagram.	Higher than for MATLAB Fcn blocks because the MATLAB interpreter is called for every S-function method used. Very flexible, but very costly.	M-file S-functions initialized as a <code>SimViewingDevice</code> do not generate code. Otherwise, M-file S-functions require a TLC-file for code generation.
C MEX S-function	Can be slower if the S-function overrides methods executed when updating the diagram.	Minimal, but proportional to the complexity of the algorithm and the efficiency of the code.	Might require a TLC-file.

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 8-5 for more information.
Fcn, MATLAB Fcn	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.
Embedded MATLAB Function	Supports multiple input and output ports, including bus signals. See “Working with Structures and Bus Signals” on page 30-100 for more information.
S-function (M-file 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 “Integrating Existing C Functions into Simulink Models with the 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, MATLAB Fcn, Embedded MATLAB Function	All create an <code>mdlOutput</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.
M-file 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 M-File 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.
MATLAB Fcn	Supports 2-D, n-D, and complex signals, but the signal must have a data type of double. Does not support frame-based signals.

Custom Block Type	Data Attribute Support
Embedded 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 (M-file 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, MATLAB Fcn	Does not incur greater update cost than other Simulink blocks.
Embedded 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, the Simulink software does not repeatedly update the block.
S-function (M-file 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 “How the Simulink Engine Interacts 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, mdlStart.

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 “Capturing Performance Data” on page 27-29 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.
MATLAB Fcn	<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 a MATLAB Fcn 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 the MATLAB software is able to optimize it, or if the code uses the highly optimized MATLAB library functions.</p>

Custom Block Type	Simulation Overhead
Embedded 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, an Embedded MATLAB function and a MATLAB function run at about the same speed. To further reduce the algorithm cost, you can disable debugging for all the Embedded MATLAB Function blocks in your model.</p> <p>If the Embedded MATLAB function uses simulation-only capabilities to call out to the MATLAB interpreter, it incurs all the costs that an M-file S-function or MATLAB Fcn block incur. Calling out to the MATLAB interpreter from an Embedded MATLAB function produces a warning to prevent you from doing so unintentionally.</p>
M-file S-function	<p>Incurs the same algorithm costs as the MATLAB Fcn block, but with a slightly higher interface cost. Because M-file S-functions can handle multiple inputs and outputs, the packaging is more complicated than for the MATLAB Fcn block. In addition, the Simulink engine calls the MATLAB interpreter for each block method you implement whereas for the MATLAB Fcn block, it calls the MATLAB interpreter only for the mdlOutput 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.

Custom Block Type	Code Generation Support
Subsystem	Supports code generation.
Fcn	Supports code generation.
MATLAB Fcn	Does not support code generation.
Embedded MATLAB Function	Supports code generation. However, if your Embedded MATLAB Function block calls out to the MATLAB interpreter, it will build with the Real-Time Workshop product only if the calls to the MATLAB interpreter do not affect the block's outputs. Under this condition, the Real-Time Workshop product omits these calls from the generated C code. This feature allows you to leave visualization code in place, even when generating embedded code.
M-file 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 M-file S-function is a <code>SimViewingDevice</code> , the Real-Time Workshop product automatically omits the block during code generation.
C MEX S-function	Supports code generation. For noninlined S-functions, the Real-Time Workshop 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 "Integrating External Code With Generated C and C++ Code" in the Real-Time Workshop 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 “Creating Block Callback Functions” on page 4-58.

GUIDE, the MATLAB graphical user interface development environment, provides tools for easily creating custom user interfaces. See *MATLAB Creating Graphical User Interfaces* for more information on using GUIDE.

Tutorial: Creating a Custom Block

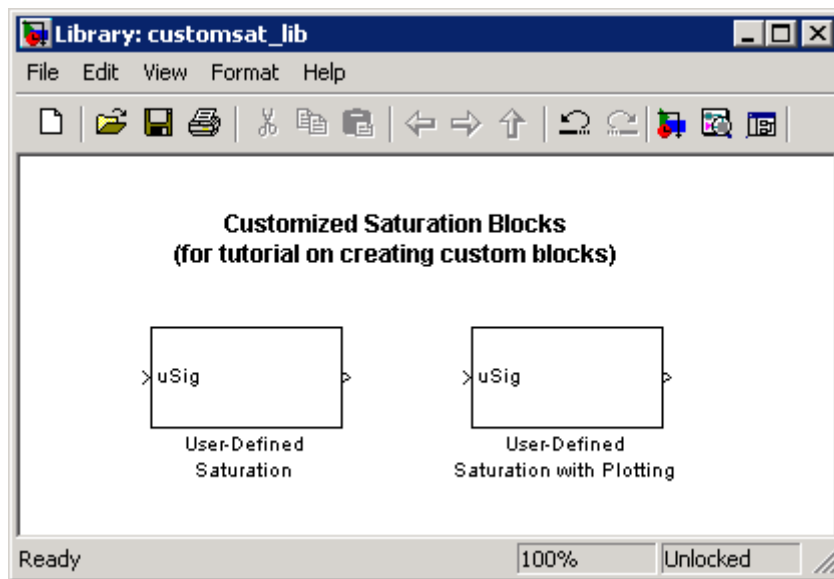
In this section...
“How to Design a Custom Block” on page 29-18
“Defining Custom Block Behavior” on page 29-20
“Deciding on a Custom Block Type” on page 29-21
“Placing Custom Blocks in a Library” on page 29-26
“Adding a Graphical User Interface to a Custom Block” on page 29-28
“Adding Block Functionality Using Block Callbacks” on page 29-37

How to Design a Custom Block

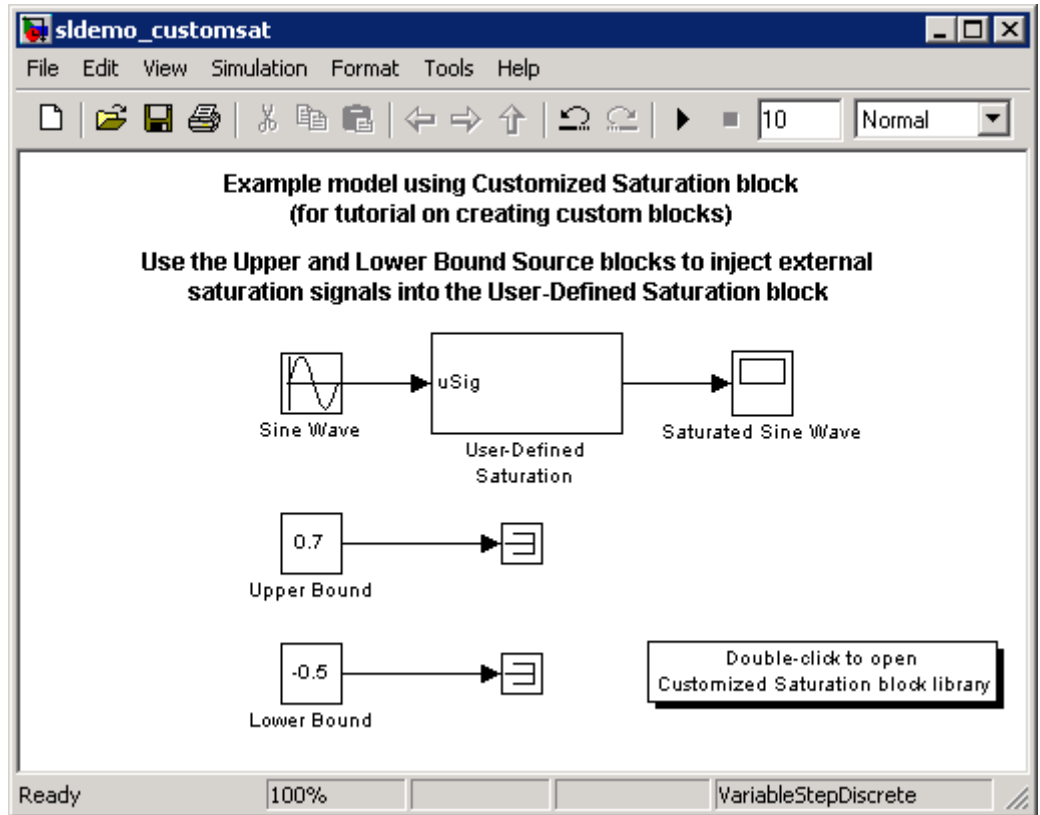
In general, you use the following process to design a custom block:

- 1 Define the behavior required by the custom block.
- 2 Decide which custom block type to use.
- 3 Determine if the block should reside in a library.
- 4 Add a graphical user interface to the block.

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.mdl` contains the two versions of the customized saturation block.



The example model `sldemo_customsat.mdl` 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's 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 M-file S-function. M-file 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 29-7 for a description of the different functionality provided by M-file S-functions as compared to other types of custom blocks.

Parameterizing the M-File 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's 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 M-File S-Function

Once the S-function parameters and functionality are defined, write the S-function. The template `msfuntmpl.m` provides a starting point for writing a Level-2 M-file 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 directory 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
    % M-file
    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 M-file 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.');


```

```

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
```

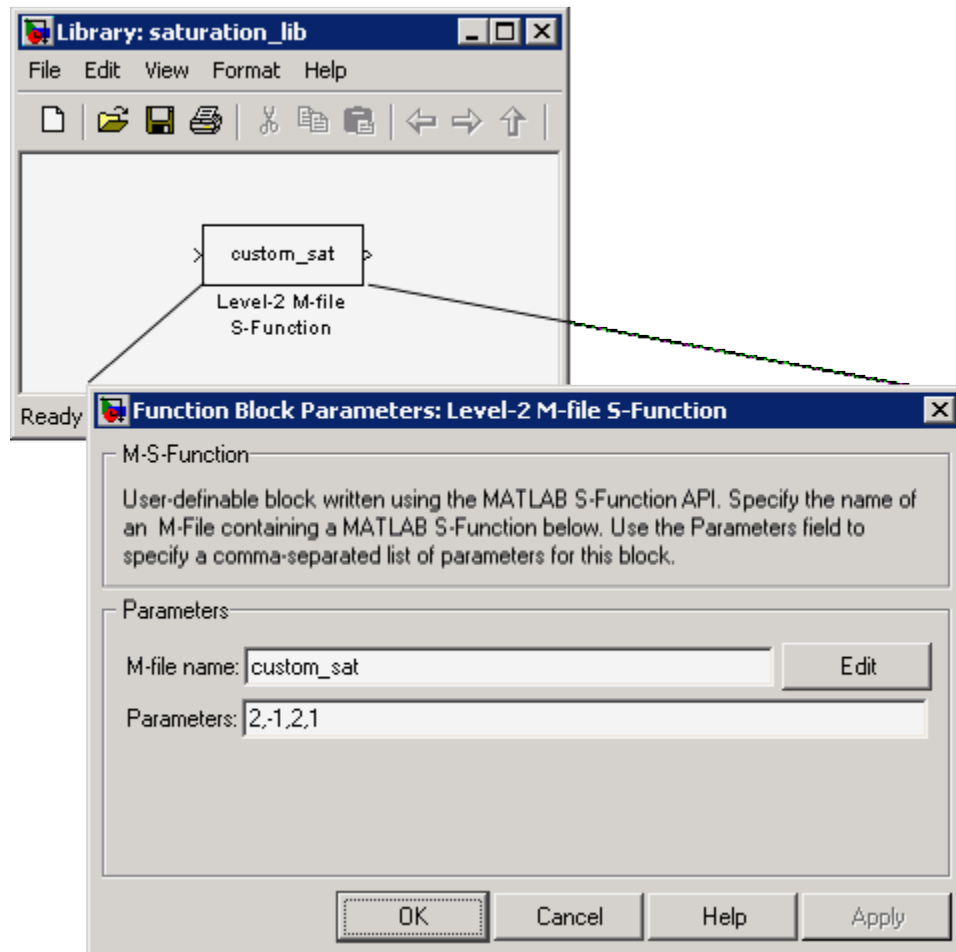
Placing Custom Blocks in a Library

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 as follows:

- 1** Open a new Simulink library (see “Creating a Library” on page 8-14).
- 2** Add a new Level-2 M-file S-Function block from the Simulink User-Defined Functions library into your new library.
- 3** Double-click the block to open its Block Parameters dialog box. Enter the name of the S-function `custom_sat` into the **M-file name** field.
- 4** Enter the following default values into the **Parameters** field.

2, -1, 2, 1
- 5** Click **OK** on the Block Parameters dialog box.
- 6** Save the library to your working directory as `saturation_lib.mdl`. The following figure shows the resulting custom saturation block library.



At this point, you have created a custom saturation block that can be shared with other users. You can make the block easier to use by adding a customized graphical user interface.

Adding a Graphical User Interface to a Custom Block

You can create a simple block dialog for the custom saturation block using the provided masking capabilities. Masking the block also allows you to add port labels to indicate which port corresponds to the input signal and the saturation limits.

To mask the block:

1 Right-click the custom saturation block in `saturation_lib.mdl` and select **Mask M-file S-Function** from the context menu. The Mask Editor opens.

2 On the **Icon** pane, enter the following into **Drawing commands**.

```
port_label('input',1,'uSig')
```

This command labels the default port as the input signal under saturation.

3 On the **Parameters** pane, add four parameters corresponding to the four S-function parameters. From top to bottom, set up each parameter's properties as follows.

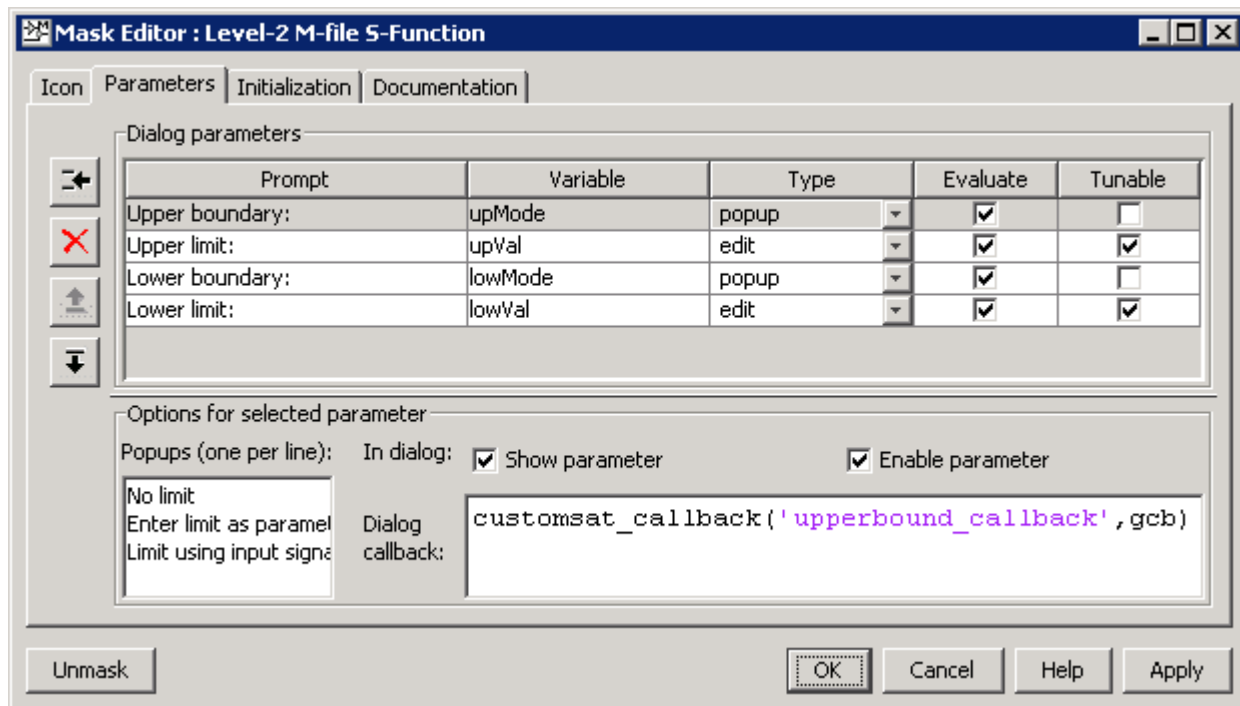
Prompt	Variable	Type	Tunable	Popups	Action for Dialog Callback
Upper boundary:	upMode	popup	No	No limit Enter limit as parameter Limit using input signal	'upperbound_callback'
Upper limit:	upVal	edit	Yes	N/A	'upperparam_callback'
Lower boundary:	lowMode	popup	No	No limit Enter limit as parameter Limit using input signal	'lowerbound_callback'
Lower limit:	lowVal	edit	Yes	N/A	'lowerparam_callback'

The dialog callback is invoked using the action string in the following command:

```
customsat_callback(action,gcb)
```

The M-file `customsat_callback.m` contains the mask parameter callbacks. If you are stepping through this tutorial, open this file and save it to your working directory. This M-file, described in detail later, 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 M-file S-Function block.

The following figure shows the final **Parameters** pane in the Mask Editor.



- 4 On the Mask Editor's **Initialization** pane, select **Allow library block to modify its contents**. This setting allows the S-function to change the number of ports on the block.

5 On the **Documentation** pane:

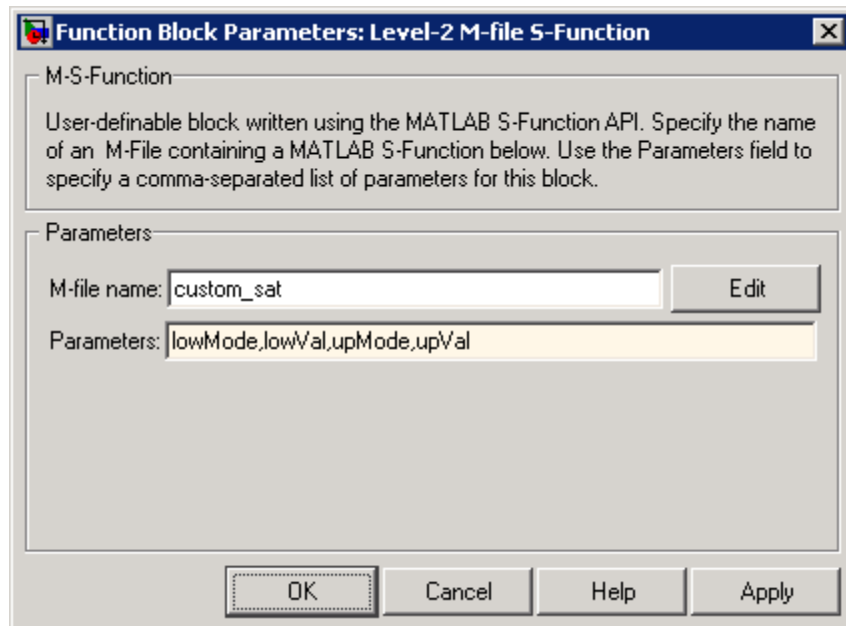
- Enter **Customized Saturation** into the **Mask type** field.
- Enter the following into the **Mask description** field.

Limit the input signal to an upper and lower saturation value set either through a block parameter or input signal.

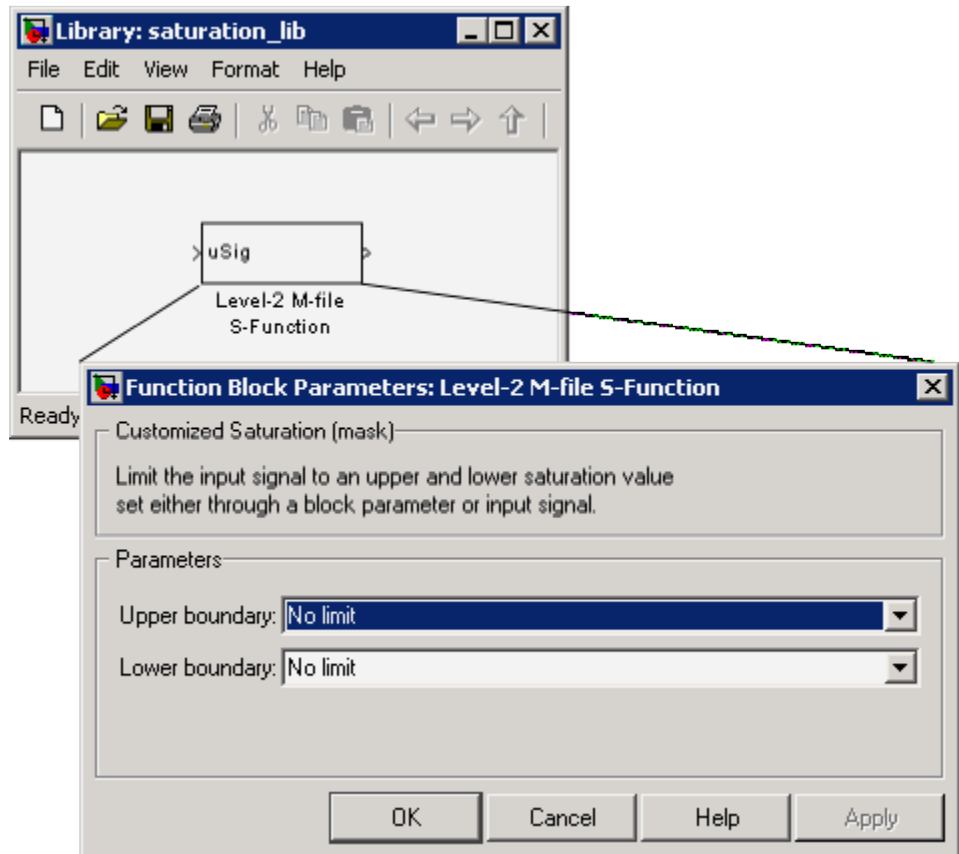
6 Click **OK** on the Mask Editor to complete the mask parameters dialog.**7** To map the S-function parameters to the mask parameters, right-click the Level-2 M-file S-Function block and select **Look Under Mask**. The Level-2 M-file S-Function Block Parameters dialog box opens.**8** Change the entry in the **Parameters** field as follows.

`lowMode,lowVal,upMode,upVal`

The following figure shows the new Block Parameters dialog.



- 9 Click **OK** on the Level-2 M-file S-Function Block Parameters dialog box. Double-clicking the new version of the customized saturation block opens the mask parameter dialog box shown in the following figure.



To create a more complicated graphical user interface, place a Handle Graphics user interface on top of the masked block. The block's `OpenFcn` would invoke the Handle Graphics user interface, which uses calls to `set_param` to modify the S-function block's 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's mask parameter dialog box. This function invokes subfunctions corresponding to each mask parameter through a call to `feval`.

The following subfunction 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 M-file 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 M-File S-Function block consistent with the values shown in the mask parameter dialog box.

The modified M-file 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 directory.

```

%% 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
% M-file
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` subfunction 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];

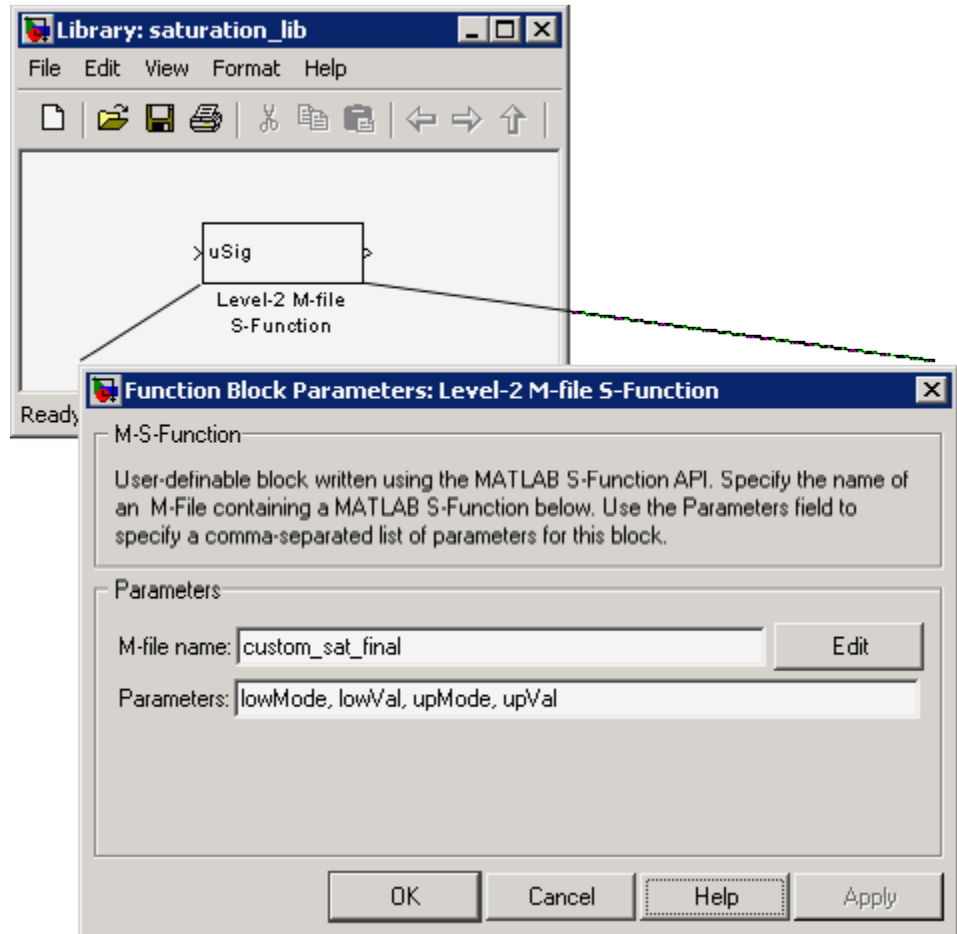
vals = get_param(block.BlockHandle, 'MaskValues');
switch vals{1}
    case 'No limit'
        ud(2) = 1;
    case 'Enter limit as parameter'
        ud(2) = 2;
    case 'Limit using input signal'
        ud(2) = 3;
end

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
```

Updating the Library

Update the library `saturation_lib.mdl` so that it calls `custom_sat_final.m`.

- 1 Right-click the Level-2 M-file S-Function block in `saturation_lib.mdl` and select **Look Under Mask**. The Level-2 M-file S-Function Block Parameters dialog box opens.
- 2 Enter `custom_sat_final` in the **M-file name** field, as shown in the following figure.

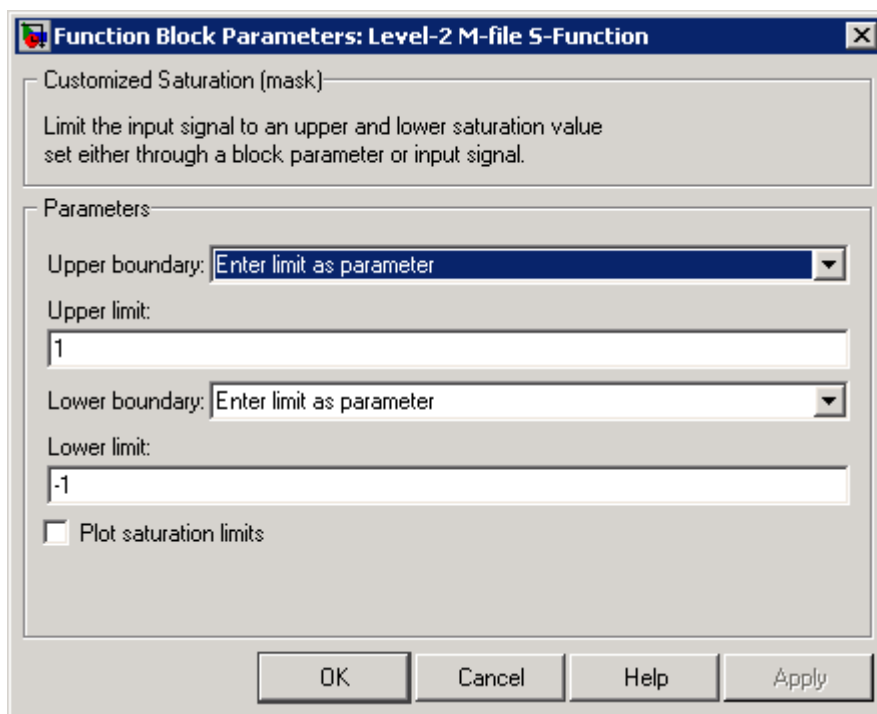


3 Click **OK** on the Block Parameters dialog box.

Adding Block Functionality Using Block Callbacks

The User-Defined Saturation with Plotting block in `customsat_lib.mdl` 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, as shown in the following figure.



To add this check box:

- a Right-click the Level-2 M-file S-Function block in `saturation_lib.mdl` and select **Edit Mask**.
- b On the Mask Editor's **Parameters** pane, add a fifth mask parameter with the following properties.

Property	Value
Prompt	Plot saturation limits
Variable	plotcheck
Type	checkbox

Property	Value
Tunable	No
Dialog callback	customsat_callback('plotsaturation',gcb);

- c Click **OK** on the Mask Editor.
- 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 M-File S-Function block's UserData. The M-file `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 subfunctions over the previous subfunctions in `customsat_callback.m`.

```
%% Plotting checkbox callback
function plotsaturation(block)

% Reinitialize the block's userdata
vals = get_param(block,'MaskValues');
ud = struct('time',[],'upBound',[],'upVal',[],'lowBound',[],'lowVal',[]);

if strcmp(vals{1},'No limit'),
    ud.upBound = 'off';
else
    ud.upBound = 'on';
end

if strcmp(vals{3},'No limit'),
    ud.lowBound = 'off';
else
    ud.lowBound = 'on';
end

set_param(gcb,'UserData',ud);
```

- 3 Update the M-file S-function's Outputs method to store the saturation limits, if applicable, as done in the new M-file 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 M-File 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 directory.

```

function plotSat(block)

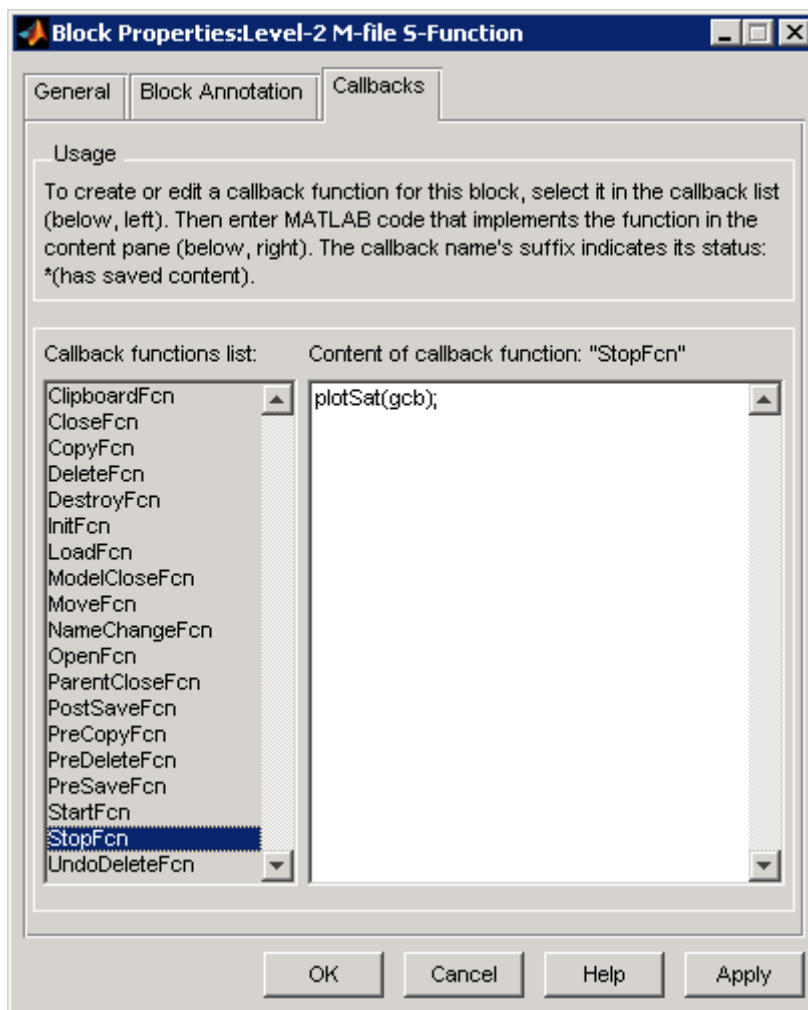
% PLOTSAT contains the plotting routine for custom_sat_plot
% This routine is called by the S-function block's StopFcn.

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
end

```

```
    % Reinitialize userdata
    ud.upVal=[];
    ud.lowVal=[];
    ud.time = [];
    set_param(block,'UserData',ud);
end
```

- 5 Right-click the Level-2 M-file S-Function block and select **Block 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 29-44

“Creating Custom Blocks from MATLAB Functions” on page 29-44

“Creating Custom Blocks from S-Functions” on page 29-45

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 Chapter 8, “Working with Block Libraries” for more general information on Simulink libraries.

Creating Custom Blocks from MATLAB Functions

The Simulink product provides a number of demonstrations that show how to incorporate MATLAB functions into a custom block.

- The Single Hydraulic Cylinder Simulation, `sldemo_hydcyl.mdl`, 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.mdl`.
- The Radar Tracking Model, `sldemo_radar.mdl`, uses a MATLAB Fcn block to model an extended Kalman filter. The M-file `aero_extkalman.m` implements the Kalman filter found inside the Radar Kalman Filter subsystem. In this example, the M-file requires three inputs, which are bundled together using a Mux block in the Simulink model.
- The Spiral Galaxy Formation demonstration, `sldemo_em1_galaxy.mdl`, uses several Embedded MATLAB Function blocks to construct two galaxy and calculate the effects of gravity as these two galaxies nearly collide. The demo also uses Embedded MATLAB Function blocks to plot the simulation results using a subset of MATLAB functions not supported for code generation. However, because these Embedded MATLAB Function blocks have no outputs, the Real-Time Workshop product optimizes them away during code generation.

Creating Custom Blocks from S-Functions

The Simulink model `sfundemos.mdl` contains various examples of M-file and C MEX-file S-functions. For more information on writing M-file S-functions, see “Writing S-Functions in M”. For more information on writing C MEX S-functions, see “Writing S-Functions in C”. For a list of available S-function demos, see “S-Function Examples” in Writing S-Functions.

Using the Embedded MATLAB Function Block

- “Introduction to Embedded MATLAB Function Blocks” on page 30-3
- “Creating an Example Embedded MATLAB Function” on page 30-8
- “Debugging an Embedded MATLAB Function Block” on page 30-23
- “Embedded MATLAB Function Editor” on page 30-37
- “Working with Compilation Reports” on page 30-66
- “Typing Function Arguments” on page 30-81
- “Sizing Function Arguments” on page 30-93
- “Parameter Arguments in Embedded MATLAB Functions” on page 30-97
- “Resolving Signal Objects for Output Data” on page 30-98
- “Working with Structures and Bus Signals” on page 30-100
- “Using Variable-Size Data in Embedded MATLAB Function Blocks” on page 30-114
- “Using Enumerated Data in Embedded MATLAB Function Blocks” on page 30-123
- “Working with Frame-Based Signals” on page 30-134
- “Using Traceability in Embedded MATLAB Function Blocks” on page 30-141
- “Enhancing Readability of Generated Code for Embedded MATLAB Function Blocks” on page 30-148

- “Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library” on page 30-157
- “Controlling Runtime Checks” on page 30-159

Introduction to Embedded MATLAB Function Blocks

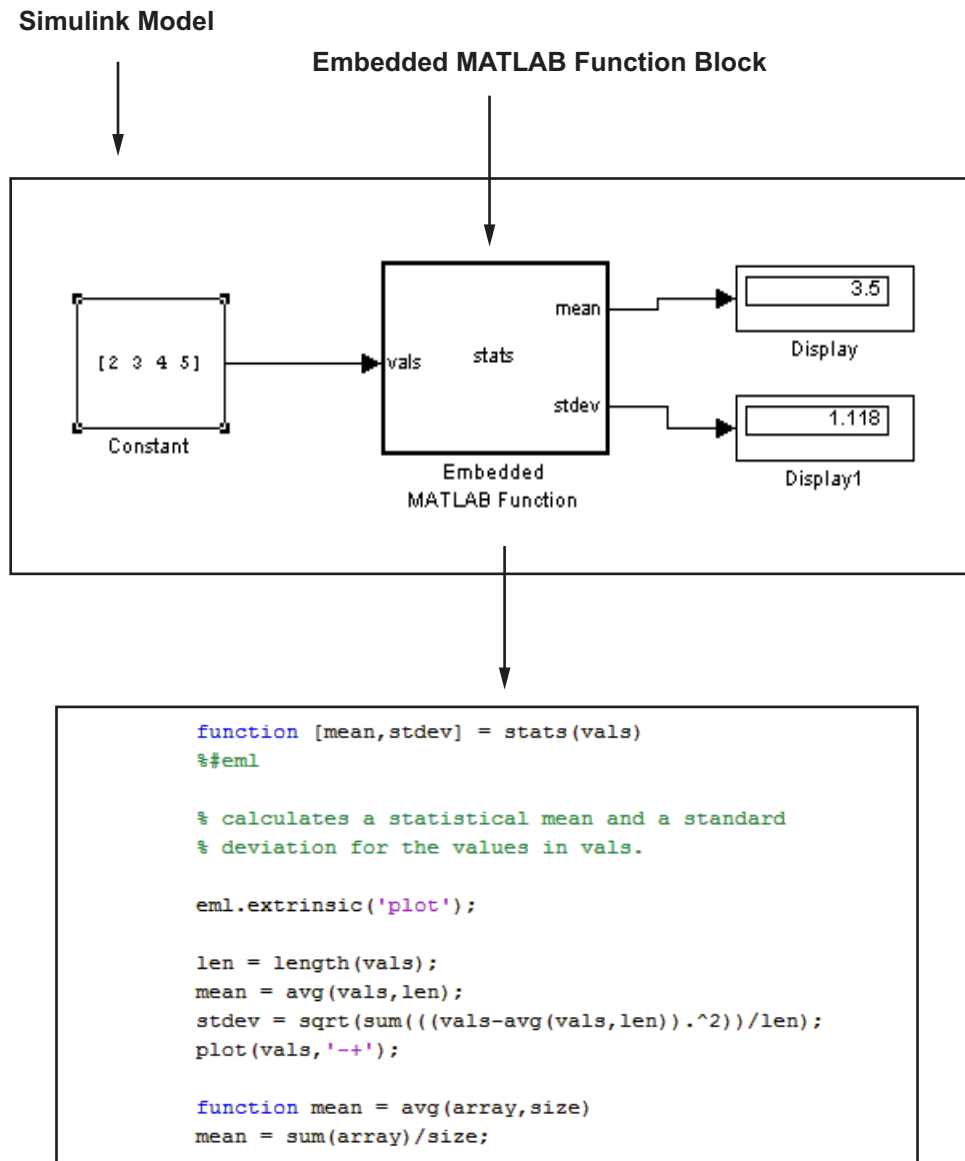
In this section...
“What Is an Embedded MATLAB Function Block?” on page 30-3
“Why Use Embedded MATLAB Function Blocks?” on page 30-6

What Is an Embedded MATLAB Function Block?

The Embedded MATLAB Function block allows you to add MATLAB functions to Simulink models for deployment to 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. This block works with a subset of the MATLAB language called the Embedded MATLAB subset, which provides optimizations for generating efficient, production-quality C code for embedded applications. For more information, see “Working with the Embedded MATLAB Subset” in the MATLAB documentation. For more information on fixed-point support in MATLAB, refer to “Working with the Fixed-Point Embedded MATLAB Subset” in the Fixed-Point Toolbox documentation.

Example: Calculating Statistical Mean and Standard Deviation

Here is an example of a Simulink model that contains an Embedded MATLAB Function block:



You will build this model in “Creating an Example Embedded MATLAB Function” on page 30-8.

Defining Local Variables. Note in this Embedded MATLAB function that you can declare local variables implicitly through assignment, just as you would in MATLAB functions. The variable takes its type and size from the context in which it is assigned. For example, the following code line declares `x` to be a scalar variable of type double.

```
x = 1.54;
```

Once you define a variable, you cannot redefine it to any other type or size in the function body. For example, you cannot declare `x` and reassign it:

```
x = 2.65; % OK: x is a scalar double
x = [x 2*x]; % Error: x cannot be changed to a vector
```

See “Creating Local Complex Variables By Assignment” in the Embedded MATLAB documentation for detailed descriptions and examples.

Declaring Extrinsic Functions. Note in this example, that you can declare functions to be extrinsic by using `eml.extrinsic`. This ensures that the Embedded MATLAB software does not attempt to compile this function. Instead, the function will execute in the MATLAB workspace during simulation of the model. For example, the following code declares the `plot` function to be extrinsic:

```
eml.extrinsic('plot');
```

See “Calling Embedded MATLAB Library Functions” in the Embedded MATLAB documentation for more information.

Calling Functions in Embedded MATLAB Function Blocks

In addition to supporting a rich subset of the MATLAB language, Embedded MATLAB Function blocks can call any of the following types of functions:

- **Subfunctions**

Subfunctions are defined in the body of the Embedded MATLAB block. In the preceding example, `avg` is a subfunction. See “Calling Subfunctions” in the Embedded MATLAB documentation.

- **Embedded MATLAB runtime library functions**

Embedded MATLAB runtime library functions are a subset of the functions that you call in MATLAB. When you build your model with Real-Time Workshop, these functions generate C code that conforms to the memory and variable type requirements of embedded environments. In the preceding example, `length`, `sqrt`, and `sum` are Embedded MATLAB runtime library functions. See “Calling Embedded MATLAB Library Functions” in the Embedded MATLAB documentation.

- **MATLAB functions**

Function calls that cannot be resolved as subfunctions or Embedded MATLAB runtime library functions are extrinsic functions and are resolved in the MATLAB workspace. These functions do not generate code; they execute only in the MATLAB workspace during simulation of the model. The Embedded MATLAB subset attempts to compile all MATLAB functions unless you explicitly declare them to be extrinsic by using `eml.extrinsic`. See “Calling MATLAB Functions” in the Embedded MATLAB documentation.

Why Use Embedded MATLAB Function Blocks?

Embedded MATLAB Function blocks provide the following capabilities:

- **Allow you to build MATLAB functions into embeddable applications** — Embedded MATLAB Function blocks support a subset of MATLAB commands that generate efficient C code (see “Embedded MATLAB Function Library Reference” in the Embedded MATLAB documentation). With this support, you can use Real-Time Workshop to generate embeddable C code from Embedded MATLAB Function blocks that implements 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 an Embedded 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

Model Explorer or Ports and Data Manager (see “Ports and Data Manager” on page 30-42).

Creating an Example Embedded MATLAB Function

In this section...

“Adding an Embedded MATLAB Function Block to a Model” on page 30-8

“Programming the Embedded MATLAB Function” on page 30-10

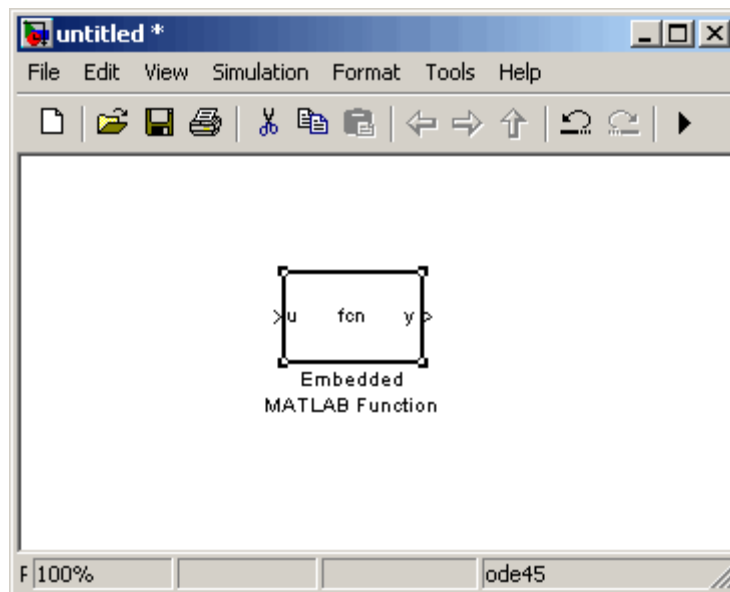
“Building the Function and Checking for Errors” on page 30-15

“Defining Inputs and Outputs” on page 30-19

Adding an Embedded MATLAB Function Block to a Model

Start by creating an empty model and filling it with an Embedded MATLAB Function block, and other blocks necessary to complete the model.

- 1 Create a new model with the Simulink product and add an Embedded MATLAB Function block to it from the User-Defined Function library.



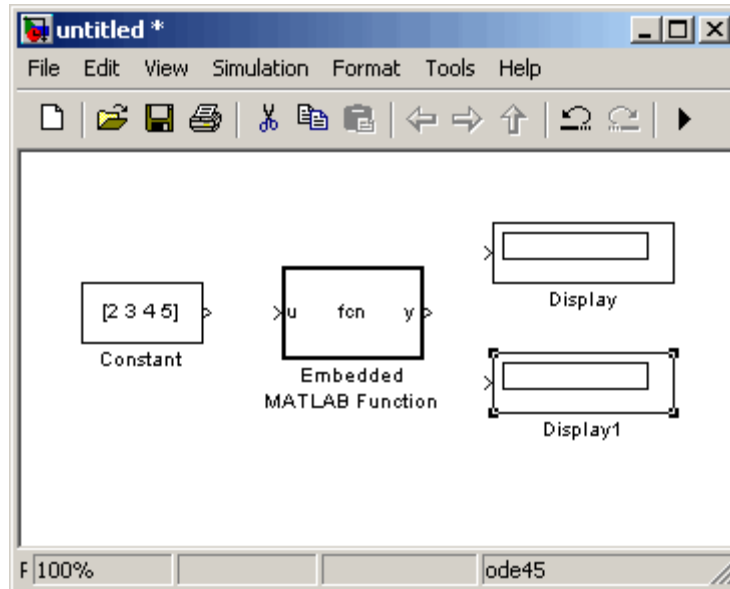
An Embedded MATLAB Function block has two names. The name in the middle of the block is the name of the function you build for the Embedded MATLAB Function block. Its name defaults to `fcn`. The name at the bottom of the block is the name of the block itself. Its name defaults to Embedded MATLAB Function.

The default Embedded MATLAB Function block has an input port and an output port. The input port is associated with the input argument `u`, and the output port is associated with the output argument `y`.

2 Add the following Source and Sink blocks to the model:

- From the Sources library, add a Constant block to the left of the Embedded 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 Embedded MATLAB Function block.

The model should now have the following appearance:



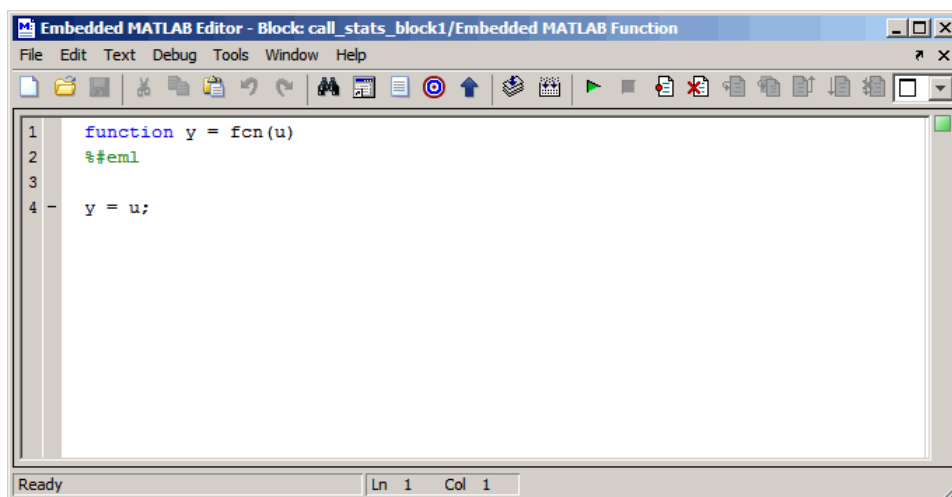
- 3 In the window displayed by the Simulink product, select **File > Save As** and save the model as `call_stats_block1`.

Programming the Embedded MATLAB Function

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 an Embedded MATLAB Function Block to a Model” on page 30-8. Double-click the Embedded MATLAB Function block `fcn` to open it for editing.

The Embedded MATLAB Editor appears.



The Embedded MATLAB Editor window is titled with the syntax *<model name>/<Embedded MATLAB Function block name>* in its header. In this example, the model name is `call_stats_block1`, and the block name is Embedded MATLAB Function, the name that appears at the bottom of the Embedded MATLAB Function block.

Inside the Embedded MATLAB Editor is an edit window for editing the function that specifies the Embedded MATLAB Function block. A function header with the function name `fcn` is at the top of the edit window. The

header specifies an argument to the function, `u`, and a return value, `y`. The `%#eml` compilation directive appears after the function header. The directive declares the function to be Embedded MATLAB compliant. To learn more, see “Adding the Compilation Directive `%#eml`” in the Embedded MATLAB documentation.

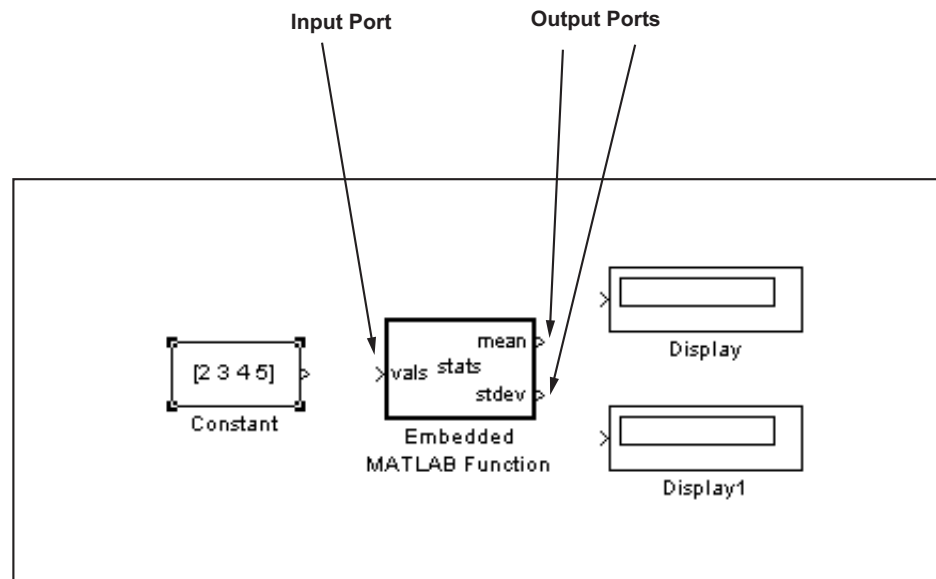
- 2 Edit the function header line with the return values, function name, and argument:

```
function [mean,stdev] = stats(vals)
```

The Embedded MATLAB function `stats` calculates a statistical mean and standard deviation for the values in the vector `vals`. The function header declares `vals` to be an argument to the `stats` function and `mean` and `stdev` to be return values from the function.

- 3 In the Embedded MATLAB Editor, select **File > Save As** and save the model as `call_stats_block2`.

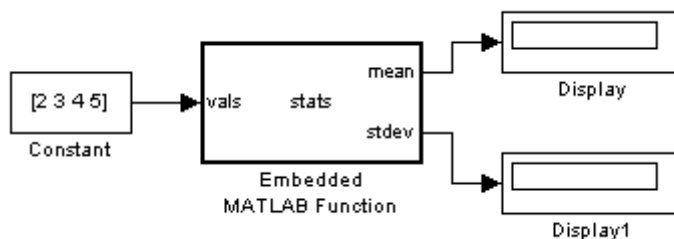
Saving the model updates the model window, which now looks like this:



Changing the function header of the Embedded MATLAB Function block makes the following changes to the Embedded MATLAB Function block in the Simulink model:

- The function name in the middle of the block changes to `stats`.
- The argument `vals` appears as an input port to the block.
- The return values `mean` and `stdev` appear as output ports to the block.

- 4 Complete the connections to the Embedded MATLAB Function block as shown.



- 5 In the Embedded MATLAB Editor, enter a line space after the function header and add the following comment lines:

```
% calculates a statistical mean and a standard
% deviation for the values in vals.
```

You specify comments with a leading percent (%) character, just as you do in MATLAB.

- 6 Enter a line space after the comments and replace the default function line `y = u;` with the following:

```
len = length(vals);
```

The function `length` is an example of a built-in function supported by the runtime function library for Embedded MATLAB Function blocks. This `length` works just like the MATLAB function `length`. It returns the vector length of its argument `vals`. However, when you simulate this model, C code is generated for this function in the simulation application. Callable functions supported for Embedded MATLAB Function blocks are listed

in the topic “Embedded MATLAB Function Library Reference” in the Embedded MATLAB documentation.

The variable `len` is a local variable that is automatically typed as a scalar `double` because the Embedded MATLAB runtime library function, `length`, returns a scalar of type `double`. If you want, you can declare `len` to have a different type and size by changing the way you declare it in the function. To learn more about declaring variables, see “Declaring Variables” in the Embedded MATLAB documentation.

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 persist implicitly declared variables between function calls, see “Declaring Persistent Variables” in the Embedded MATLAB documentation.

- 7** Enter the following lines to calculate the value of `mean` and `stdev`:

```
mean = avg(vals, len);
stdev = sqrt(sum(((vals-avg(vals, len)).^2))/len);
```

`stats` stores the mean and standard deviation values for the values in `vals` in the variable `mean` and `stdev`, which are output by port to the Display blocks in the model. The line that calculates `mean` calls a subfunction, `avg`, that has not been defined yet. The line that calculates `stdev` calls the Embedded MATLAB runtime library functions `sqrt` and `sum`.

- 8** Enter the following line to plot the input values in `vals`.

```
plot(vals, '-+');
```

This line calls the function `plot` to plot the input values sent to `stats` against their vector indices. Because the Embedded MATLAB runtime library has no `plot` function, you need to declare the `plot` function to be extrinsic. An extrinsic function is a function that is executed by MATLAB software during simulation.

- 9** To declare the `plot` function to be extrinsic, add the declaration:

```
eml.extrinsic('plot');
```

after the comments at the top of the Embedded MATLAB function.

See “Calling MATLAB Functions” in the Embedded MATLAB documentation for more details on using this mechanism to call MATLAB functions from Embedded MATLAB functions.

- 10** At the bottom of the Embedded MATLAB function, enter a line space followed by the following lines for the subfunction `avg`, which is called in an earlier line.

```
function mean = avg(array,size)
mean = sum(array)/size;
```

These two lines define the subfunction `avg`. You are free to use subfunctions in Embedded MATLAB function code with single or multiple return values, just as you do in regular MATLAB functions.

Your Embedded MATLAB function should now look like this:

```
function [mean,stdev] = stats(vals)
%#eml

% calculates a statistical mean and a standard
% deviation for the values in vals.

eml.extrinsic('plot');

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;
```

- 11** Save the model as `call_stats_block2`.

Building the Function and Checking for Errors

After programming an Embedded 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 Embedded MATLAB Function block, you must set up your C compiler by running the `mex -setup` command, as described in the documentation for `mex` in the MATLAB Function Reference. 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 “Building MEX-Files” in the MATLAB External Interfaces documentation.

Supported Compilers for Simulation Builds. You can use the following compilers to build models containing Embedded MATLAB Function blocks for simulation:

- Lcc-win32 C 2.4.1
- Microsoft® Visual C++® 2005
- Microsoft Visual C++ .NET 2003
- Microsoft Visual C++ 6.0
- Open WATCOM C++ 1.7


Supported Compilers for Code Generation. To generate code for models that contain Embedded MATLAB Function blocks, you can use any of the C compilers supported by Simulink software for code generation with Real-Time Workshop. 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 Real-Time Workshop.

How to Build the Embedded MATLAB Function

To build the function that calculates the mean and standard deviation:

- Open the `call_stats_block2` model that you saved at the end of “Programming the Embedded MATLAB Function” on page 30-10.
- Double-click its Embedded MATLAB Function block `stats` to open it for editing.
- In the Embedded MATLAB Editor, click the **Build** icon  to compile and build the example model.

If no errors occur, the **Diagnostics Manager** window displays the following message:

```
Parsing successful for machine:model_name
```

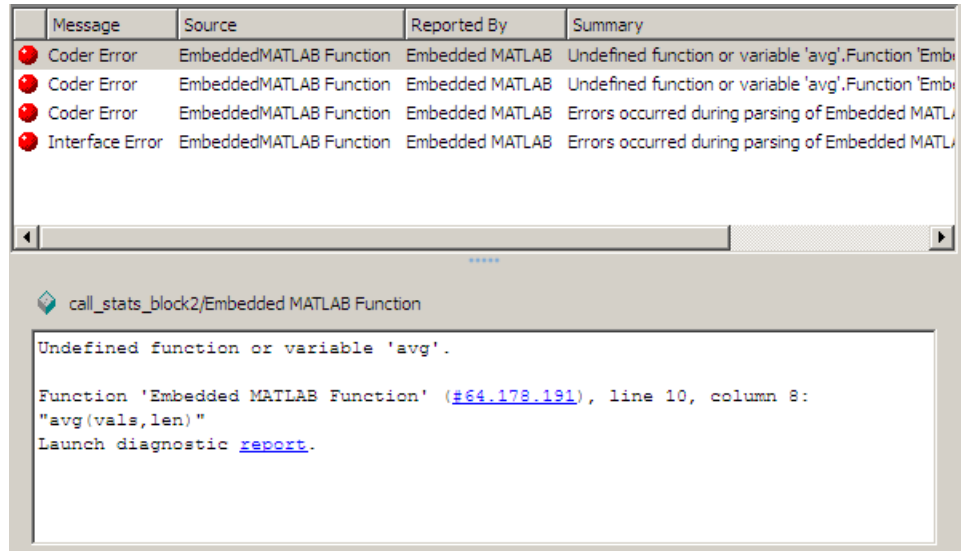
Otherwise, the **Diagnostics Manager** helps you locate errors, as described in “How to Locate and Fix Errors” on page 30-16.

How to Locate and Fix Errors

If errors occur during the build process, the **Diagnostics Manager** window lists the errors with links to the offending code.

The following exercise shows how to locate and fix an error in an Embedded MATLAB Function block:

- 1 In the `stats` function, change the subfunction `avg` to a fictitious subfunction `aug` and then compile to see the following messages in the **Diagnostics Manager** window:



Each detected error appears 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 Embedded MATLAB functions. This information helps you diagnose error messages and understand type propagation rules. For more information about the compilation report, see “Working with Compilation Reports” on page 30-66.

- 3** In the diagnostic message for the selected error, click the blue link after the function name to display the offending code:

Message	Source	Reported By	Summary
Coder Error	EmbeddedMATLAB Function	Embedded MATLAB	Undefined function or variable 'avg'.Function 'Embi
Coder Error	EmbeddedMATLAB Function	Embedded MATLAB	Undefined function or variable 'avg'.Function 'Embi
Coder Error	EmbeddedMATLAB Function	Embedded MATLAB	Errors occurred during parsing of Embedded MATL
Interface Error	EmbeddedMATLAB Function	Embedded MATLAB	Errors occurred during parsing of Embedded MATL

call_stats_block2/Embedded MATLAB Function

```
Undefined function or variable 'avg'.  
  
Function 'Embedded MATLAB Function' (#64.178.191), line 10, column 8:  
"avg(vals,len)"  
Launch diagnostic report.
```

Click to display the offending code
in the Embedded MATLAB Editor

The offending line appears highlighted in the Embedded MATLAB Editor:

```

1  function [mean,stdev] = stats(vals)
2  %#eml
3
4  % calculates a statistical mean and a standard
5  % deviation for the values in vals.
6
7  eml.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;

```

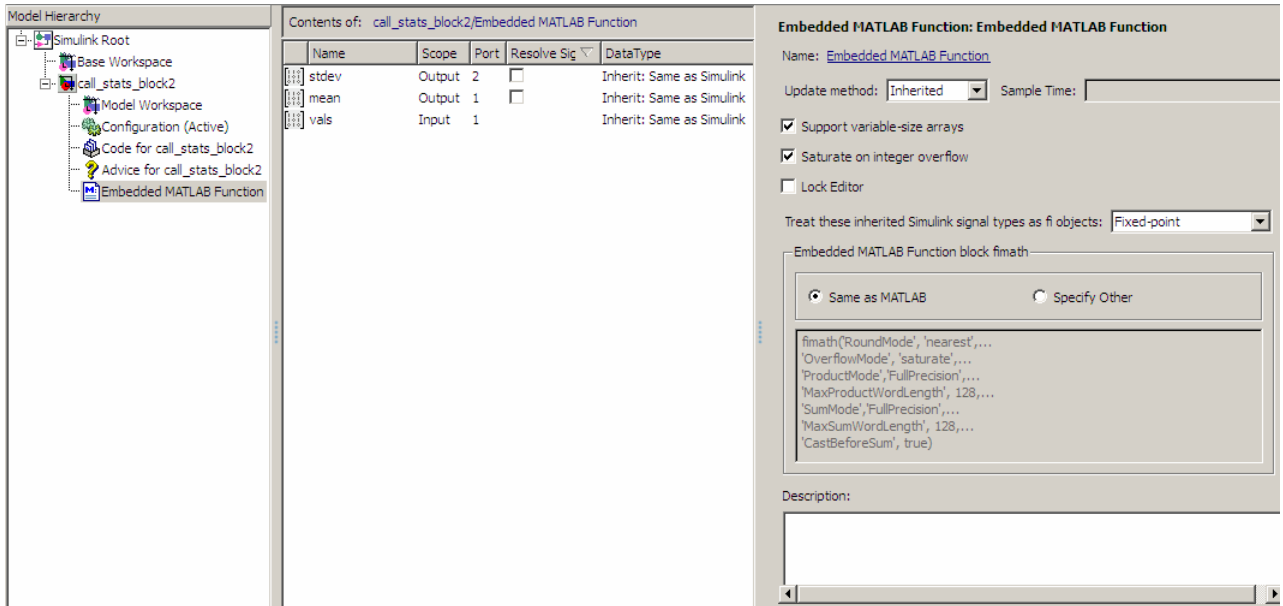
- 4 Correct the error by changing avg back to avg and recompile. No errors are found and the compile completes successfully.

Defining Inputs and Outputs

In the stats function header for the Embedded MATLAB Function block you defined in “Programming the Embedded MATLAB Function” on page 30-10, 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 Embedded 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 Embedded MATLAB Function” on page 30-10. Double-click the Embedded MATLAB Function block `stats` to open it for editing.
- 2 In the Embedded MATLAB Editor, select **Tools > Model Explorer**.

The Model Explorer window opens:

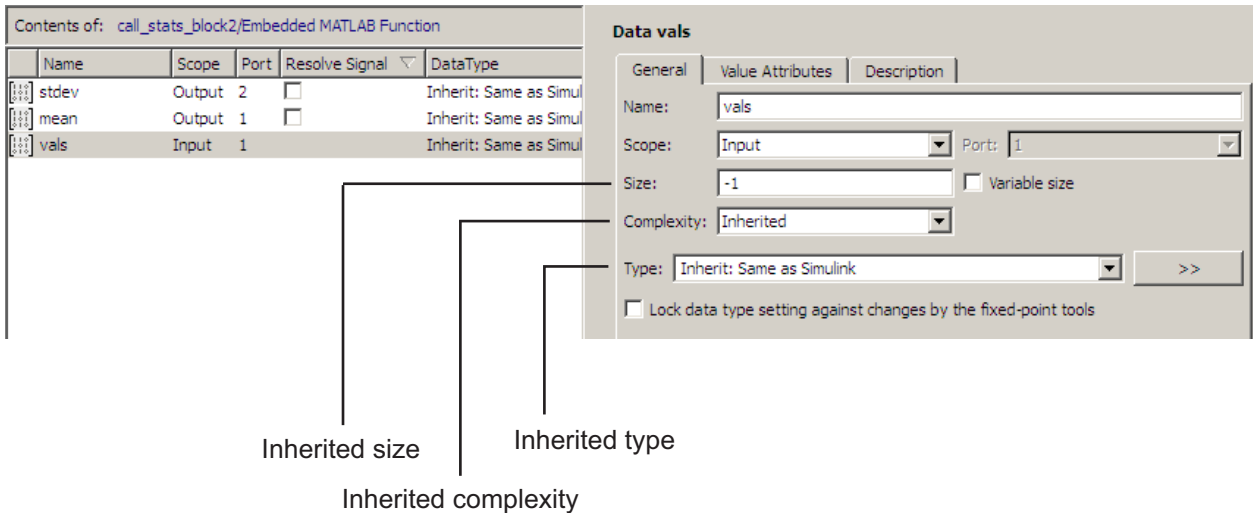


You can use the Model Explorer to define arguments for Embedded MATLAB Function blocks. Notice that the Embedded MATLAB Function block Embedded MATLAB is highlighted in the left **Model Hierarchy** pane.

The **Contents** pane displays the argument **vals** and the return values **mean** and **stdev** that you have already created for the Embedded 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**.

You can also use the Ports and Data Manager to define arguments for Embedded MATLAB Function blocks (see “Ports and Data Manager” on page 30-42).

- 3** In the **Contents** pane of the Model Explorer window, click anywhere in the row for **vals** to highlight it:



The right pane displays the **Data** properties dialog box for **vals**. By default, the type, 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 **Contents** 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 of the **Data** properties dialog box for the argument. For example, you can enter **[2 3]** in the **Size** field to size **vals** as a 2-by-3 matrix. See “Typing Function Arguments” on page 30-81 and “Sizing Function Arguments” on page 30-93 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 an Embedded MATLAB Function block function is 1, just as it would be in MATLAB.

Debugging an Embedded MATLAB Function Block

In this section...

“How Debugging Affects Simulation Speed” on page 30-23

“Enabling and Disabling Debugging” on page 30-23

“Debugging the Function in Simulation” on page 30-24

“Watching Function Variables During Simulation” on page 30-31

“Checking for Data Range Violations” on page 30-34

“Debugging Tools” on page 30-34

How Debugging Affects Simulation Speed

Debugging an Embedded MATLAB function slows simulation speed. If your model has many Embedded MATLAB Function blocks and debugging is enabled, the simulation speed is much slower than when debugging is disabled. For maximum simulation speed, disable debugging as described in “Enabling and Disabling Debugging” on page 30-23.

Enabling and Disabling Debugging

There are two levels of debugging available when using Embedded MATLAB Function blocks, model level debugging and block level debugging.

Debugging is enabled for all Embedded MATLAB functions by default, except for Embedded MATLAB functions in Stateflow. Debugging for Embedded MATLAB functions in Stateflow is controlled in the **Simulation Target** pane in the **Configuration Parameters** dialog.

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 Embedded MATLAB Function block by clicking **Enable Debugging** in the Embedded MATLAB Editor Debug menu. If **Enable Debugging** is unavailable, then the **Simulation Target** pane in the **Configuration Parameters** dialog is controlling debugging.

Debugging the Function in Simulation

In “Creating an Example Embedded MATLAB Function” on page 30-8, you created an example model with an Embedded MATLAB Function block. You use this block to specify an Embedded MATLAB function `stats` that calculates the mean and standard deviation for a set of input values. In this section, you debug `stats` in the example model.

You can debug your Embedded MATLAB Function block just like you can debug a function in MATLAB. In simulation, you test your Embedded MATLAB functions for runtime errors with tools similar to the MATLAB debugging tools. See “Watching with the Command Line Debugger” on page 30-32 and “Debugging Tools” on page 30-34 for more information.

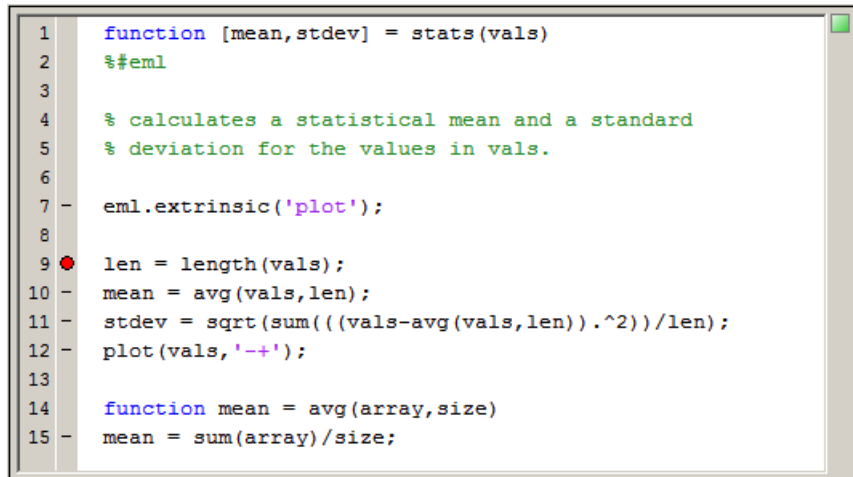
When you start simulation of your model, the Simulink software checks to see if the Embedded MATLAB Function block has been built since creation, or since a change has been made to the block. If not, it performs the build described in “Building the Function and Checking for Errors” on page 30-15. If no diagnostic errors are found, the simulation of your model begins.

Use the following procedure to debug the `stats` Embedded MATLAB function during simulation of the model:

- 1 Open the `call_stats_block2` model that you saved at the end of “Programming the Embedded MATLAB Function” on page 30-10. Double-click its Embedded MATLAB Function block `stats` to open it for editing in the Embedded MATLAB Editor.

- 2 In the Embedded MATLAB Editor, click the dash (-) character in the left margin of the line:

```
len = length(vals);
```



```
1 function [mean,stdev] = stats(vals)
2 %#eml
3
4 % calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 - eml.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;
```

A small red ball appears in the margin of this line, indicating that you have set a breakpoint.

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, as shown.

```
1 function [mean,stdev] = stats(vals)
2 %#eml
3
4 % calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 - eml.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 Click the **Step** icon  to advance execution:


The execution arrow advances to the next line of `stats`. Notice that this line calls the subfunction `avg`. If you click **Step** here, execution advances to the next line, past the execution of the subfunction `avg`. To track execution of the lines in the subfunction `avg`, you need to click the **Step In** icon.

- 5 Click the **Step In** icon 

Execution advances to enter the subfunction avg:

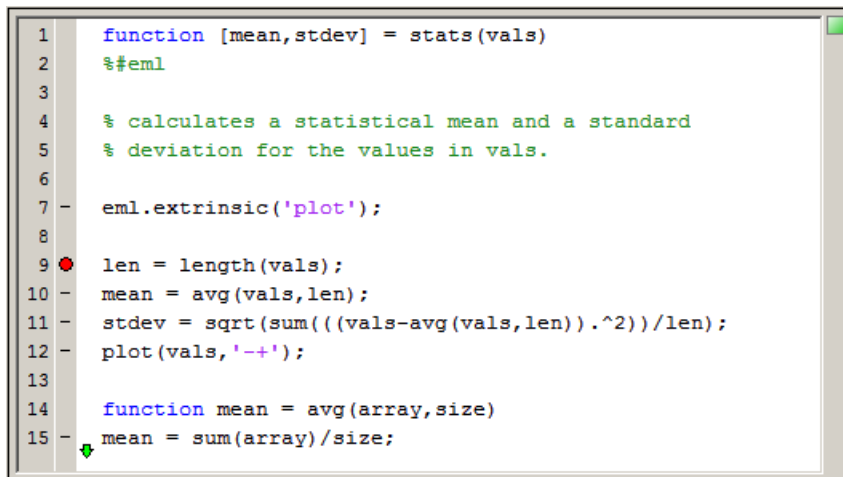
```
1  function [mean,stdev] = stats(vals)
2  %#eml
3
4  % calculates a statistical mean and a standard
5  % deviation for the values in vals.
6
7  -  eml.extrinsic('plot');
8
9  ●  len = length(vals);
10 -  mean = avg(vals,len);
11 -  stdev = sqrt(sum((vals-avg(vals,len)).^2)/len);
12 -  plot(vals,'-+');
13
14  function mean = avg(array,size)
15 -  → mean = sum(array)/size;
```

Once you are in a subfunction, you can use the **Step** or **Step In** icons to advance execution. If the subfunction calls another subfunction, use the **Step In** icon to enter it. If you want to execute the remaining lines of the

subfunction, click the **Step Out** icon .

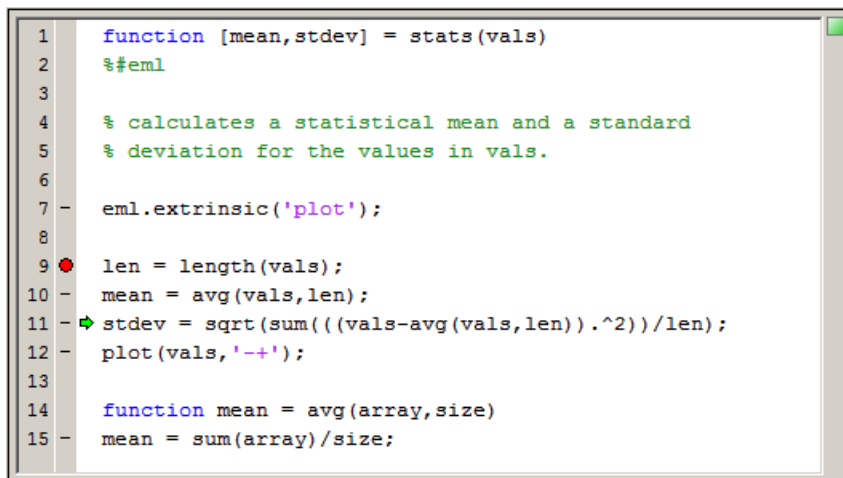
- 6 Click the **Step** icon to execute the only line in the subfunction avg.

The subfunction avg finishes its execution, and you see a green arrow pointing down under its last line as shown.



```
1 function [mean,stdev] = stats(vals)
2 %#eml
3
4 % calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 - eml.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;
```

- 7 Click the **Step** icon to return to the function stats.

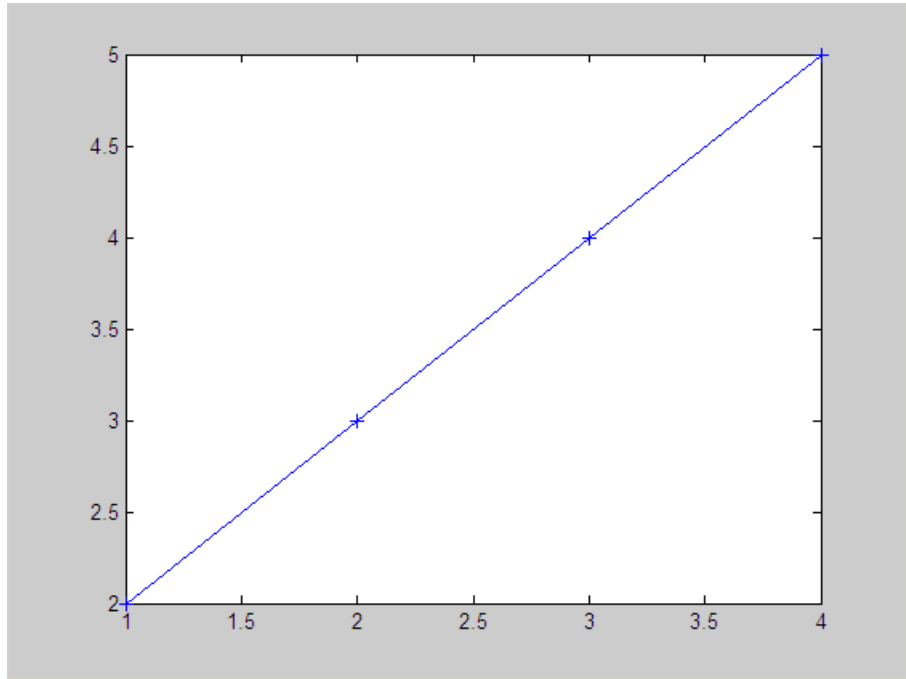


```
1 function [mean,stdev] = stats(vals)
2 %#eml
3
4 % calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 - eml.extrinsic('plot');
8
9 ● len = length(vals);
10 - mean = avg(vals,len);
11 → stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
12 - plot(vals,'-+');
13
14 function mean = avg(array,size)
15 - mean = sum(array)/size;
```

Execution advances to the line after the call to the subfunction avg.

8 Click **Step** twice to calculate the `stdev` and to execute the `plot` function.

The `plot` function executes in MATLAB, and you see the following plot.



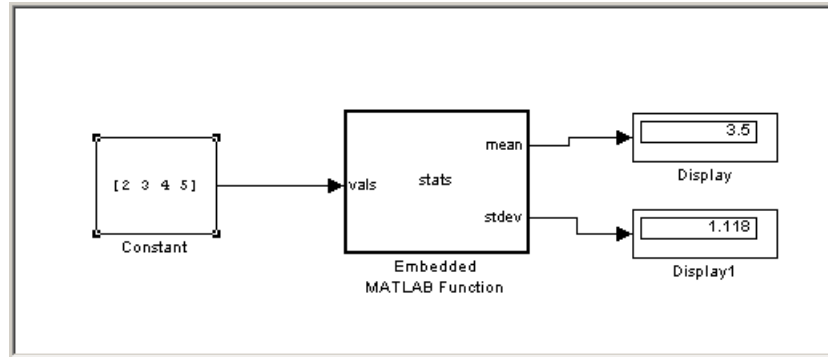
In the Embedded MATLAB Editor, a green arrow points down under the last line of code, indicating the completion of the function stats.


```
1 function [mean,stdev] = stats(vals)
2 %#eml
3
4 % calculates a statistical mean and a standard
5 % deviation for the values in vals.
6
7 - eml.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;
```

- 9 Click the **Continue Debugging** icon  to continue execution of the model.

At any point in a function, you can advance through the execution of the remaining lines of the function with the **Continue Debugging** icon. If you are at the end of the function, clicking the **Step** icon accomplishes the same thing.

The computed values of mean and stdev now appear in the Display blocks.



- 10** In the Embedded MATLAB Editor, click the **Exit Debug Mode** icon  to stop simulation.

Watching Function Variables During Simulation

While you are simulating the function of an Embedded MATLAB Function block, you can use several tools to keep track of variable values in the function. These tools are described in the topics that follow.

Watching with the Interactive Display

To display the value of a variable in the function of an Embedded MATLAB Function block during simulation, in the Embedded MATLAB 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:

```

1  function [mean,stdev] = stats(vals)
2  %#eml
3
4  % calculates a statistical mean and a standard
5  % deviation for the values in vals.
6
7  eml.extrinsic('plot');
8
9  len = length(vals);
10 mean = avg(vals,len);
11 std = sqrt(sum((vals-avg(vals,len)).^2)/len);
12 plot(vals,'-+');
13
14 function mean = avg(array,size)
15 mean = sum(array)/size;

```

The screenshot shows a code editor window with MATLAB code. A red dot is on line 9 at the variable `len`. A yellow tooltip box containing the number `4` is positioned over the `len` variable. A green arrow points from the tooltip down to the text "Display of value for variable len".

Display of value for variable `len`

You can display the value for any variable in the Embedded MATLAB function in this way, no matter where it appears in the function.

Watching with the Command Line Debugger

You can report the values for an Embedded MATLAB 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 Embedded 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 an Embedded MATLAB subfunction.
<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 Embedded 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 an Embedded MATLAB function line by omitting the terminating semicolon. If you do, execution results for the line are echoed to the MATLAB window during simulation.

Checking for Data Range Violations

When you enable debugging, Embedded 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 Model Explorer, select the input or output of interest in the Embedded MATLAB Function block.







The data properties dialog box opens in the Dialog pane of the Model Explorer.





- 2 In the data properties dialog box, select the Value Attributes tab and enter a limit range, as described in “Setting Value Attributes Properties” on page 30-56 .

Debugging Tools

Use the following tools during an Embedded MATLAB function debugging session:

Tool Button	Description	Shortcut Key
 Build	Check for errors and build a simulation application (if no errors are found) for the model containing this Embedded MATLAB function. Alternatively, from the Tools menu, select Build .	Ctrl+B

Tool Button	Description	Shortcut Key
 Start Simulation	Start simulation of the model containing the Embedded MATLAB function.	F5
 Stop Simulation	Stop simulation of the model containing the Embedded MATLAB function. Alternatively, from the Debug menu, select Exit debug mode if execution is paused at a breakpoint.	Shift+F5
 Set/Clear Breakpoint	Set a new breakpoint or clear an existing breakpoint for the selected Embedded MATLAB code line. The presence of the text cursor or highlighted text selects the line. A Breakpoint Indicator  appears in the breakpoints column for the selected line. Alternatively, click the hyphen character (-) in the breakpoints column for the line. A breakpoint indicator appears in place of the hyphen. Click the breakpoint indicator to clear the breakpoint.	F12
 Clear All Breakpoints	Clear all existing breakpoints in the Embedded MATLAB function.	None
 Step	Step through the execution of the next Embedded MATLAB code line. 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. Alternatively, from the Debug menu, select Step .	F10

Tool Button	Description	Shortcut Key
 <p>Step In</p>	<p>Step through the execution of the next Embedded MATLAB code line. If the line calls a subfunction, step into the first line of the subfunction. You can use this tool only after execution has stopped at a breakpoint. Alternatively, from the Debug menu, select Step In.</p>	<p>F11</p>
 <p>Step Out</p>	<p>Step out of line-by-line execution of the current function or subfunction. If in a subfunction, the debugger continues to the line following the call to this subfunction. You can use this tool only after execution has stopped at a breakpoint. Alternatively, from the Debug menu, select Step Out.</p>	<p>Shift+F11</p>
 <p>Continue Debugging</p>	<p>Continue debugging after a pause, such as stopping at a breakpoint. You can use this tool only after execution has stopped at a breakpoint. Alternatively, from the Debug menu, select Continue.</p>	<p>F5</p>
 <p>Exit Debug Mode</p>	<p>Exit debug mode. You can use this tool only after execution has stopped at a breakpoint. Alternatively, from the Debug menu, select Exit Debug Mode.</p>	<p>Shift+F5</p>

Embedded MATLAB Function Editor

In this section...

“Customizing the Embedded MATLAB Editor” on page 30-37

“Embedded MATLAB Editor Tools” on page 30-37

“Editing and Debugging Embedded MATLAB Code” on page 30-38


“Ports and Data Manager” on page 30-42





Customizing the Embedded MATLAB Editor

Use the toolbar icons to customize the appearance of the Embedded MATLAB Editor in the same manner as the MATLAB editor. See “Arranging the Desktop” in the MATLAB documentation.

Embedded MATLAB Editor Tools

The following tools are specific to Embedded MATLAB:

Tool Button	Description
 <p data-bbox="394 1072 609 1098">Edit Data/Ports</p>	<p data-bbox="642 956 1322 1107">Opens the Ports and Data Manager dialog to add or modify arguments for the current Embedded MATLAB Function block (see “Ports and Data Manager” on page 30-42). You can also open this dialog by selecting Edit Data/Ports from the Tools menu.</p> <p data-bbox="642 1133 1322 1255">To define and modify input and output arguments for any Embedded MATLAB Function block in the model hierarchy, use the Model Explorer, which you can open from the Tools menu.</p>

Tool Button	Description
 <p data-bbox="394 392 566 484">Open Compilation Report</p>	<p data-bbox="644 302 1297 394">Opens the compilation report for the Embedded MATLAB Function block. For more information, see “Working with Compilation Reports” on page 30-66.</p>
 <p data-bbox="394 621 547 682">Simulation Target</p>	<p data-bbox="644 505 1317 751">Opens the Simulation Target pane in the Configuration Parameters dialog to enable debugging or include custom code. See “Enabling and Disabling Debugging” on page 30-23 for more information on debugging, and “Including C Functions in Simulation Targets” in the Embedded MATLAB documentation for more information on including custom code.</p>
 <p data-bbox="394 852 605 881">Go To Diagram</p>	<p data-bbox="644 775 1326 868">Displays the Embedded MATLAB function in its native diagram without closing the Embedded MATLAB Editor.</p>
 <p data-bbox="394 973 580 1003">Update Ports</p>	<p data-bbox="644 900 1326 1022">Updates the ports of the Embedded MATLAB Function block with the latest changes made to the function argument and return values without closing the Embedded MATLAB Editor.</p>

See “Defining Inputs and Outputs” on page 30-19 for an example of defining an input argument for an Embedded MATLAB Function block.

Editing and Debugging Embedded MATLAB Code

Commenting Out a Block of Code

To comment out a block of code in the Embedded MATLAB 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 **Comment** from the **Text** menu).

Note To uncomment a block of code, follow the same steps, but select **Uncomment** instead of **Comment**.

For more information, refer to “Adding Comments” in the MATLAB Desktop Tools and Development Environment documentation.

Manual Indenting

To indent a block of code manually:

- 1 Highlight the text that you would like to indent.
- 2 Hold the mouse over the highlighted text and then right-click and select one of the following options from the context menu:
 - **Smart Indent** to indent lines that start with keyword functions or that follow lines containing certain keyword functions. (Alternatively, select **Smart Indent** from the **Text** menu).
 - **Decrease Indent** to move selected lines further to the left. (Alternatively, select **Decrease Indent** from the **Text** menu).
 - **Increase Indent** to move selected lines further to the right. (Alternatively, select **Increase Indent** from the **Text** menu).

For more information, refer to “Indenting” in the MATLAB Desktop Tools and Development Environment documentation.

Opening a Selection

You can open a subfunction, function, file, or variable from within a file in the Embedded MATLAB Function 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 in the MATLAB Desktop Tools and Development Environment documentation

Note If you open an Embedded 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 30-42.

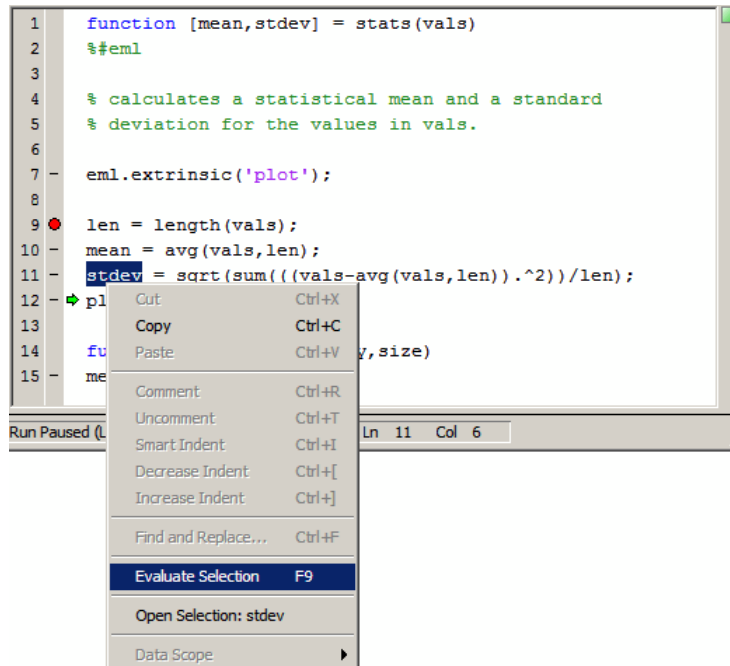
Evaluating a Selection

You can use the **Evaluate a Selection** menu option to report the value for an Embedded 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 an M-file.

Setting Data Scope

To set the data scope of an Embedded 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** from the context menu.
- 3 Select:
 - **Input** if your input data is provided by the Simulink model via an input port to the Embedded 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 30-52 in the Ports and Data Manager documentation.

Ports and Data Manager

The Ports and Data Manager provides a convenient method for defining objects and modifying their properties in an Embedded MATLAB Function block that is open and has focus.

The Ports and Data Manager provides the same data definition capabilities as the Model Explorer, but supports only individual Embedded MATLAB Function blocks. To modify objects and properties for blocks across the model hierarchy, use the “The Model Explorer” on page 19-2.

Ports and Data Manager Dialog

The Ports and Data Manager dialog allows you to add and define data arguments, input triggers, and function call outputs for Embedded MATLAB Function blocks. Using this dialog, you can also modify properties for the Embedded MATLAB Function block and the objects it contains.

The dialog consists of two panes:


- The **Contents** pane lists the objects that have been defined for the Embedded MATLAB Function block.
- The **Dialog** 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, it displays the properties of the Embedded MATLAB Function block.

Opening the Ports and Data Manager



To open the Ports and Data Manager from the Embedded MATLAB Editor,

select **Tools > Edit Data/Ports** or click the **Edit Data/Ports** icon 

The Ports and Data Manager appears for the Embedded 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
 Goto Block Editor	Displays the Embedded MATLAB function in the Embedded MATLAB Editor.
 Show Block Dialog	Displays the default Embedded MATLAB function properties (see “Setting Embedded MATLAB Function Block Properties” on page 30-44). Use this button to return to the settings used by the block after viewing data associated with the block arguments.

Setting Embedded MATLAB Function Block Properties

The **Dialog** pane for an Embedded MATLAB Function block looks like this:

Embedded MATLAB Function: Embedded MATLAB Function

Name: [Embedded MATLAB Function](#)

Update method: Sample Time:

Support variable-size arrays

Saturate on integer overflow

Lock Editor

Treat these inherited Simulink signal types as fi objects:

Embedded MATLAB Function block fimath

Same as MATLAB Specify Other

```

fimath('RoundMode', 'nearest', ...
'OverflowMode', 'saturate', ...
'ProductMode', 'FullPrecision', ...
'MaxProductWordLength', 128, ...
'SumMode', 'FullPrecision', ...
'MaxSumWordLength', 128, ...
'CastBeforeSum', true)

```

Description:

Document link:

This section describes each property of an Embedded MATLAB Function block.

Name. Name of the Embedded MATLAB Function block, following the same naming conventions as for Simulink blocks (see “Manipulating Block Names” on page 7-39).

Update method. Method for activating the Embedded MATLAB Function block. You can choose from the following update methods:

Update Method	Description
Inherited (default)	<p>Input from the Simulink model activates the Embedded MATLAB Function block.</p> <p>If you define an input trigger, the Embedded 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 Embedded 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 Embedded MATLAB Function block samples at the rate of the fastest data input. If you do not define data inputs, the Embedded MATLAB Function block samples as defined by its parent subsystem’s execution behavior.</p>
Discrete	<p>The Embedded 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.</p>
Continuous	<p>The Simulink software wakes up (samples) the Embedded 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.</p>

Saturate on integer overflow. Option that determines how the Embedded MATLAB Function block handles overflow conditions during integer operations:

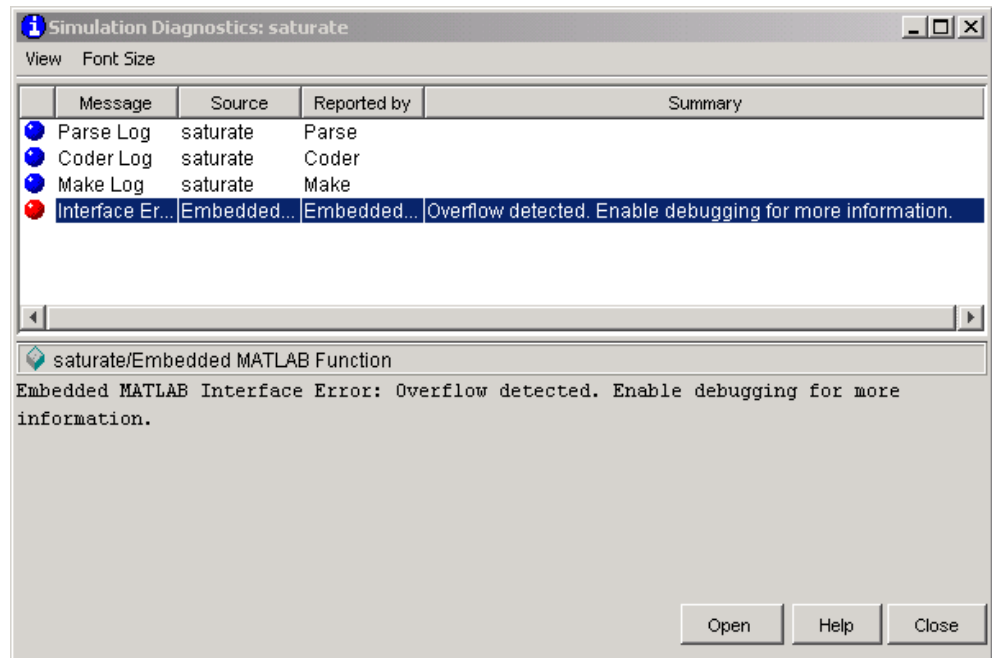
Setting	Action When Overflow Occurs
Enabled (default)	Saturates an integer by setting it to the maximum positive or negative value allowed by the word size. Matches MATLAB behavior.
Disabled	In simulation mode, generates a runtime error. For Real-Time Workshop code generation, the behavior depends on 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**, the Embedded MATLAB Function block adds additional checks in the generated code 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 Embedded MATLAB function 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 enabled debugging for the Embedded MATLAB Function block, the debugger displays the error and lets you examine the data.

If you have not enabled debugging for the Embedded MATLAB Function block, the block generates a runtime error, as in this example:



It is important to note that the code for a Real-Time Workshop target 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 the Real-Time Workshop target.

Lock Editor. Option for locking the Embedded MATLAB Editor. When enabled, this option prevents users from making changes to the Embedded MATLAB Function block.

Treat these inherited Simulink signal types as fi objects. Parameter that applies to Embedded MATLAB Function blocks in models that use fixed-point or integer data types. You can control how the Embedded MATLAB Function block handles Simulink input signals using the **Treat these inherited Simulink signal types as fi objects** parameter.

Property	Description
Treat these inherited Simulink signal types as fi objects	<p>Determines whether to treat inherited fixed-point and integer signals as Fixed-Point Toolbox <code>fi</code> objects.</p> <ul style="list-style-type: none"> • When you select Fixed-point, the Embedded MATLAB Function block treats all fixed-point inputs as Fixed-Point Toolbox <code>fi</code> objects. • When you select Fixed-Point & Integer, the Embedded MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Toolbox <code>fi</code> objects.

Embedded MATLAB Function block `fimath`. Setting that defines `fimath` properties for the Embedded MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the Embedded MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the Embedded MATLAB Function block.

You can select one of the following options for the **Embedded MATLAB Function block `fimath`**.

Setting	Description
Same as MATLAB	<p>When you select this option, the block uses the same <code>fimath</code> properties as the current global <code>fimath</code>. The edit box appears dimmed and displays the current global <code>fimath</code> in read-only form.</p> <p>For more information on the global <code>fimath</code>, see “Working with the Global <code>fimath</code>” in the Fixed-Point Toolbox documentation.</p>
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

Setting	Description
	<p>Embedded MATLAB Function Blocks” in the Fixed-Point Toolbox documentation for more information.</p> <p>For more information on <code>fimath</code> objects, see “Working with <code>fimath</code> Objects” in the Fixed-Point Toolbox documentation.</p>

Description. Description of the Embedded MATLAB Function block.

Document link. Link to documentation for the Embedded MATLAB Function block. To document an Embedded 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 Embedded MATLAB Function block evaluates the expression when you click the blue **Document link** text.

Adding Data to an Embedded MATLAB Function Block


You can define input and output data arguments for an Embedded MATLAB Function block directly in the script, or by using the Ports and Data Manager or Model Explorer. You can use the Ports and Data Manager to add data arguments to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of data arguments in the block.

You can define data arguments for Embedded MATLAB Function blocks in the following methods:

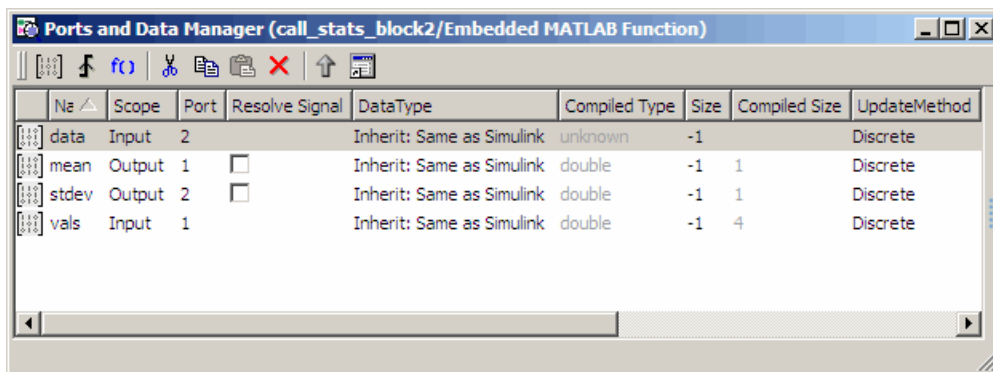
Method	For Defining	Reference
Define data directly in the Embedded MATLAB function script	Input and output data	See “Defining Inputs and Outputs” on page 30-19.

Method	For Defining	Reference
Use the Ports and Data Manager	Input, output, and parameter data in the Embedded MATLAB Function block that is open and has focus	See “Defining Data in the Ports and Data Manager” on page 30-50.
Use the Model Explorer	Input, output, and parameter data in Embedded MATLAB Function blocks at all levels of the model hierarchy	See “Defining Data in the Model Explorer” on page 30-51





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, click the **Add Data** icon .


The Ports and Data Manager adds a default definition of the data to the Embedded MATLAB Function block and displays the new data argument.



- 2 Select the row containing the new data argument.
- 3 Select the data property you want to modify, and specify a new value, as in this example:

	Name	Scope	Port	Resolve Signal	DataType	Compiled Ty
	data	Input	2		Inherit: Same as Simulink	unknown
	mean	Output	1	<input type="checkbox"/>	Inherit: Same as Simulink	unknown
	stdev	Output	2	<input type="checkbox"/>	double	unknown
	vals	Input	1		single	unknown
					int32	
					int16	
					int8	
					uint32	
					uint16	
					uint8	
					boolean	

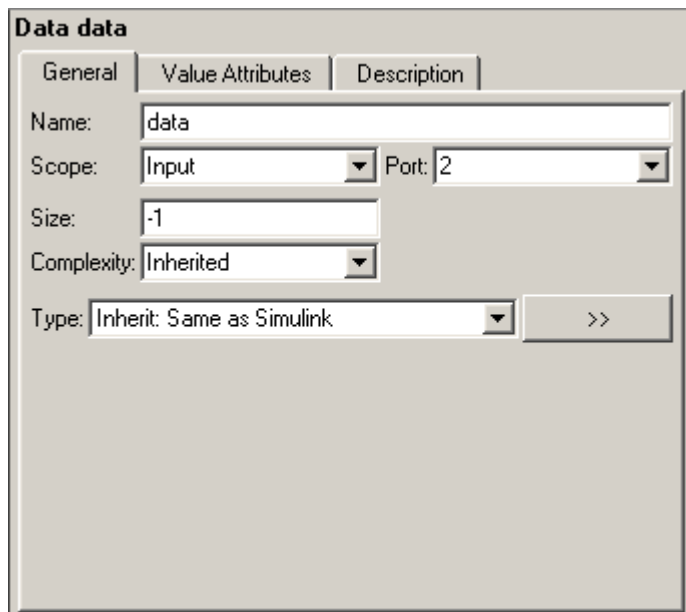
4 Repeat step 3 to specify values for other data properties.

5 Return to the Embedded MATLAB Function block properties at any time by clicking the **Show Block Dialog** icon .

Defining Data in the Model Explorer. The Data properties dialog in the Model Explorer allows you to set and modify the properties of data arguments in Embedded MATLAB Function blocks. Properties vary according to the scope and type of the data object. Therefore, the Data properties dialog is dynamic, displaying only the property fields that are relevant for the data argument you are defining.

Open the Data properties dialog by selecting a data argument in the Contents pane.

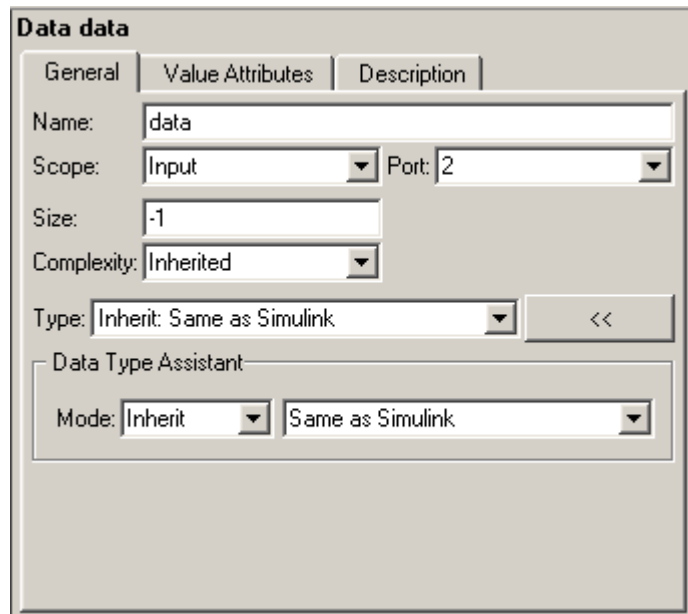
The Data properties dialog provides a set of tabbed panes, as in this example:



Each tab lets you define different features of your data argument:

- The **General** tab lets you define the scope, size, complexity, and type of the data argument. See “Setting General Properties” on page 30-52.
- The **Value Attributes** tab lets you set a limit range and save data argument values. See “Setting Value Attributes Properties” on page 30-56.
- The **Description** tab lets you enter a description and link to documentation about the data argument. See “Setting Description Properties” on page 30-57.

Setting General Properties. The General tab of the Data properties dialog looks like this:

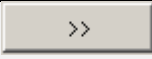


Note If you cannot see the Data Type Assistant, click the Show data type assistant button .

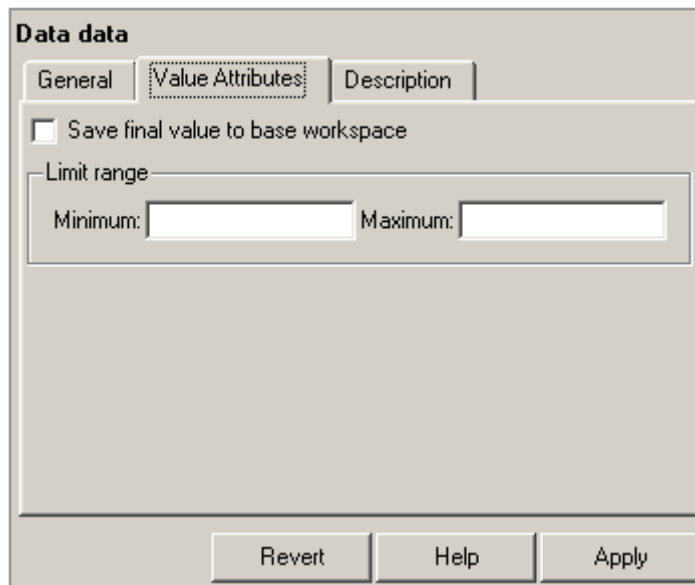
You can set the following properties in the General tab:

Property	Description
Name	Name of the data argument, following the same naming conventions used in MATLAB (see “Naming Variables” in the MATLAB documentation).

Property	Description
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 in the workspace hierarchy is used (see “Using Model Workspaces” on page 4-66). • Input— Data provided by the model via an input port to the Embedded MATLAB Function block. • Output— Data provided by the Embedded MATLAB Function block via an output port to the model. <p>For more information, see “Defining Inputs and Outputs” on page 30-19 and “Parameter Arguments in Embedded MATLAB Functions” on page 30-97.</p>
Port	<p>Index of the port associated with the data argument. This property applies only to input and output data.</p>
Tunable	<p>Indicates whether the parameter used as the source of this data item is tunable (see “Tunable Parameters” on page 2-9). This property applies only to parameter data. You must clear this option if you want to use the parameter where Embedded MATLAB requires a constant expression, such as <code>zeros</code> (see entry for <code>zeros</code> in “Embedded MATLAB Function Library — Alphabetical List” in the Embedded MATLAB documentation).</p>
Data must resolve to Simulink signal object	<p>Specifies that the data argument must resolve to a Simulink signal object. This property applies only to output data. See “Resolving Symbols” on page 4-75 for more information.</p>
Size	<p>Size of the data argument. Size can be a scalar value or a MATLAB vector of values. Size defaults to <code>-1</code>, which means that it is inherited, as described in “Inheriting Argument Sizes from Simulink” on page 30-93. For more details, see “Sizing Function Arguments” on page 30-93.</p>

Property	Description
<p>Complexity</p>	<p>Indicates real or complex data arguments. You can set complexity to one of the following values:</p> <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.
<p>Sampling mode</p>	<p>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:</p> <ul style="list-style-type: none"> • Sample based: Propagate the signal sample by sample (default) • Frame based: Propagate the signal in batches of samples
<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 “Working with Data Types” on page 13-2 in the Simulink® User’s Guide on page 1). • Using the Data Type Assistant to specify a data Mode, then specifying the data type based on that mode. <hr/> <p>Note Click the Show data type assistant button  to display the Data Type Assistant.</p> <hr/> <p>For more information, see “Specifying Argument Types” on page 30-81.</p>

Setting Value Attributes Properties. The **Value Attributes** tab of the **Data** properties dialog looks like this:

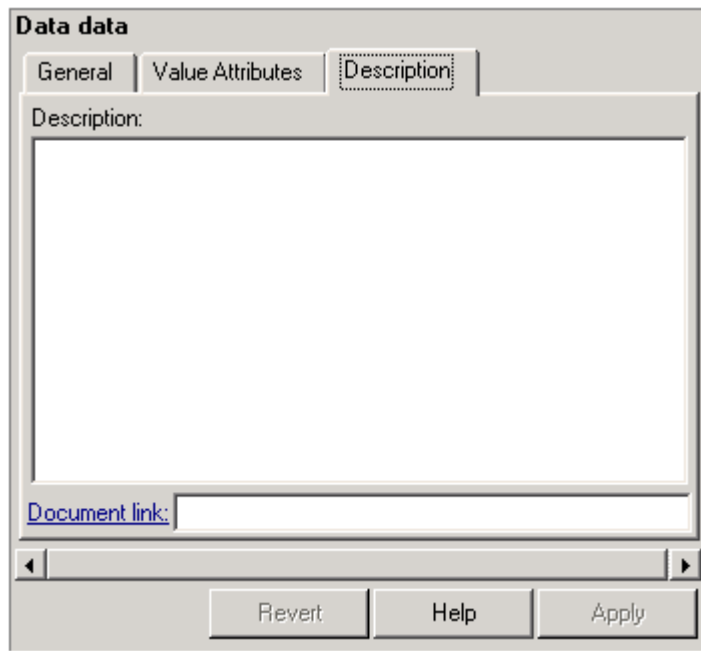


You can set the following properties on the **Value Attributes** tab:

Property	Description
Save final value to base workspace	The Embedded 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.
Limit range properties	Specify the range of acceptable values for input or output data. The Embedded 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. <ul style="list-style-type: none"> • Minimum — The smallest value allowed for the data item during simulation. The default value is <code>-inf</code>.

Property	Description
	<ul style="list-style-type: none">• Maximum — The largest value allowed for the data item during simulation. The default value is <code>inf</code>.

Setting Description Properties. The **Description** tab of the Data properties dialog looks like this:



You can set the following properties on the Description tab:

Property	Description
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 Embedded MATLAB Function block evaluates the link and displays the documentation.

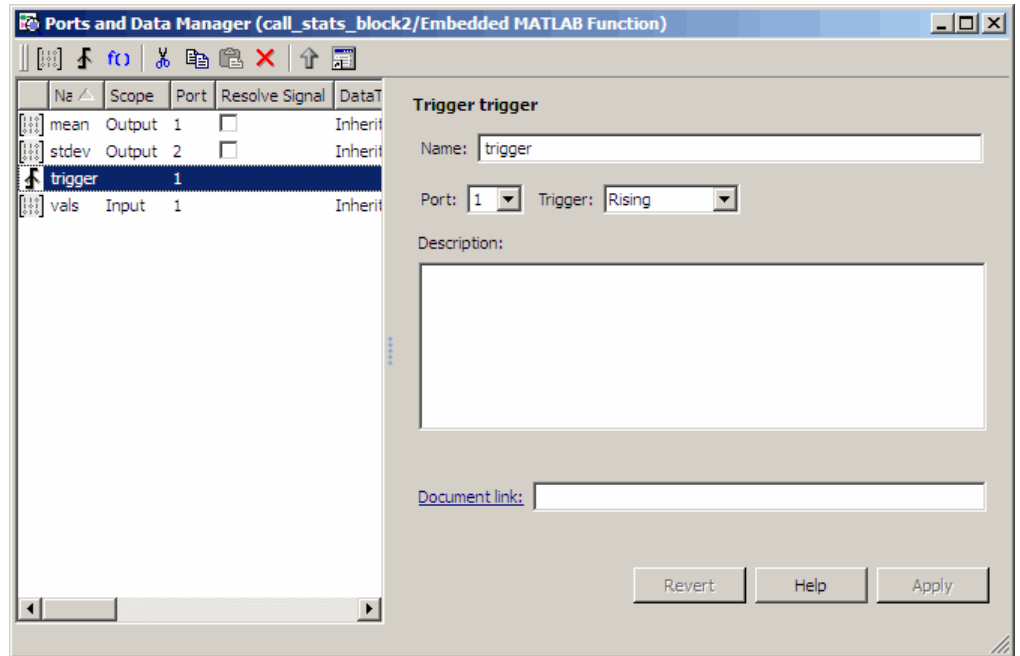
Adding Input Triggers to an Embedded MATLAB Function Block

You can use the Ports and Data Manager to add input triggers to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of input triggers in the block.

To add an input trigger and modify its properties, follow these steps:

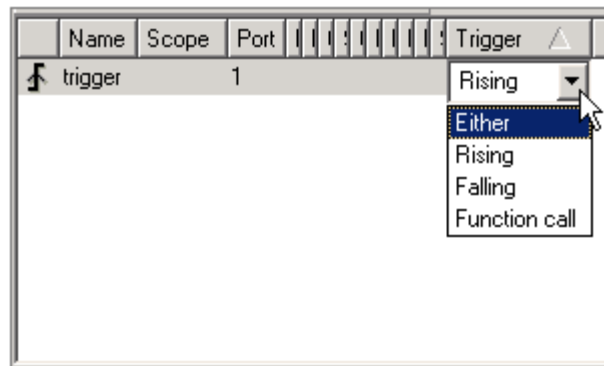
- 1 In the Ports and Data Manager, click the **Add Input Trigger** icon .


The Ports and Data Manager adds a default definition of the new input trigger to the Embedded MATLAB Function block and displays the Trigger properties dialog.



2 Modify properties for the new input trigger, using one of the following methods:

- In the Contents pane, select the row that contains the input trigger you want to modify and then edit the property of interest, as in this example:



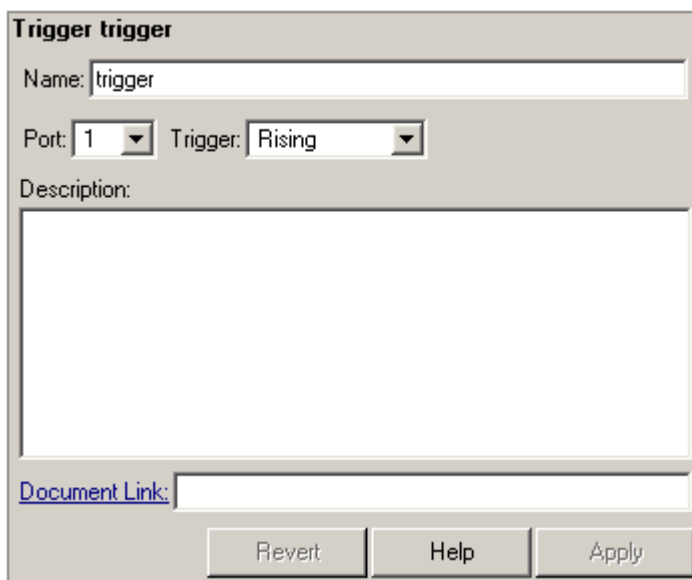
- Modify fields in the Trigger properties dialog, as described in “The Trigger Properties Dialog” on page 30-60.
- Return to the Embedded MATLAB Function block properties at any time by clicking the **Show Block Dialog** icon .

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 Embedded MATLAB Function blocks.

You can open the Trigger properties dialog using one of these methods:

- Select an input trigger in the Contents pane of the Ports and Data Manager to open the Trigger properties dialog in the Dialog pane.
- Right-click an input trigger in the Contents pane and select **Properties** from the submenu to open the Trigger properties dialog outside the Ports and Data Manager.

The Trigger properties dialog looks like this:



The screenshot shows a dialog box titled "Trigger trigger". It contains the following fields and controls:

- Name:** A text input field containing the text "trigger".
- Port:** A dropdown menu with "1" selected.
- Trigger:** A dropdown menu with "Rising" selected.
- Description:** A large, empty text area.
- Document Link:** A text input field with a blue hyperlink label "Document Link:" to its left.
- Buttons:** Three buttons at the bottom: "Revert", "Help", and "Apply".


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 (see “Naming Variables” in the MATLAB documentation).
Port	Index of the port associated with the input trigger. The default value is 1.
Trigger	Type of event that triggers execution of the Embedded 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 Embedded 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 Embedded 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 Embedded MATLAB Function block when the control signal is either rising or falling. • Function call— Triggers execution of the Embedded 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 Embedded MATLAB Function block evaluates the link and displays the documentation.

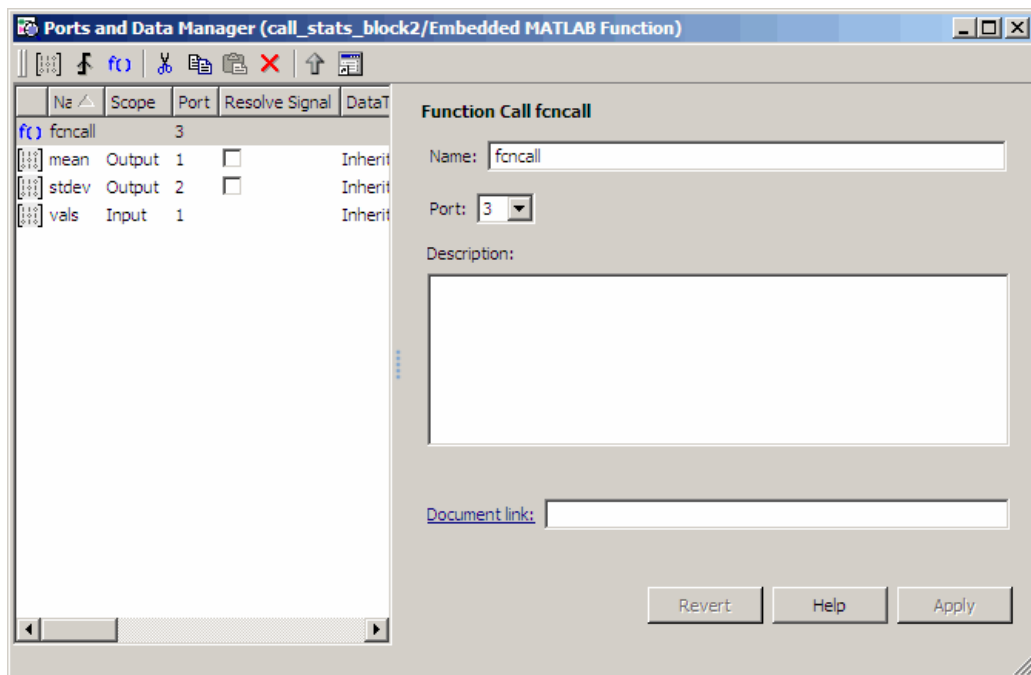
Adding Function Call Outputs to an Embedded MATLAB Function Block

You can use the Ports and Data Manager to add and modify function call outputs to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of function call outputs in the block.

To add a function call output and modify its properties, follow these steps:

- 1 In the Ports and Data Manager, click the **Add Function Call Output** icon .


The Ports and Data Manager adds a default definition of the new function call output to the Embedded MATLAB Function block and displays the Function Call properties dialog.



- 2 Modify properties for the new function call output, using one of the following methods:

- In the Contents pane, select the row that contains the function call output you want to modify and then edit the property of interest, as in this example:

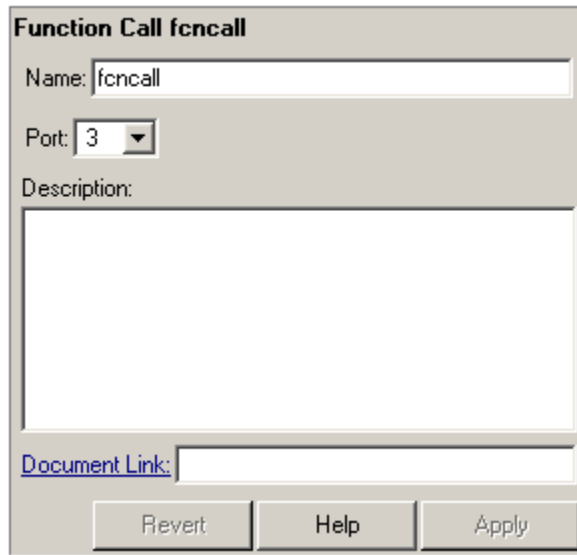
	Name	Scope	Port	Data Type	Mode	Data Type
f()	fcnca1		1			

- Modify fields in the Function Call properties dialog, as described in “The Function Call Properties Dialog” on page 30-63.
- Return to the Embedded MATLAB Function block properties at any time by clicking the **Show Block Dialog** icon .

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 Embedded MATLAB Function blocks.

You can open the Function Call properties dialog in the Dialog pane by selecting a function call output in the Contents pane of the Ports and Data Manager.

The Function Call properties dialog looks like this:



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 (see “Naming Variables” in the MATLAB documentation).
Port	Index of the port associated with the function call output. The default value is 3.

Property	Description
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 Embedded MATLAB Function block evaluates the link and displays the documentation.

Working with Compilation Reports

In this section...

- “About Compilation Reports” on page 30-66
- “Location of Compilation Reports” on page 30-67
- “Opening Compilation Reports” on page 30-67
- “Description of Compilation Reports” on page 30-68
- “Viewing Your Embedded MATLAB Function Code” on page 30-69
- “Viewing Call Stack Information” on page 30-70
- “Viewing the Compilation Summary Information” on page 30-71
- “Viewing Error and Warning Messages” on page 30-71
- “Viewing Variables in Your M-Code” on page 30-73
- “Keyboard Shortcuts for the Compilation Report” on page 30-77
- “Compilation Report Limitations” on page 30-78

About Compilation Reports

Whenever you build a Simulink model that contains Embedded MATLAB Function blocks, Simulink automatically generates a compilation report in HTML format for each Embedded MATLAB Function block in your model. The report helps you debug your Embedded MATLAB functions and verify compliance with the Embedded MATLAB subset. The report provides links to your Embedded 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 There is one compilation report for each Stateflow chart, regardless of the number of Embedded MATLAB functions it contains.


Location of Compilation Reports

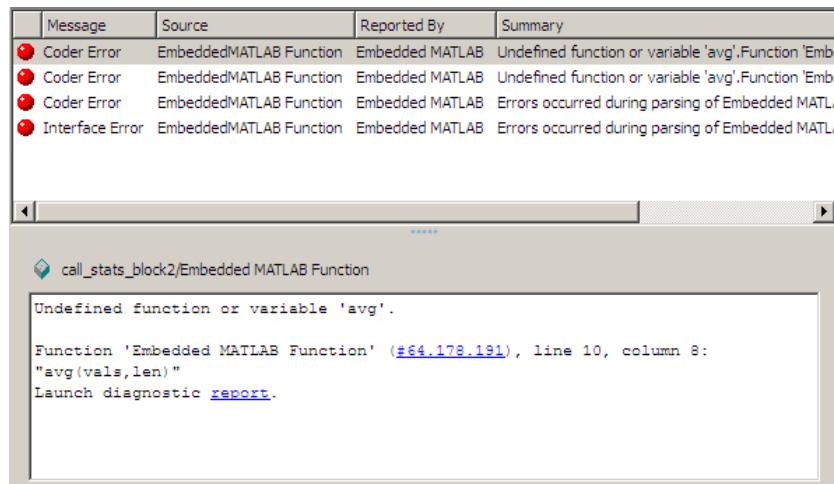
Embedded MATLAB provides a compilation report for each Embedded MATLAB Function block in the model `model_name` at the following location:

```
slprj/_sfprj/  
model_name/_self/  
sfun/html/
```

Opening Compilation Reports

Open the compilation report using one of the following methods:

- Click the **Open compilation report**  in the Embedded MATLAB Editor.
- Select **Tools > Open Compilation Report** from the Embedded MATLAB Editor toolbar.
- Click the **report** link in the **Diagnostics Manager** window if compilation errors occur.



Description of Compilation Reports

When you build the Embedded MATLAB function, Embedded MATLAB generates an HTML report. The following example shows a successful compilation:

M-code pane: displays M-code for function selected on M-code tab

The screenshot shows the Embedded MATLAB compilation report interface. On the left, the 'M-code' tab is selected, displaying a list of functions with checkboxes and filter options. The 'call_stats' function is selected. On the right, the M-code for the selected function is displayed, showing the function definition for 'call_stats' with line numbers 1 through 7. Below the M-code, a summary pane shows the C source code generated on 14-May-2009 11:48:03, with 0 errors and 0 warnings.

The report provides the following information, as applicable:

- M-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 Embedded MATLAB function

Viewing Your Embedded MATLAB Function Code

To view your Embedded MATLAB function code, click the **M-code** tab. The compilation report displays the M-code for the function highlighted in the list on this tab.

M-code for function selected on M-code tab

The screenshot shows the MATLAB IDE interface. On the left, the 'M-code' tab is active, displaying a list of functions. The function 'call_stats > avg' is selected and highlighted in blue. The right pane shows the M-code for this function, which is:





```

  9 function mean = avg(array, size)
 10 mean = sum(array)/size;
  
```

 A blue arrow points from the text 'M-code for function selected on M-code tab' to the function name 'call_stats > avg' in the code. Below the code, there is a 'Summary' section with the following information:

Summary	All Messages (0)	Variables
C source code generated on: 15-May-2009 09:55:33		
Number of errors:	0	
Number of warnings:	0	

The **M-code** tab provides:

- A list of the Embedded 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, **Filter function list by attributes**, that you can use to sort your functions by:
 - Size
 - Complexity

- Class

Viewing Subfunctions

The compilation report annotates the subfunction with the name of the parent function in the list of functions on the **M-code** tab.

For example, if the M-functions `fcn1` contains the subfunction `subfcn` and `fcn2` contains the subfunction `subfcn2`, the report displays:

```
fcn1 > subfcn1  
fcn2 > subfcn2
```

Viewing Specializations

If your Embedded MATLAB function calls the same function with different types of inputs, the compilation report numbers each of these **specializations** in the list of functions on the **M-code** tab.

For example, if the function `fcn` calls the function `subfcn` with different types of inputs:

```
function y = fcn(u) %#eml  
% Specializations  
y = y + subfcn(single(u));  
y = y + subfcn(double(u));
```

The compilation report numbers the specializations in the list of functions.

```
fcn > subfcn > 1  
fcn > subfcn > 2
```

Viewing Call Stack Information

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 subfunctions that each function calls.

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 compilation 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 compilation report provides information about warnings and errors by:

- Listing all errors and warnings in the **All Messages** tab. The report lists these messages in chronological order.
- Highlighting all errors and warnings in the M-code pane

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. The compilation report marks messages as follows:



Error



Warning

To locate the offending line of code for an error or warning in the list, click the message in the list. The compilation report highlights errors in the list and M-code in red and warnings in orange. Click the blue line number next to the offending line of code in the M-code window to go to the error in the source file.

Note You can only fix errors in the source file.

Compilation report outlines selected error in black and moves offending line of code to top of screen

Compilation report highlights selected error in list

```

Function: call_stats (call_stats/Embedded MATLAB Function)
1  function [mean, stdev] = call_stats(vals) %#eml
2  eml.extrinsic('plot');
3
4  len = length('plot');
5  mean = avg(vals, len);
6  stdev = sqrt(sum(((vals-avg(vals, len)).^2))/len);
7  plot(vals, '-+');
    
```

Order	Type	Function	Location	Description
1	✘	call_stats	Line 0005	Undefined function or variable 'avg'.
2	✘	call_stats	Line 0006	Undefined function or variable 'avg'.

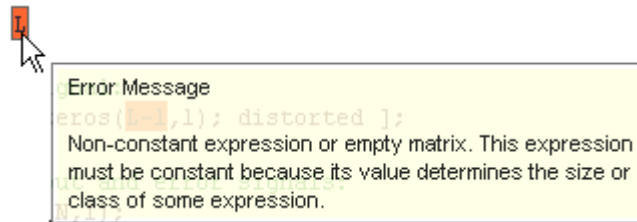
Click the blue line number next to the line of code with the error to go to the offending code in the source file

Viewing Error and Warning Information in Your M-Code

If errors or warnings occurred during compilation, the compilation report highlights them in your M-code.

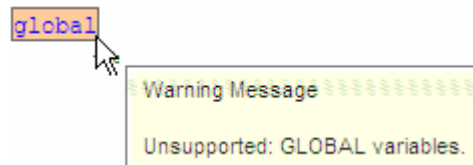
To learn more about a particular error or warning, place your pointer over the highlighted text. The compilation report provides more information as follows:

- The report highlights errors in red and provides an error message.



You can click the error to go to the message in the **All Messages** tab.

- The report highlights warnings in orange and provides a warning message.



You can click the warning to go to the message in the **All Messages** tab.

Viewing Variables in Your M-Code

The compilation report provides compile-time type information for the variables and expressions in your M-code, including name, type, size, complexity, and class. The report 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 Embedded MATLAB function by:

- Viewing the list in the **Variables** tab
- Placing your pointer over the variable name in your M-code

Viewing Variable Information in the Variables Tab

To view a list of all the variables in your M-function, click the **Variables** tab. The compilation report displays a complete list of variables in the order they appear in the M-function selected in the **M-code** tab. Clicking a variable in

the list highlights all instances of that variable, and scrolls the M-code panel so that the first instance is in view.

Tip You can sort the variables by clicking the column headings in the **Variables** tab. To sort the variables by multiple columns, hold down the Shift key when clicking the column headings.

The report provides the following information about each variable, as applicable.

- Name
- Type
- Size
- Complexity
- Class
- DataTypeMode (DT mode) — for fixed-point data types only.
- 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

The report 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 any fixed-point data types, the report does not display the DT mode, WL or FL columns.

Note For more information on viewing fixed-point data types, see “Working with Fixed-Point Compilation Reports” in the Fixed-Point Toolbox documentation.

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 `:`.

Variable	Type	Size	Complex	Class
out	Output variable	:5x:5	No	double
out2	Output variable	:4x2	No	double
in	Input variable	1x1	No	double
in2	Input variable	:4x:4	No	double
t1	Local variable	:3x:3	No	double

The size of the variable `in` is fixed at '1 x 1'.

The array `in2` is variable-sized. Its maximum computed size is '4 x 4'.

If Embedded MATLAB technology cannot compute the maximum size of a variable-sized array, the compilation report displays the size as `?:?`.

Order	Type	Function	Location	Description
1	✘	my_emidemo_lms_01_mlint	Line 0047	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [? x 1].
2	✘	my_emidemo_lms_01_mlint	Line 0050	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [? x 1].
3	✘	my_emidemo_lms_01_mlint	Line 0058	Computed maximum size is not bounded. Static memory allocation requires all sizes to be bounded. The computed size is [? x 1].

'?:?' denotes that the computed maximum size is not bounded

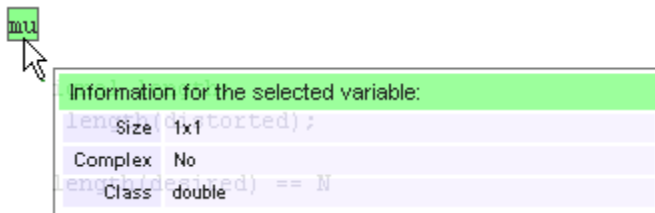
For more information on how to use the size information for variable-sized arrays, see [Representing Data that Varies in Size at Runtime](#) in the Embedded MATLAB documentation.

Viewing Information about Variables and Expressions in Your Embedded MATLAB Function Code

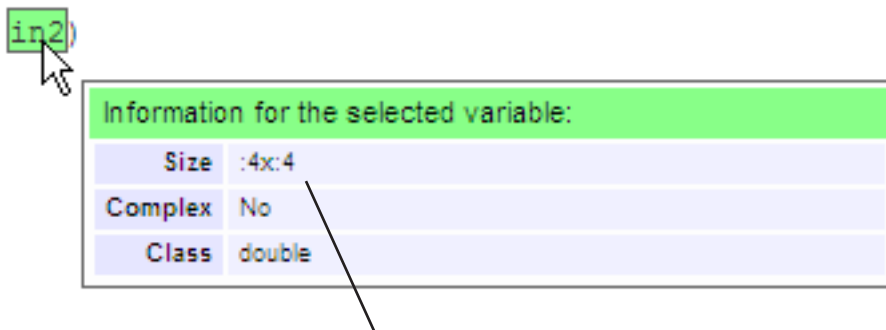
To view information about a particular variable or expression in your Embedded MATLAB function code, place your pointer over the variable name

or expression in the M-code pane. The compilation report highlights variables and expressions in different colors:

Green, when the variable has data type information at this location in the code.

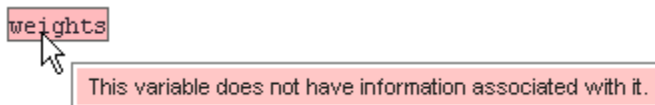


For variable-sized arrays, the size field includes information on the computed maximum size of the array. The size of each array dimension that varies is prefixed with a colon :.

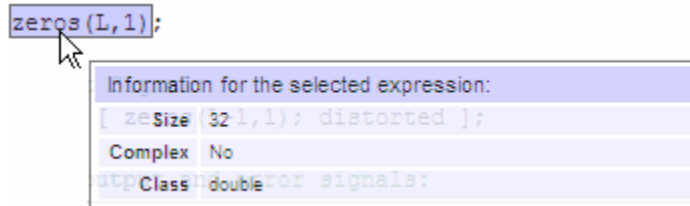


The array in2 is variable-sized. Its maximum computed size is '4 x 4'.

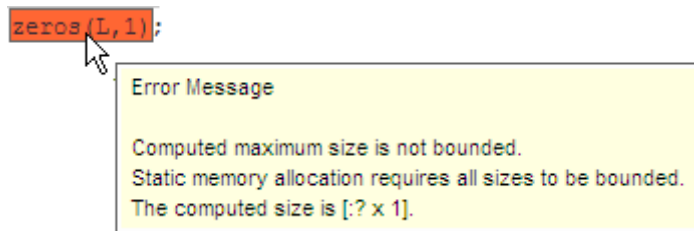
Pink, when the variable has no data type information.



Purple, information about expressions. You can also view information about expressions in your M-code. Place your pointer over an expression in the M-code pane. The compilation report highlights expressions in purple and provides more detailed information.



Red, when there is error information for an expression. If the Embedded MATLAB software cannot compute the maximum size of a variable-sized array, the compilation report provides error information.



Keyboard Shortcuts for the Compilation Report

You can use the following keyboard shortcuts to navigate between the different panes in the compilation report. Once you have selected a pane, use the Tab key to advance through data in that pane.

To select ...	Use...
M-Code Tab	Ctrl+m
Call Stack Tab	Ctrl+k
M-Code Pane	Ctrl+w
Summary Tab	Ctrl+s

To select ...	Use...
All Messages Tab	Ctrl+a
Variables Tab	Ctrl+v

Compilation Report Limitations

The compilation report displays information about the variables and expressions in your Embedded MATLAB function, with the following limitations:

varargin and varargout

The report does not support varargin and varargout arrays.

Loop Unrolling

The report does not display the correct information for unrolled loops.

Dead Code

The report does not display information about any dead code.

Structures

The report does not provide complete information about structures.

- You cannot expand a **struct** in the list on the **Variables** tab to view all its fields
- In the **M-code** pane, the report does not provide information about all structure fields in the `struct()` constructor
- In the **M-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 down through the list of variables, the report does not display the column headings on the **Variables** tab.

Comments

The report does not display comments that are:

- Before the primary function declaration. For example, the report does not display the comment `%Testing comments` in the following M-code.

```
%Testing comments

function test_comments    %#eml

test_comments2;

function test_comments2
```

- Between two functions in a M-file. For example, the report does not display the comment `%Testing comments` in the following M-code.

```
function test_comments    %#eml

test_comments2;

%Testing comments

function test_comments2
```

Multiline Matrices

In the **M-code** pane, the report does not support selection of multiline matrices. It only supports 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];
```

Typing Function Arguments

In this section...

“About Function Arguments” on page 30-81

“Specifying Argument Types” on page 30-81

“Inheriting Argument Data Types” on page 30-84

“Built-In Data Types for Arguments” on page 30-86

“Specifying Argument Types with Expressions” on page 30-86

“Specifying Simulink® Fixed Point Data Properties” on page 30-87

About Function Arguments

You create function arguments for an Embedded MATLAB Function block by entering them in its function header in the Embedded MATLAB Editor. When you define arguments, the Simulink software creates corresponding ports on the Embedded MATLAB Function block that you can attach to signals. You can select a *data type mode* for each argument that you define for an Embedded MATLAB Function block. Each data type mode presents its own set of options for selecting a *data type*.

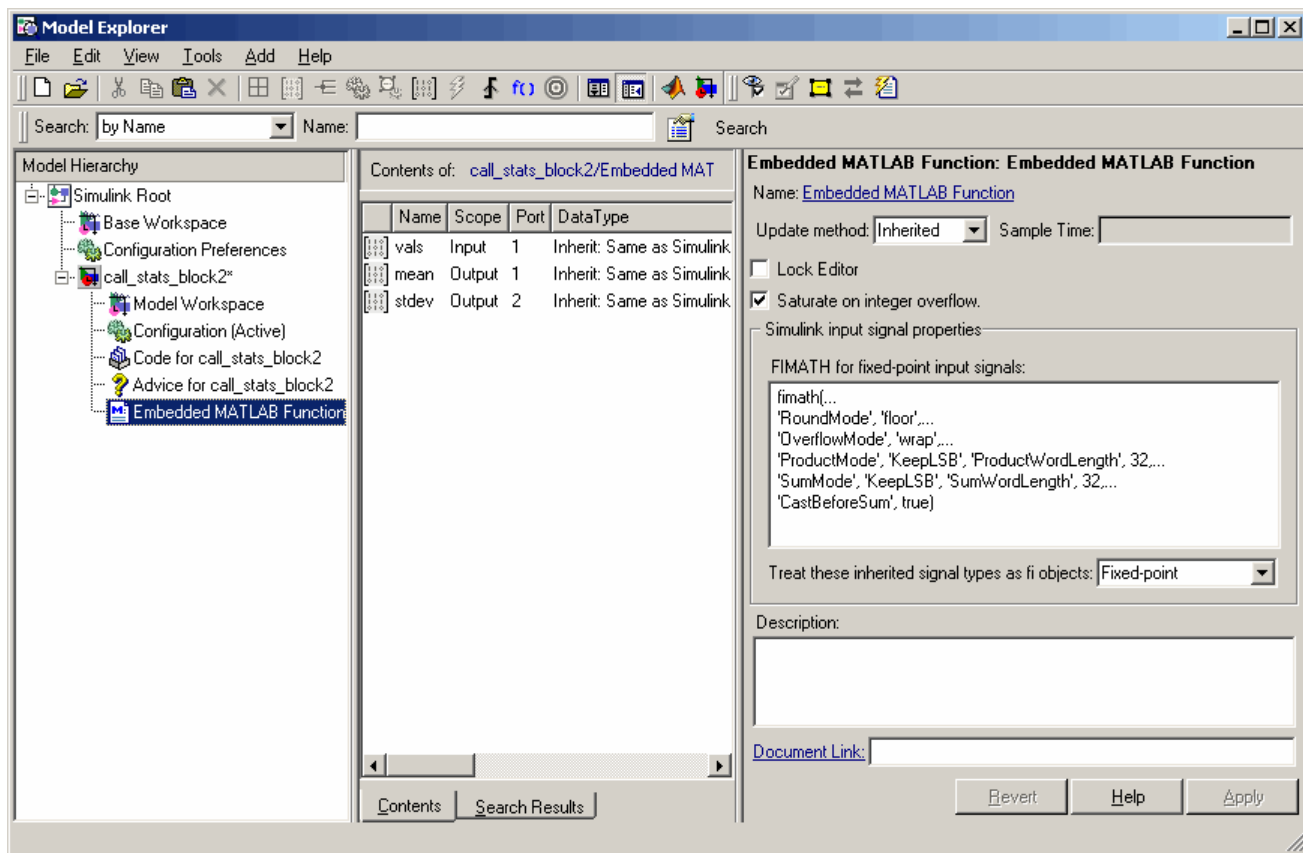
By default, the data type mode for Embedded MATLAB function arguments is **Inherited**. This means that the function argument inherits its data type from the incoming or outgoing signal. To override the default type, you first choose a data type mode and then select a data type based on the mode. The following procedure describes how to use the Model Explorer to set data types for function arguments. You can also use the Ports and Data Manager tool (see “Ports and Data Manager” on page 30-42).


Specifying Argument Types

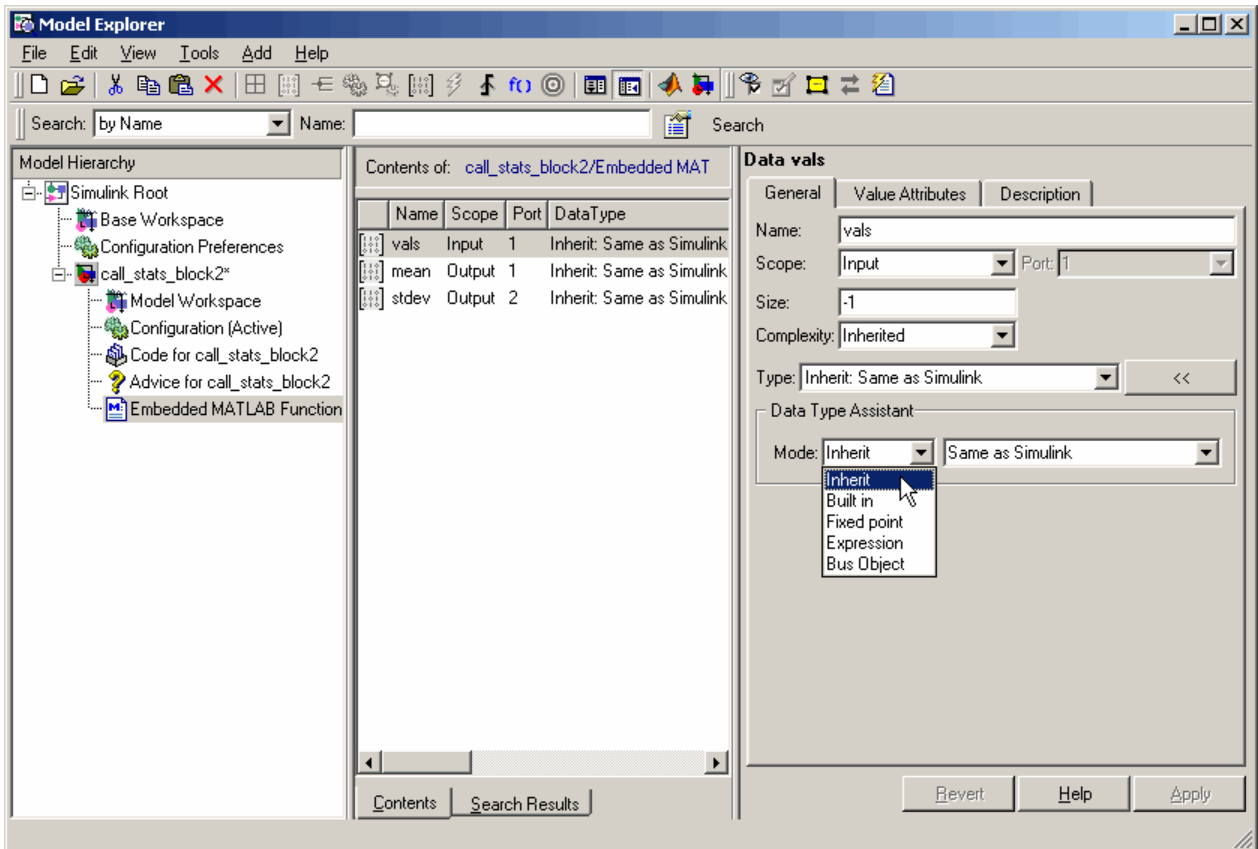
To specify the type of an Embedded MATLAB function argument:

- 1 From the Embedded MATLAB Editor, select **Tools > Model Explorer**.

Model Explorer appears with the Embedded MATLAB Function block highlighted in the **Model Hierarchy** pane.



- 2 In the **Contents** pane (in the middle), click the row containing the argument of interest.
- 3 In the **Data** properties dialog (on the right), click the Show data type assistant button  to display the Data Type Assistant. Then, choose an option from the **Mode** drop-down menu, as shown:



The **Data** properties dialog 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 Embedded MATLAB 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 30-84.</p>
Built in	In the Data type field, select from the drop-down list of supported data types, as described in “Built-In Data Types for Arguments” on page 30-86.
Fixed point	Specify the fixed-point data properties as described in “Specifying Simulink® Fixed Point Data Properties” on page 30-87.
Expression	Enter an expression that evaluates to a data type, as described in “Specifying Argument Types with Expressions” on page 30-86.
Bus Object	<p>In the Bus object field, enter the name of a <code>Simulink.Bus</code> object to define the properties of an Embedded MATLAB structure. You must define the bus object in the base workspace. See “Working with Structures and Bus Signals” on page 30-100.</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 “Using the Bus Editor” on page 12-13 in the Simulink User’s Guide).</p> <hr/>

Inheriting Argument Data Types

Embedded MATLAB function arguments can inherit their data types, including fixed point types, from the signals to which they are connected.

Select the argument of interest in the Contents pane of the Model Explorer or Ports and Data Manager, and set data type mode using one of these methods:

- In the **Data** properties dialog, select **Inherit: Same as Simulink** from the **Type** drop-down menu.
- In the Contents pane, set the **Data Type** column to **Inherit: Same as Simulink**.

See “Built-In Data Types for Arguments” on page 30-86 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**.

Once you build the model, the **Compiled Type** column of the Model Explorer or Ports and Data Manager gives the actual type used in the compiled simulation application.

In the following figure, an Embedded MATLAB Function block argument inherits its data type from an input signal of type double:

Contents of: call_stats_block2/Embedded MATLAB Function					
	Name	Scope	Port	DataType	Compiled Type
	vals	Input	1	Inherit: Same as Simulink	double
	mean	Output	1	Inherit: Same as Simulink	double
	stdev	Output	2	Inherit: Same as Simulink	double

Actual compiled types

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 Embedded MATLAB Function blocks can have inherited data types, sizes, and complexities like ordinary Embedded 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 **Contents** pane of the Model Explorer or 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 Embedded MATLAB function arguments as expressions in the Model Explorer or Ports and Data Manager. Follow these steps:

- 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” in the Simulink reference documentation.
 - `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

In the following figure, the data type of input argument `data1` is **int32**. The data type of input argument `data2` is based on `data1` using the expression `type(data1)`.

When the model is compiled, the actual type of `data2` appears in the **Compiled Type** column in the **Contents** pane:

Compiled data types

	Name	Port	Scope	Data Type	Compiled Type
[+]	vals	1	Input	Inherit: Same as Simulink	unknown
[+]	mean	1	Output	Inherit: Same as Simulink	unknown
[+]	stdev	2	Output	Inherit: Same as Simulink	unknown
[+]	data1	2	Input	int32	int32
[+]	data2	3	Input	type(data1)	int32

Contents of: `call_stats_block2/Embedded MATLAB Function`

Data data2

General | Value Attributes | Description

Name: `data2`

Scope: `Input` Port: `3`

Size: `-1`

Complexity: `Inherited`

Type: `type(data1)` <<

Data Type Assistant

Mode: `Expression` `type(data1)`

Specifying Simulink Fixed Point Data Properties

Embedded MATLAB Function blocks can represent signals and parameter values as fixed-point numbers. To simulate models that use fixed-point data in Embedded MATLAB Function blocks, you must install the Simulink Fixed

Point product on your system (see “Product Overview” in the Simulink Fixed Point documentation).

When you select the Fixed point data type Mode, the Data Type Assistant displays fields for additional information about your fixed-point data, as in this example:

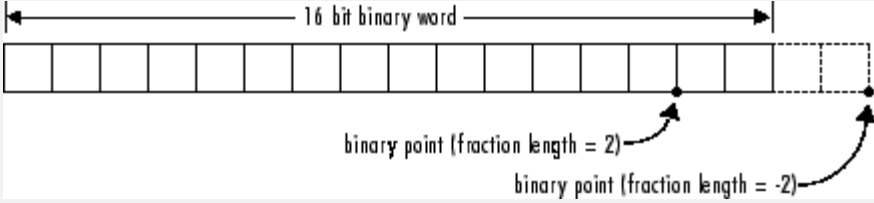
The screenshot shows the 'Data vals' dialog box with the 'Data Type Assistant' tab selected. The 'General' tab is also visible. The 'Name' field is 'vals', 'Scope' is 'Input', 'Port' is '1', 'Size' is '-1', and 'Complexity' is 'Inherited'. The 'Type' is 'fixdt(1,16,0)'. The 'Data Type Assistant' section shows 'Mode' set to 'Fixed point', 'Signedness' to 'Signed', 'Word length' to '16', 'Scaling' to 'Binary point', and 'Fraction length' to '0'. There is a 'Calculate Best-Precision Scaling' button. Below this, a checkbox for 'Fixed-point details' is checked, showing 'Representable maximum: 32767', 'Maximum:', 'Minimum:', 'Representable minimum: -32768', and 'Precision: 1'. A 'Refresh Details' button is at the bottom right. At the very bottom, there is a checkbox for 'Lock data type setting against changes by the fixed-point tools' which is unchecked.

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.

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 Select the **Value Attributes** tab.
- 2 Specify **Minimum**, **Maximum**, or both Limit range properties.
- 3 Select the **General** tab.
- 4 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, the Simulink software 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).

Fixed-point Detail	Description
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).

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

Specify whether you want to prevent replacement of the current data type with a type chosen by the Fixed-Point Tool or Fixed-Point Advisor. The default setting allows replacement. See “Scaling” in the Simulink Fixed Point documentation for instructions on autoscaling fixed-point data.

Using Data Type Override with the Embedded MATLAB Function Block

If you set the Data Type Override mode to `true doubles` or `true singles` in Simulink, the Embedded MATLAB function block sets the type of all inherited input signals and parameters to `fi double` or `fi single` objects respectively (see “Using the Embedded MATLAB Function Block with Data Type Override” in the Fixed-Point Toolbox User’s Guide 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 30-42) 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 Tools menu, select **Fixed-Point > Fixed-Point Tool**.

The Fixed-Point Tool appears.

- 2** Set the value of the **Data type override** parameter to `true doubles` or `true singles`.

Sizing Function Arguments

In this section...

“Specifying Argument Size” on page 30-93

“Inheriting Argument Sizes from Simulink” on page 30-93

“Specifying Argument Sizes with Expressions” on page 30-95

Specifying Argument Size

To examine or specify the size of an argument, follow these steps:

- 1 From the Embedded MATLAB Editor, select **Model Explorer** or **Tools > Edit Data/Ports**.
- 2 In the **Contents** pane, click the row that contains the data argument.
- 3 Enter the size of the argument in one of two places:
 - Size field of the Data properties dialog, located in the **Dialog** pane
 - Size column in the row that contains the data argument, located in the **Contents** pane

Note The default value is -1, indicating that size is inherited, as described in “Inheriting Argument Sizes from Simulink” on page 30-93.

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.

For Scope	Inherits Size
Output	From the Simulink output signal connected to the argument.
Parameter	From the Simulink or MATLAB parameter to which it is bound. See “Parameter Arguments in Embedded MATLAB Functions” on page 30-97.

After you compile the model, the **Compiled Size** column in the **Contents** pane displays the actual size used in the compiled simulation application:

Actual compiled size



The screenshot shows the Data Inspector window with a table of data values and a configuration panel for the 'vals' data value.

Name	Scope	Port	Compiled Type	Data Type	Size	Compiled Size
vals	Input	1	double	Inherit: Same as Simulink	-1	[1, 4]
mean	Output	1	double	Inherit: Same as Simulink	-1	1
stdev	Output	2	double	Inherit: Same as Simulink	-1	1

The configuration panel for 'vals' shows the following settings:

- Name: vals
- Scope: Input
- Port: 1
- Size: -1
- Complexity: Inherited
- Type: Inherit: Same as Simulink
- Data Type Assistant Mode: Inherit

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 Embedded 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

- The number of dimensions equals the length of the vector.
- The 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 defined in the MATLAB Workspace or the parent Simulink masked subsystem
- 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 in the **Contents** pane displays the actual size used in the compiled simulation application.

Parameter Arguments in Embedded MATLAB Functions

Parameter arguments for Embedded MATLAB Function blocks do not take their values from signals in the Simulink model. Instead, their values come from parameters defined in a parent Simulink masked subsystem or variables defined in the MATLAB base workspace. Using parameters allows you to pass read-only constants in the Simulink model to the Embedded MATLAB Function block.

Use the following procedure to add a parameter argument to a function for an Embedded MATLAB Function block.

- 1** In the Embedded MATLAB Editor, add an argument to the function header of the Embedded MATLAB Function block.

The name of the argument must be identical to the name of the masked subsystem parameter or MATLAB variable that you want to pass to the Embedded MATLAB Function block. For information on declaring parameters for masked subsystems, see .

- 2** Bring focus to the Embedded MATLAB Function block.

The new argument appears as an input port in the Simulink diagram.

- 3** In the Embedded MATLAB Editor, select **Model Explorer** or **Tools > Edit Data/Ports**.
- 4** In the **Contents** pane, click the row that contains the new argument.
- 5** Set **Scope** to **Parameter**.
- 6** Examine the Embedded MATLAB Function block.

The input port no longer appears for the parameter argument.

Note Parameter arguments appear as arguments in the function header of the Embedded MATLAB Function block to maintain MATLAB consistency. This lets you test functions in an Embedded MATLAB Function block by copying and pasting them to MATLAB.

Resolving Signal Objects for Output Data

In this section...

“Implicit Signal Resolution” on page 30-98

“Eliminating Warnings for Implicit Signal Resolution in the Model” on page 30-98

“Disabling Implicit Signal Resolution for an Embedded MATLAB Function Block” on page 30-99

“Forcing Explicit Signal Resolution for an Output Data Signal” on page 30-99

Implicit Signal Resolution

Embedded MATLAB Function blocks participate in signal resolution with Simulink signal objects. By default, output data from Embedded 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 in the Reference documentation.

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 “Resolving Symbols” on page 4-75 and “Explicit and Implicit Symbol Resolution” on page 4-78 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 Model Editor, select **Simulation > 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 an Embedded MATLAB Function Block

To disable implicit signal resolution for an Embedded MATLAB Function block in your model, follow these steps:

- 1 Right-click the Embedded MATLAB Function block and select **Subsystem Parameters** 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 an Embedded 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 **Signal 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**.

Working with Structures and Bus Signals

In this section...

“About Structures in Embedded MATLAB Function Blocks” on page 30-100

“Example of Structures in an Embedded MATLAB Function Block” on page 30-101

“How Structure Inputs and Outputs Interface with Bus Signals” on page 30-105

“Rules for Defining Structures in Embedded MATLAB Function Blocks” on page 30-106

“Workflow for Creating Structures in Embedded MATLAB Function Blocks” on page 30-106

“Indexing Substructures and Fields” on page 30-107

“Assigning Values to Structures and Fields” on page 30-108

“Working with Non-Tunable Structure Parameters in Embedded MATLAB Function Blocks” on page 30-110

“Limitations of Structures in Embedded MATLAB Function Blocks” on page 30-113

About Structures in Embedded MATLAB Function Blocks

Embedded MATLAB Function blocks support MATLAB structures (see “Structures” in the MATLAB getting started guide). In Embedded MATLAB Function blocks, you can define structure data as inputs or outputs that interact with bus signals. You can also define structures inside Embedded MATLAB functions that are not part of Embedded MATLAB Function blocks (see “Working with Structures” in the Embedded MATLAB documentation).

The following table summarizes how to create different types of structures in Embedded 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 Embedded MATLAB function for interfacing to other environments. See “Workflow for Creating Structures in Embedded MATLAB Function Blocks” on page 30-106.
Output	Create structure data with scope of Output.	
Local	Create a local variable implicitly in an Embedded MATLAB function.	See “Defining Local Structure Variables” in the Embedded MATLAB documentation.
Persistent	Declare a variable to be persistent in an Embedded MATLAB function.	See “Making Structures Persistent” in the Embedded MATLAB documentation.
Parameter	Create structure data with scope of Parameter	See “Working with Non-Tunable Structure Parameters in Embedded MATLAB Function Blocks” on page 30-110.

Structures in Embedded MATLAB Function blocks can contain fields of any type and size, including muxed signals, buses, and arrays of structures, as described in “Elements of Structures in the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Example of Structures in an Embedded MATLAB Function Block

The following example shows how to use structures in an Embedded MATLAB Function block:

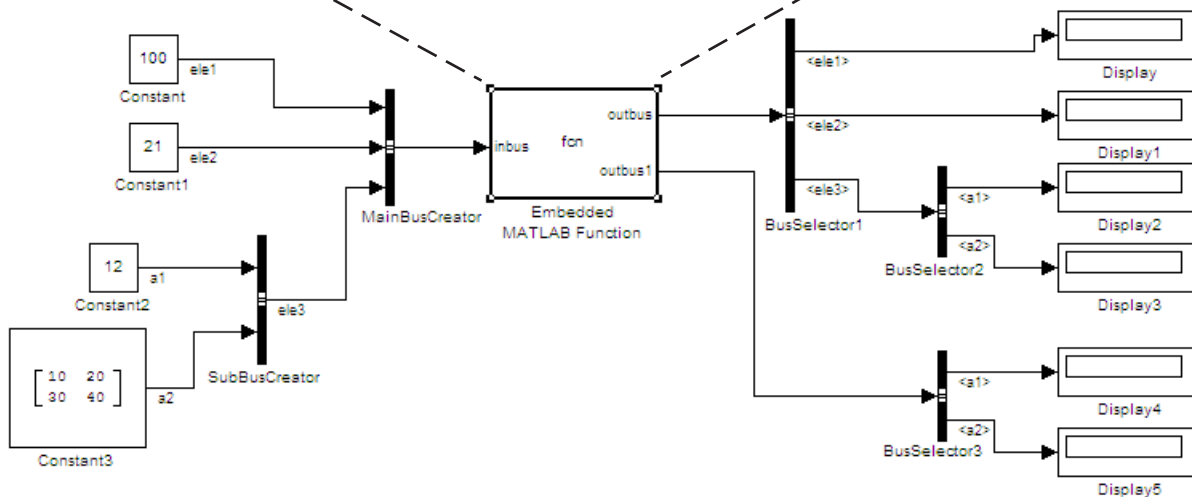
```
function [outbus, outbus1] = fcn(inbus)
%#eml

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;
```






In this model, an Embedded MATLAB Function block receives a bus signal using the structure `inbus` at input port 1 and outputs two bus signals from the structures `outbus` at output port 1 and `outbus1` at output port 2. The input signal comes from the Bus Creator block `MainBusCreator`, which bundles signals `ele1`, `ele2`, and `ele3`. The signal `ele3` is the output of another Bus Creator block `SubBusCreator`, which bundles the signals `a1` and

a2. The structure `outbus` connects to a Bus Selector block `BusSelector1`; the structure `outbus1` connects to another Bus Selector block `BusSelector3`.

Like other outputs in the Embedded MATLAB subset, structure outputs must be initialized. The Embedded MATLAB function in this example 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`.

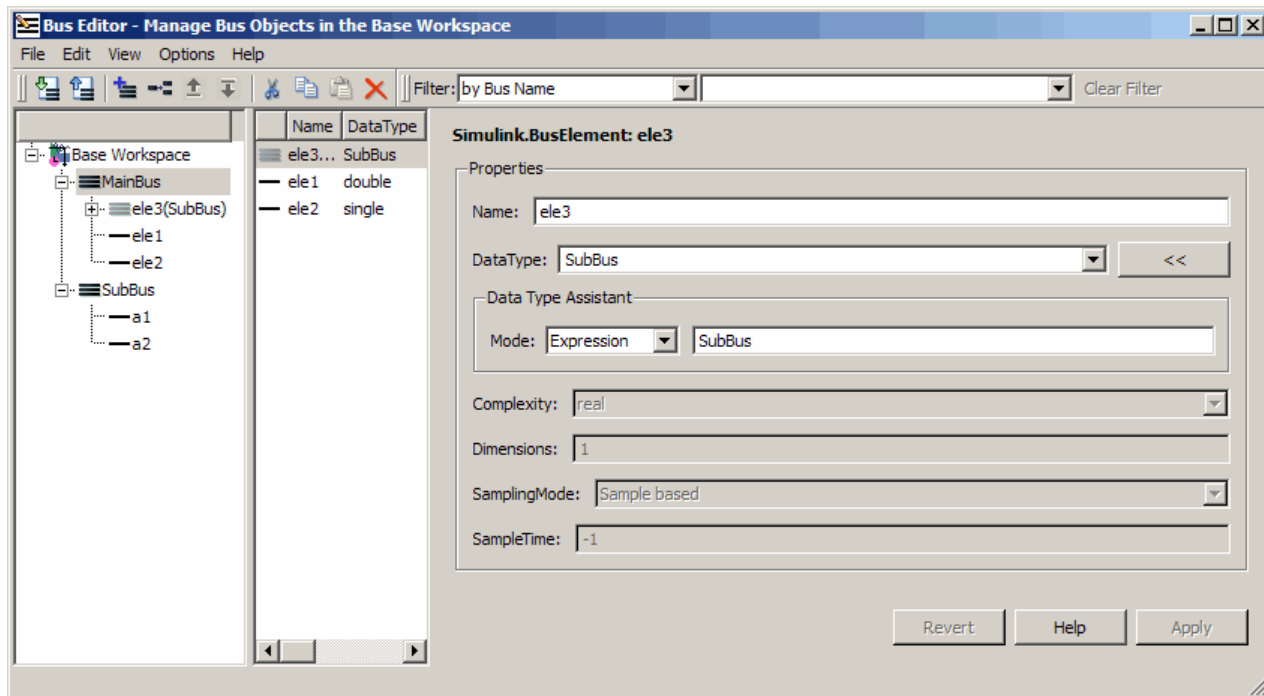
Structure Definitions in Example

Here are the definitions of the structures in the Embedded MATLAB Function block in the example, as they appear in the Ports and Data Manager:

	Name	Scope	Port	Data Type Mode	Data Type	Compiled Type
	<code>inbus</code>	Input	1	Inherited		MainBus
	<code>outbus</code>	Output	1	Bus Object	MainBus	MainBus
	<code>outbus1</code>	Output	2	Bus Object	SubBus	SubBus

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 “Workflow for Creating Structures in Embedded MATLAB Function Blocks” on page 30-106). This means that the structure shares the same properties as the bus object, including number, name, and type of fields. In this example, the following bus objects define the structure inputs and outputs:



The `Simulink.Bus` object `MainBus` defines structure input `inbus` and structure output `outbus`. The `Simulink.Bus` object `SubBus` defines structure output `outbus1`. Based on these definitions, `inbus` and `outbus` have the same properties as `MainBus` and, therefore, reference their fields by the same names as the fields in `MainBus`, using dot notation (see “Indexing Substructures and Fields” on page 30-107). 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
<code>inbus</code>	<code>inbus.ele1</code>	<code>inbus.ele2</code>	<code>inbus.ele3</code>
<code>outbus</code>	<code>outbus.ele1</code>	<code>outbus.ele2</code>	<code>outbus.ele3</code>
<code>outbus1</code>	<code>outbus1.a1</code>	<code>outbus1.a2</code>	—

To learn how to define structures in Embedded MATLAB Function blocks, see “Workflow for Creating Structures in Embedded MATLAB Function Blocks” on page 30-106.

How Structure Inputs and Outputs Interface with Bus Signals

Buses in a Simulink model appear inside the Embedded MATLAB Function block as structures; structure outputs from the Embedded MATLAB Function block appear as buses in Simulink models. When you create structure inputs, the Embedded 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 Embedded MATLAB function.

You connect structure inputs and outputs from Embedded 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 Embedded MATLAB Function blocks

Working with Virtual and Nonvirtual Buses

Embedded MATLAB Function blocks supports nonvirtual buses only (see “Virtual and Nonvirtual Buses” on page 12-48 in the Simulink User’s Guide). When models that contain Embedded MATLAB function inputs and outputs are built, hidden converter blocks are used to convert bus signals for use with Embedded MATLAB, as follows:

- Converts incoming virtual bus signals to nonvirtual buses for Embedded MATLAB structure inputs
- Converts outgoing nonvirtual bus signals from Embedded MATLAB to virtual bus signals

Rules for Defining Structures in Embedded MATLAB Function Blocks

Follow these rules when defining structures in Embedded MATLAB Function blocks:

- For each structure input or output in an Embedded MATLAB Function block, you must define a `Simulink.Bus` object in the base workspace to specify its type. For more information, see `Simulink.Bus`.
- Embedded MATLAB Function blocks support nonvirtual buses only (see “Working with Virtual and Nonvirtual Buses” on page 30-105).

Workflow for Creating Structures in Embedded MATLAB Function Blocks

Here is the workflow for creating a structure in Embedded MATLAB:

- 1 Decide on the type (or scope) of the structure (see “About Structures in Embedded MATLAB Function Blocks” on page 30-100).
- 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 Embedded MATLAB Function block, as described in “Adding Data to an Embedded MATLAB Function Block” on page 30-49. The data should have the following properties <ul style="list-style-type: none"> • Scope = Input • Mode = Bus Object • Bus object = name of the <code>Simulink.Bus</code> object that defines the structure input <p>See “Rules for Defining Structures in Embedded MATLAB Function Blocks” on page 30-106.</p>

For Structure Scope:	Follow These Steps:
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 Embedded MATLAB Function block with the following properties: <ul style="list-style-type: none"> • Scope = Output • Mode = Bus Object • Bus object = name of the <code>Simulink.Bus</code> object that defines the structure input 3 Define and initialize the output structure implicitly as a variable in the Embedded MATLAB function, as described in “Defining Outputs as Structures” in the Embedded MATLAB documentation. 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.
Local	Define the structure implicitly as a local variable in the Embedded MATLAB function, as described in “Defining Local Structure Variables” in the Embedded MATLAB documentation. By default, local variables in Embedded MATLAB are temporary.
Persistent	Define the structure implicitly as a persistent variable in the Embedded MATLAB function, as described in “Making Structures Persistent” in the Embedded MATLAB documentation.
Parameter	<ol style="list-style-type: none"> 1 Create a structure variable in the base workspace. 2 Add data to the Embedded 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 3 Make sure the Tunable property is unchecked. <p>See “Working with Non-Tunable Structure Parameters in Embedded MATLAB Function Blocks” on page 30-110.</p>

Indexing Substructures and Fields

As in MATLAB, you index substructures and fields of Embedded MATLAB structures by using dot notation. Unlike MATLAB, you must reference field

values individually (see “Limitations with Structures” in the Embedded MATLAB documentation).

For example, in the model described in “Example of Structures in an Embedded MATLAB Function Block” on page 30-101, the Embedded MATLAB function uses dot notation to index fields and substructures:

```
function [outbus, outbus1] = fcn(inbus)

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 Embedded MATLAB resolves symbols in dot notation for indexing elements of the structures in this example:

Dot Notation	Symbol Resolution
substruct.a1	Field a1 of local structure substruct
inbus.ele3.a1	Value of field a1 of field ele3, a substructure of structure inputinbus
inbus.ele3.a2(1,1)	Value in row 1, column 1 of field a2 of field ele3, a substructure of structure input inbus

Assigning Values to Structures and Fields

You can assign values to any Embedded MATLAB structure, substructure, or field. 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 “Workflow for Creating Structures in Embedded MATLAB Function Blocks” on page 30-106).
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 “Example of Structures in an Embedded MATLAB Function Block” on page 30-101:

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.e1e3;</code>	Valid	Both <code>outbus1</code> and <code>inbus.e1e3</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> .

Working with Non-Tunable Structure Parameters in Embedded MATLAB Function Blocks

You can define non-tunable structure parameters in Embedded MATLAB Function blocks. Models that contain non-tunable structure parameters will compile both for simulation and code generation with Real-Time Workshop.

Defining Non-Tunable Structure Parameters

To define non-tunable structure parameters in Embedded 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 “Working with Block Parameters” on page 7-15.

- 2 In the Ports and Data Manager or Model Explorer, add data in the Embedded 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	Uncheck this check box
Type	Select Inherit: Same as Simulink

For example, the data dialog box fields should look something like this:

Data p

General | Value Attributes | Description

Name: p

Scope: Parameter Tunable

Size: -1

Complexity: Inherited

Type: Inherit: Same as Simulink >>

Caution Embedded MATLAB Function blocks do not support tunable structure parameters. If you define a structure parameter as tunable, you receive a run-time error:

Parameter is a tunable structure parameter
which is not supported. Set the parameter
to be non-tunable, or do not use a structure value.

3 Click **Apply**.

FIMATH Properties of Non-Tunable Structure Parameters

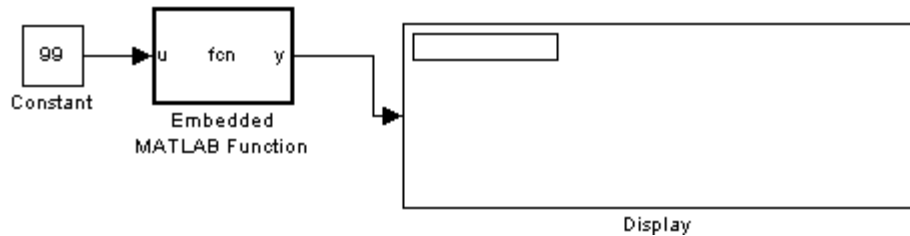
FIMATH properties for non-tunable structure parameters containing fixed-point values are based on the initial values of the structure. They do *not* come from the FIMATH properties specified for fixed-point input signals to the parent Embedded MATLAB Function block. (These FIMATH properties appear in the properties dialog box for Embedded MATLAB Function blocks.)

Rules for Defining Non-Tunable Structure Parameters

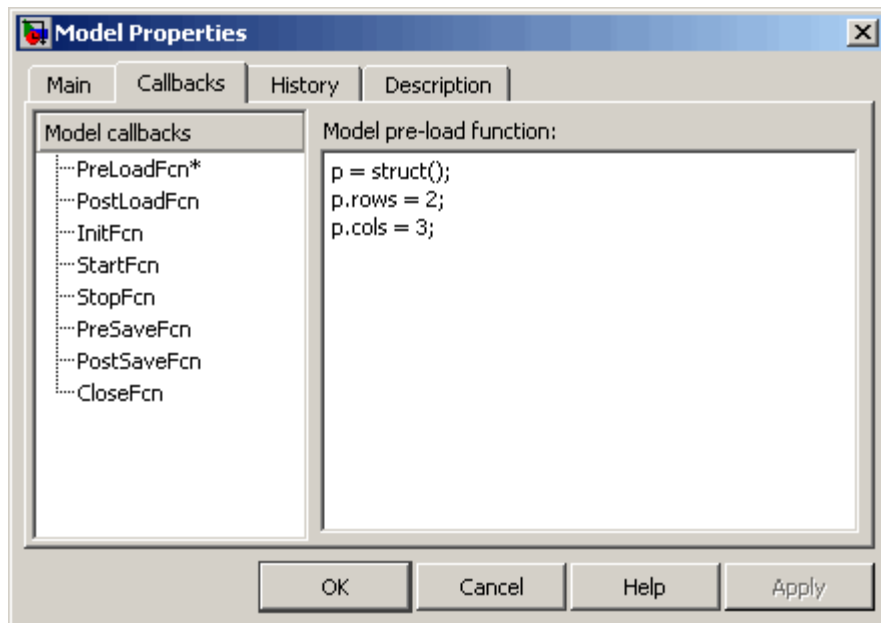
Non-tunable structure parameters in Embedded MATLAB Function blocks have the same limitations as structures in Embedded MATLAB functions. See “Limitations with Structures” in the Embedded MATLAB documentation.

Example: Using a Non-Tunable Structure Parameter to Initialize a Matrix

The following simple example uses a non-tunable structure parameter input to initialize a matrix output. The model looks like this:



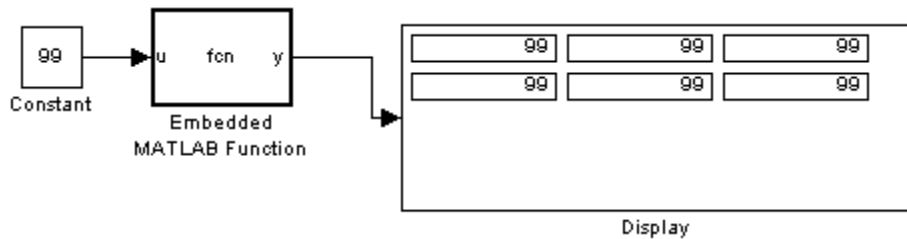
This model defines a structure variable `p` in its pre-load callback function, as follows:



The structure `p` has two fields, `rows` and `cols`, which specify the dimensions of a matrix. The Embedded 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`:



Limitations of Structures in Embedded MATLAB Function Blocks

Structures in Embedded MATLAB Function blocks support a subset of the operations available for MATLAB structures (see “Limitations with Structures” in the Embedded MATLAB documentation).

Using Variable-Size Data in Embedded MATLAB Function Blocks

In this section...

“What Is Variable-Size Data?” on page 30-114

“How Embedded MATLAB Function Blocks Implement Variable-Size Data” on page 30-114

“Enabling Support for Variable-Size Data” on page 30-115

“Declaring Variable-Size Inputs and Outputs” on page 30-115

“Declaring Variable-Size Data Locally” on page 30-115

“Simple Example: Defining and Using Variable-Size Data in Embedded MATLAB Function Blocks” on page 30-116

“Limitations of Variable-Size Support in Embedded MATLAB Function Blocks” on page 30-122

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 Embedded MATLAB Function Blocks Implement Variable-Size Data

You can define variable-size arrays and matrices as inputs, outputs, and local data in Embedded 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 working with variable-size data in the Embedded MATLAB subset, see “Generating Code for Variable-Size Data” in the Embedded MATLAB user guide documentation. For more information about using variable-size data in Simulink, see Chapter 11, “Working with Variable-Size Signals” in the Simulink user guide documentation.

Enabling Support for Variable-Size Data

Support for variable-size data is enabled by default for Embedded MATLAB Function blocks. To modify this property for individual blocks:

- 1 In the Embedded MATLAB Editor, select **Tools > Edit Data/Ports**

The Ports and Data Manager dialog box opens.

- 2 Select or clear the check box **Support variable-size arrays**.

Declaring Variable-Size Inputs and Outputs

- 1 In the Embedded MATLAB Editor, select **Tools > Edit Data/Ports**

The Ports and Data Manager dialog box opens.

- 2 Click the Add Data icon:



- 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.

Declaring Variable-Size Data Locally

Use the function `eml.varsize` to declare variable-size data locally, as described in “Declaring Variable-Size Data in Embedded MATLAB Code” in the Embedded MATLAB user guide documentation.

Simple Example: Defining and Using Variable-Size Data in Embedded MATLAB Function Blocks

- “About the Example” on page 30-116
- “Simulink Model” on page 30-116
- “Source Signal” on page 30-117
- “Embedded MATLAB Function Block: uniquify ” on page 30-118
- “Embedded MATLAB Function Block: avg” on page 30-119
- “Variable-Size Results” on page 30-120

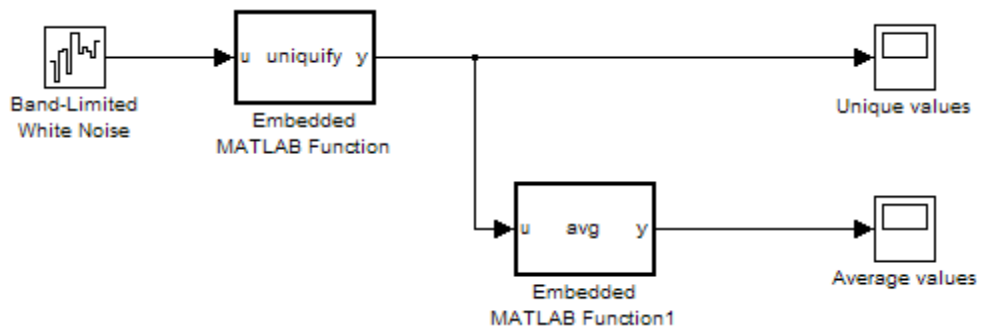
About the Example

The following example appears throughout this section to illustrate how Embedded 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

The model that generates and filters the white noise signal looks like this:



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.
Embedded MATLAB Function uniuify	Filters out signal values that are not unique to within a specified tolerance of each other.
Embedded 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 uniuify function.
Average values	Scope that displays the average signal values output from the avg function.

Source Signal

The band-limited white noise signal has these properties:

Parameters

Noise power:
[0.1 0.2 0.3 0.4 0.5 0.6 0.1 0.2 0.3]

Sample time:
0.1

Seed:
[2223334]

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.

Embedded 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) %#eml
y = uniquetol(u,0.2);
```

The `uniquify` function calls an external Embedded 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) %#eml

A = sort(A);
eml.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:

```
eml.varsize('B',[1 100]);
```

In this statement, `eml.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`:

The image shows a configuration dialog box for a data block named 'y'. It has three tabs: 'General', 'Value Attributes', and 'Description'. The 'General' tab is selected. The 'Name' field is 'y'. The 'Scope' is 'Output' and 'Port' is '1'. There is a checkbox for 'Data must resolve to Simulink signal object' which is unchecked. The 'Size' field is '[1 9]' and the 'Variable size' checkbox is checked. The 'Complexity' is 'Inherited' and 'Sampling mode' is 'Sample based'. The 'Type' is 'Inherit: Same as Simulink'. A red oval highlights the 'Size' field and the 'Variable size' checkbox.

For variable-size outputs, you must specify an explicit size and upper bound.

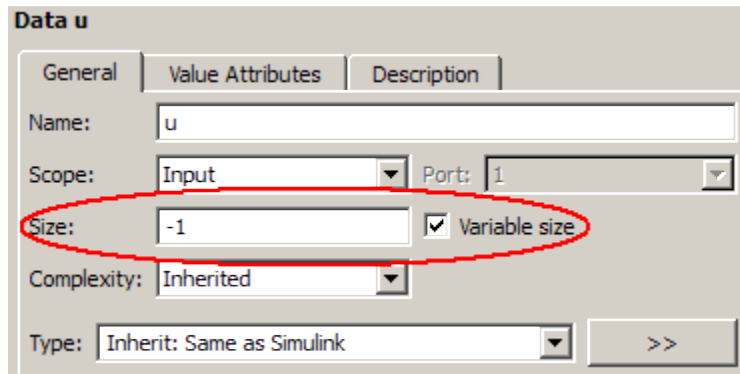
Embedded MATLAB Function Block: avg

This block averages signal values filtered by the `uniquify` block and outputs the resulting means to the **Average values** scope. Here is the code:

```
function y = avg(u) %#eml

k = numel(u)/2;
if k == floor(k)
    y = nway(u,2);
else
    y = u;
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 Embedded MATLAB function nway to calculate the average of every two signal values. Here is the code for nway:

```
function B = nway(A,n) %#eml

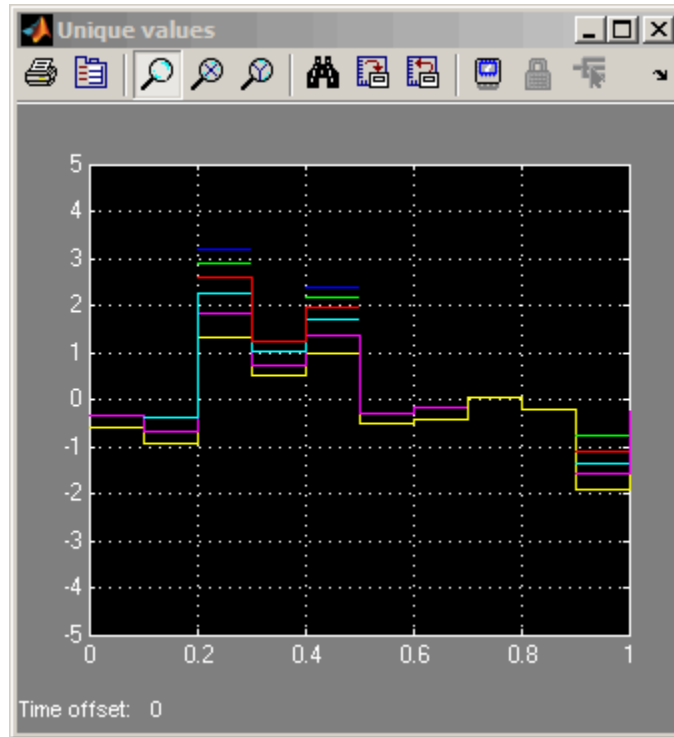
assert(n>=1 && n<=numel(A));

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
```

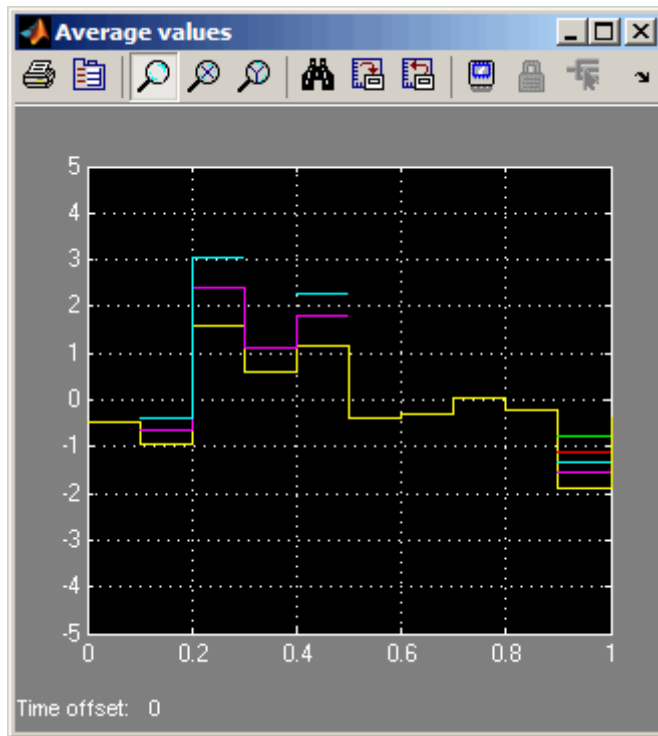
Variable-Size Results

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:



Limitations of Variable-Size Support in Embedded MATLAB Function Blocks

- You cannot declare variable-size fields in buses.
- The Embedded MATLAB debugger does not display structure fields that are variable-size arrays.

Using Enumerated Data in Embedded MATLAB Function Blocks

In this section...

“Enumerated Data in Embedded MATLAB Function Blocks” on page 30-123

“When to Use Enumerated Data” on page 30-124

“Simple Example: Defining and Using Enumerated Types in Embedded MATLAB Function Blocks” on page 30-125

“Using Enumerated Data in Embedded MATLAB Function Blocks” on page 30-129

“How to Define Enumerated Data Types for Embedded MATLAB Function Blocks” on page 30-130

“How to Add Enumerated Data to Embedded MATLAB Function Blocks” on page 30-130

“How to Instantiate Enumerated Data in Embedded MATLAB Function Blocks” on page 30-132

“Operations on Enumerated Data” on page 30-133

“Limitations of Enumerated Types” on page 30-133

Enumerated Data in Embedded 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, Embedded 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 Embedded 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 Chapter 14, “Using Enumerated Data” in the Simulink documentation.

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, Embedded MATLAB 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.

Simple Example: Defining and Using Enumerated Types in Embedded MATLAB Function Blocks

- “About the Example” on page 30-125
- “Class Definition: myMode” on page 30-126
- “Class Definition: myLED” on page 30-126
- “Simulink Model” on page 30-126
- “Embedded MATLAB Function Block: checkState” on page 30-128
- “How the Model Displays Enumerated Data” on page 30-128

About the Example

The following example appears throughout this section to illustrate how Embedded 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 Embedded 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: `myMode` to represent the set of allowable modes and `myLED` 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.

See Also.

- “Using Enumerated Data in Embedded MATLAB Function Blocks” on page 30-129

- “How to Define Enumerated Data Types for Embedded MATLAB Function Blocks” on page 30-130
- “How to Instantiate Enumerated Data in Embedded MATLAB Function Blocks” on page 30-132

Class Definition: myMode

Here is the class definition of the myMode enumerated data type:

```
classdef(Enumeration) myMode < Simulink.IntEnumType
    enumeration
        OFF(0)
        ON(1)
    end
end
```

This definition must reside on the MATLAB path in an M-file with the same name as the class, myMode.m.

Class Definition: myLED

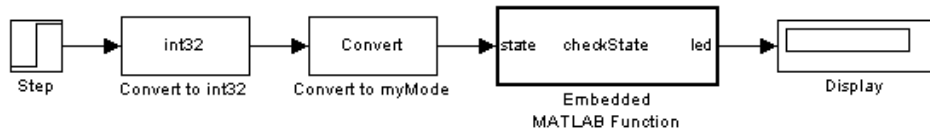
Here is the class definition of the myLED enumerated data type:

```
classdef(Enumeration) myLED < Simulink.IntEnumType
    enumeration
        GREEN(1),
        RED(8),
    end
end
```

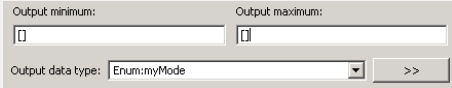
This definition must reside on the MATLAB path in an M-file called myLED.m. The set of allowable values do not need to be consecutive integers.

Simulink Model

The model that controls the LED display looks like this:



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 myMode	<p>Converts the value of type int32 to the enumerated type myMode. In the Data 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 Conversion block parameters have these settings:</p>  <p>For more information about specifying enumerated types in Simulink models, see “Referencing an Enumerated Type” on page 14-14.</p>



Simulink Block	Description
Embedded MATLAB Function	Evaluates enumerated data input state to determine the color to output as enumerated data led. See “Embedded MATLAB Function Block: checkState” on page 30-128.
Display	Displays the enumerated value of output led.

Embedded MATLAB Function Block: checkState

The function `checkState` in the Embedded 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 led = checkState(state)
    %#eml
    if state == myMode.ON
        led = myLED.GREEN;
    else
        led = myLED.RED;
    end
end
```

The input `state` inherits its enumerated type `myMode` from the Simulink step signal; the enumerated type of output `led` is explicitly declared as `myLED`:

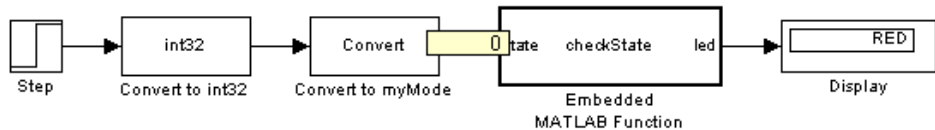
	Name	Scope	Port	DataType	Compiled Type
	state	Input	1	Inherit: Same as Simulink	myMode
	led	Output	1	Enum: myLED	myLED

Explicit enumerated type declarations must include the prefix `Enum:`. For more information, see “Referencing an Enumerated Type” on page 14-14.

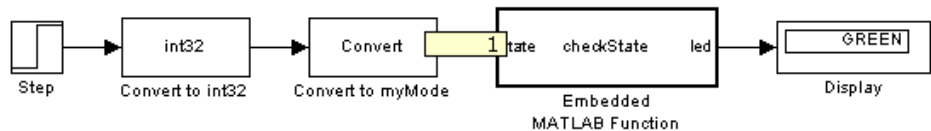
How the Model Displays Enumerated Data

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 “Displaying Enumerated Values” on page 14-17.

Using Enumerated Data in Embedded MATLAB Function Blocks

Here is the basic workflow for using enumerated data in Embedded MATLAB function blocks:

Step	Action	How?
1	Define an enumerated data type that inherits from <code>Simulink.IntEnumType</code> .	See “How to Define Enumerated Data Types for Embedded MATLAB Function Blocks” on page 30-130.
2	Add the enumerated data to your Embedded MATLAB Function block.	See “How to Add Enumerated Data to Embedded MATLAB Function Blocks” on page 30-130.

Step	Action	How?
3	Instantiate the enumerated type in your Embedded MATLAB Function block.	See “How to Instantiate Enumerated Data in Embedded MATLAB Function Blocks” on page 30-132.
4	Simulate and/or generate code.	See “Simulating with Enumerated Types” on page 14-11 and “Enumerated Data Type Considerations” in the Real-Time Workshop documentation.

How to Define Enumerated Data Types for Embedded MATLAB Function Blocks

You define enumerated data types for Embedded MATLAB Function blocks in the same way as for other Simulink blocks. The basic workflow is:

- 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 M-file on the MATLAB path.

For complete descriptions of each procedure, see “Defining an Enumerated Data Type” on page 14-4 .

How to Add Enumerated Data to Embedded MATLAB Function Blocks

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.

For:	Do This:
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 an Embedded MATLAB Function block:

- 1** In the Embedded MATLAB Editor, select **Tools > Edit Data/Ports**.

The Ports and Data Manager dialog box appears with the **General** pane selected.

- 2** 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.

- 3** In the **Type** field, specify an enumerated type.

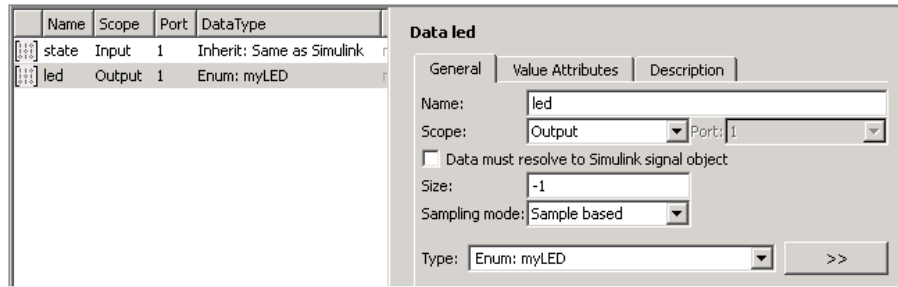
To specify an explicit enumerated type:

- Select Enum:<class name> from the drop-down menu in the **Type** field.
- Replace <class name> with the name of an enumerated data type that you defined in an M-file on the MATLAB path.

For example, you can enter Enum:myLED in the **Type** field. (See “How to Define Enumerated Data Types for Embedded MATLAB Function Blocks” on page 30-130.)

Note The **Complexity** field disappears when you select Enum:<class name> because enumerated data types do not support complex values.

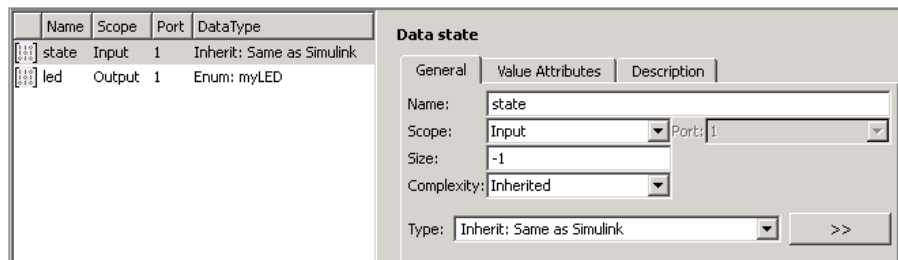
For example, the following output `led` has an explicit enumerated type, `myLED`:



To inherit the enumerated type from a connected Simulink signal (for inputs only):

- 1 Select `Inherit:Same as Simulink` from the drop-down menu in the **Type** field.

For example, the following input `state` inherits its enumerated type `myMode` from a Simulink signal:



- 4 Click **Apply**.

How to Instantiate Enumerated Data in Embedded MATLAB Function Blocks

To instantiate an enumerated type in an Embedded MATLAB Function block, use dot notation to specify `ClassName.EnumName`. For example, the following Embedded MATLAB function `checkState` instantiates the enumerated types `myMode` and `myLED` from “Simple Example: Defining and Using Enumerated

Types in Embedded MATLAB Function Blocks” on page 30-125. The dot notation appears highlighted in the code.

```
function led = checkState(state)
    %#eml

    if state == myMode.ON
        led = myLED.GREEN;
    else
        led = myLED.RED;
    end
```

Operations on Enumerated Data

Simulink software prevents enumerated values from being used as numeric values in mathematical computation (see “Enumerated Values in Computation” on page 14-18) .

The Embedded MATLAB subset supports the following enumerated data operations:

- Assignment (=)
- Relational operations (==, ~=, <, >, <=, >=,)
- Cast
- Indexing

For more information, see “Operations on Enumerated Data in the Embedded MATLAB Subset”.

Limitations of Enumerated Types

Enumerated types in Embedded MATLAB Function blocks are subject to the limitations imposed by the Embedded MATLAB subset. See “Limitations of Enumerated Types in the Embedded MATLAB Subset”. To learn about the limitations of enumerated types in Simulink, see “Simulink Enumerated Type Limitations” on page 14-25 .

Working with Frame-Based Signals

In this section...

“About Frame Based Signals” on page 30-134

“Supported Types for Frame-Based Data” on page 30-135

“Adding Frame-Based Data in Embedded MATLAB Function Blocks” on page 30-135

“Examples of Frame-Based Signals in Embedded MATLAB Function Blocks” on page 30-136

About Frame Based Signals

Embedded 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 install Signal Processing Blockset. For more information about using frame-based signals, see “Frame-Based Signals” in the Signal Processing Blockset documentation.

Embedded 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, Embedded MATLAB provides a data property called **Sampling mode** that lets you specify whether your output is a frame-based or sample-based signal for downstream processing.

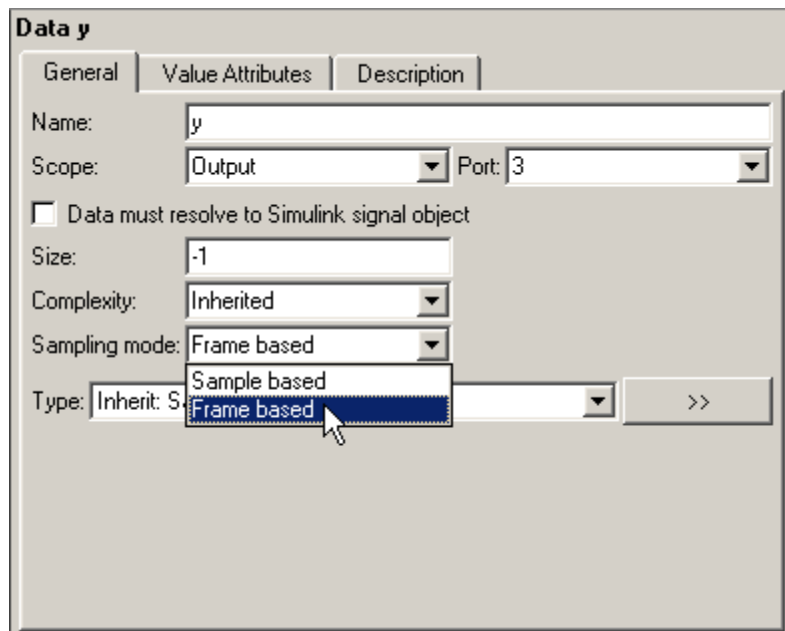
Supported Types for Frame-Based Data

Embedded MATLAB Function blocks accept frame-based signals of any data type *except* bus objects. For a list of supported types, see “Supported Variable Types” in the Embedded MATLAB documentation.

Adding Frame-Based Data in Embedded MATLAB Function Blocks

To add frame-based data to an Embedded MATLAB Function block, follow these steps:

- 1 Add an input or output, as described in “Adding Data to an Embedded MATLAB Function Block” on page 30-49.
- 2 If your data is an output, set **Sampling mode** to Frame based.



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 Model Explorer” on page 30-51.

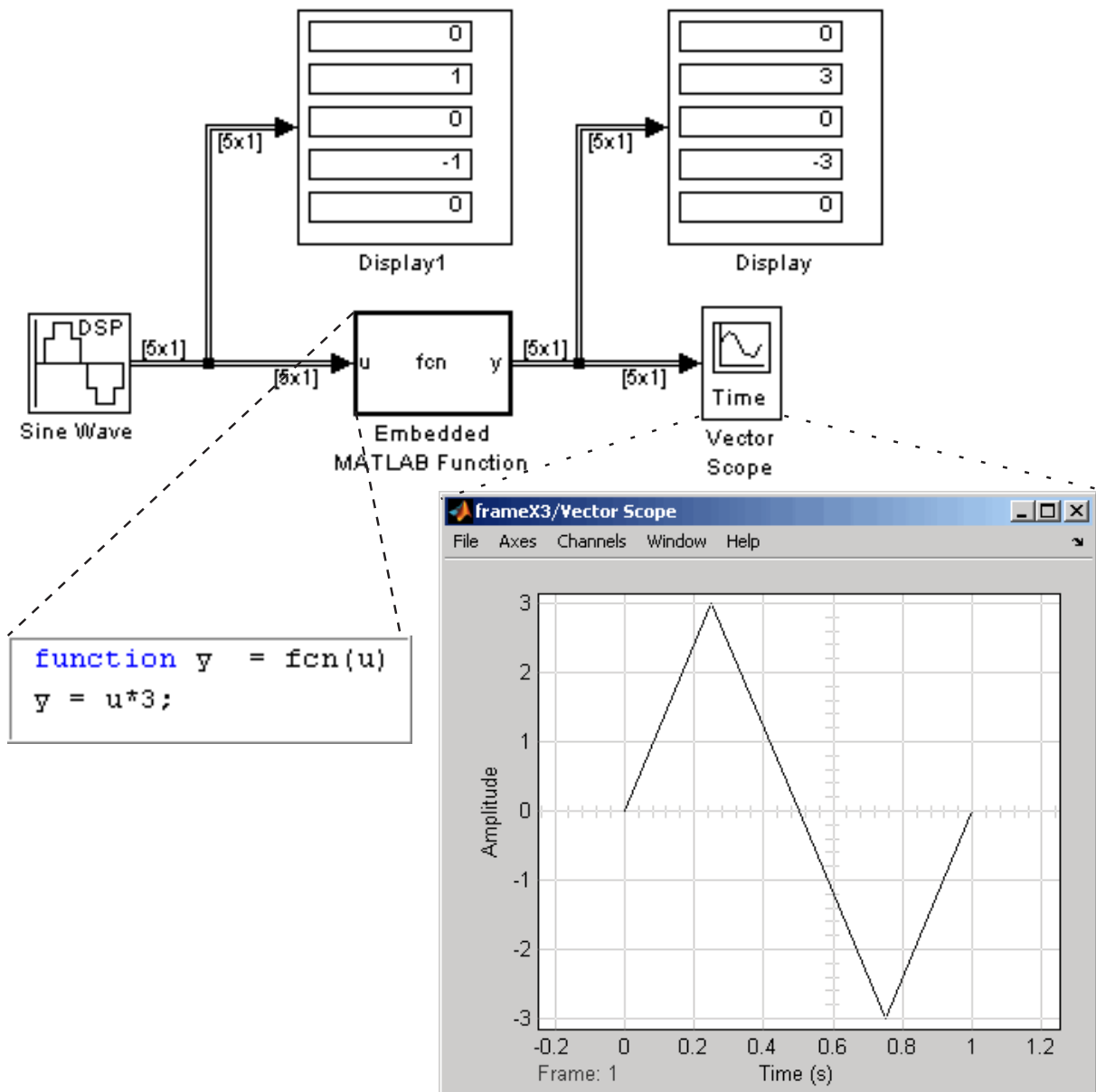
Examples of Frame-Based Signals in Embedded MATLAB Function Blocks

This topic presents examples of how to work with frame-based signals in Embedded MATLAB Function blocks.

- “Multiplying a Frame-Based Signal by a Constant Value” on page 30-136
- “Adding a Channel to a Frame-Based Signal” on page 30-138

Multiplying a Frame-Based Signal by a Constant Value

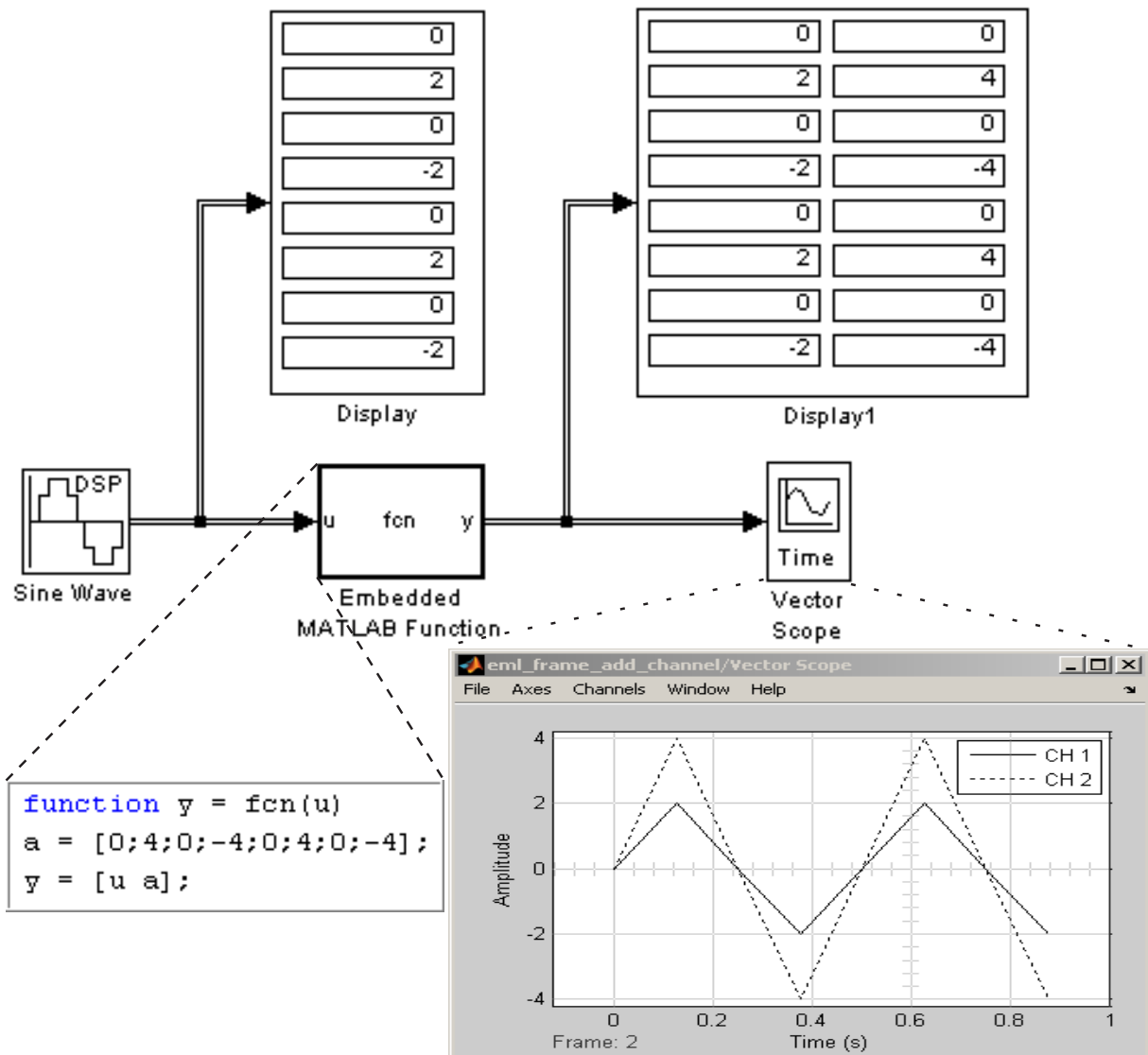
In the following example, an Embedded MATLAB Function block multiplies all the signal values in a frame-based single-channel input by a constant value and outputs the result as a frame. The input signal is a sine wave that contains 5 samples per frame. Here is the model:



In the Embedded MATLAB Function block, 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 Embedded MATLAB Function Blocks” on page 30-135). When you simulate this model, the Embedded 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, an Embedded 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. Here is the model:



In the Embedded MATLAB Function block, 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 Embedded MATLAB Function Blocks” on page 30-135). 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 Embedded MATLAB Function block outputs the new multichannel signal.

Using Traceability in Embedded MATLAB Function Blocks

In this section...

“Extent of Traceability in Embedded MATLAB Function Blocks” on page 30-141

“Traceability Requirements” on page 30-141

“Basic Workflow for Using Traceability” on page 30-141

“Tutorial: Using Traceability in an Embedded MATLAB Function Block” on page 30-143

Extent of Traceability in Embedded MATLAB Function Blocks

Like other Simulink blocks, Embedded 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.

For information about how traceability works in Simulink blocks, see “Tracing Generated Code” in the Real-Time Workshop User’s Guide.

Traceability Requirements

To enable traceability comments in your code, you must have a license for Real-Time Workshop Embedded Coder software. These comments appear only in code that you generate for an embedded real-time (ert) based target.

Note Traceability is not supported for M-files that you call from an Embedded MATLAB Function block.

Basic Workflow for Using Traceability

The workflow for using traceability is described in “Generating Reports for Code Reviews and Traceability Analysis” in the Real-Time Workshop Embedded Coder User’s Guide.

Here are the basic steps:

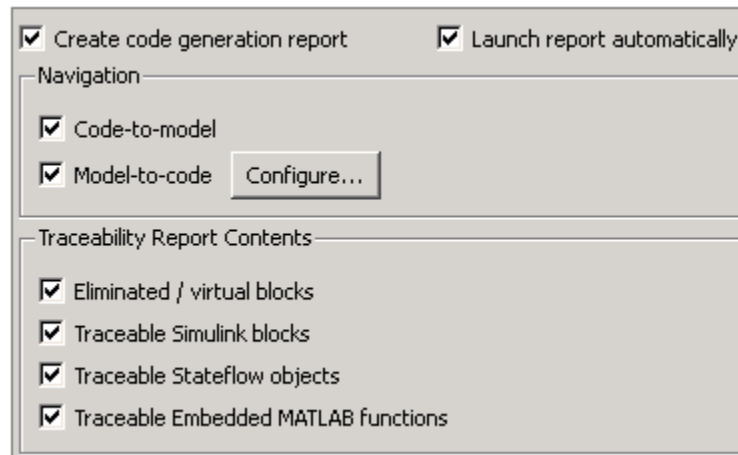
- 1 Open the Embedded MATLAB Function block in your Simulink model.
- 2 Define your system target file to be an embedded real-time (ert) target.

How?

- a In the model, open **Simulation > Configuration Parameters**.
 - b In the Real-Time Workshop pane, enter `ert.tlc` for the system target file.
- 3 Enable traceability options.

How?

In the Real-Time Workshop Report pane, enable options, as shown:



- 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 Real-Time Workshop > 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 an Embedded MATLAB Function Block” on page 30-143

Tutorial: Using Traceability in an Embedded MATLAB Function Block

This example shows how to trace between source code and generated code in an Embedded MATLAB Function block in the `eml_fire` demo 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 Embedded MATLAB Editor.
- 3 In the Simulink model window, select **Simulation > Configuration Parameters**.
- 4 In the **Real-Time Workshop** 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 **Real-Time Workshop > Report** pane, select the **Create code generation report** option.

This action automatically selects the **Launch 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.

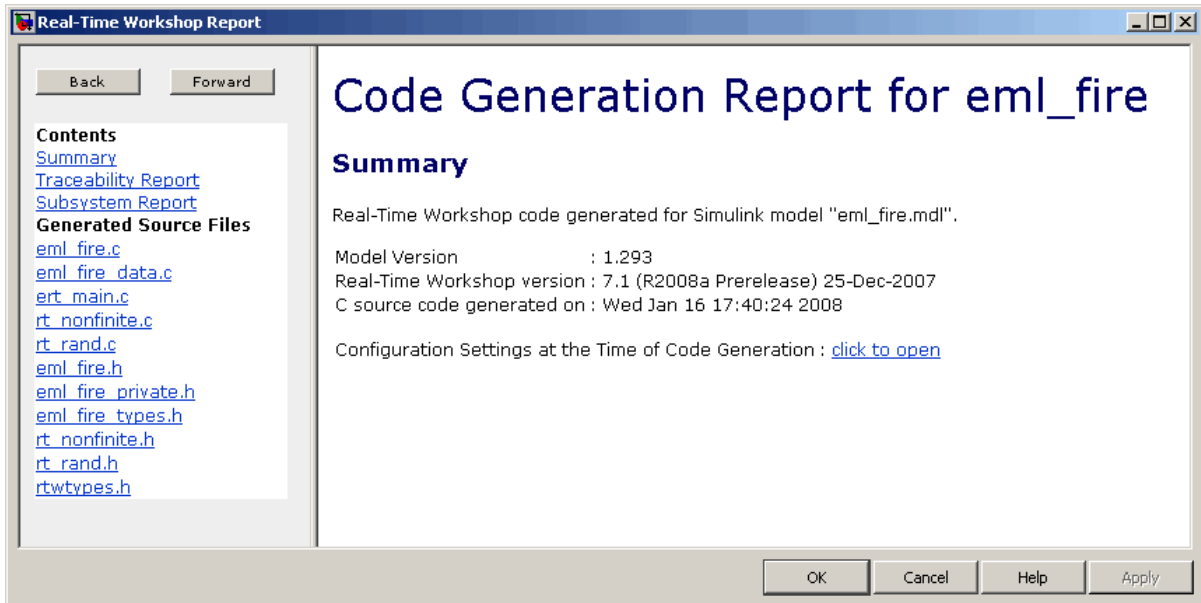
Note For large models that contain over 1000 blocks, disable the **Model-to-code** option to speed up code generation.

- 7** Go to the **Real-Time Workshop > Interface** pane. In the **Software environment** section, select the **continuous time** option. Then click **Apply**.

Note Because this demo model contains a block with a continuous sample time, you must perform this step before generating code.

- 8** In the **Real-Time Workshop** 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.

```

61     for (eml_y = 0; eml_y < 60; eml_y += 2) {
62         /* '<S2>:1:18' */
63         for (eml_x = 0; eml_x < 256; eml_x++) {
64             /* '<S2>:1:19' */ ← Traceability comment for a
65             /* '<S2>:1:21' */ ← line of code in an Embedded
66             eml_yb = eml_y + 3;
67
68             /* '<S2>:1:22' */ ← Traceability comment for a
69             eml_xb1 = eml_x;

```

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 Embedded MATLAB function appears highlighted in the Embedded MATLAB Editor.

```

18  for y = 1 : 2 : (HEIGHT-4)
19  for x = 1 : WIDTH
20
21      yb = y+2;
22      xb1 = x-1;
23      xb2 = x+1;
24
25      if xb1 < 1
26          xb1 = WIDTH;
27          yb = yb + 1;
28      end
29      if xb2 > WIDTH
30          xb2 = 1;

```

Highlighted line of function

- 12** You can also trace a line in an Embedded MATLAB function to a line of generated code. For example, right-click in line 21 of your function and select **Real-Time Workshop > Navigate to Code** from the context menu.

```

18  for y = 1 : 2 : (HEIGHT-4)
19  for x = 1 : WIDTH
20
21      yb = y+2;
22      xb1 = x-1;
23      xb2 = x+1;
24
25      if xb1 < 1
26          xb1 = WIDTH;
27          yb = yb + 1;
28      end
29      if xb2 > WIDTH
30          xb2 = 1;
31          yb = yb - 1;
32      end
33
34      if yb > HEIGHT
35          yb = 1;
36      end
37

```

Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Comment	Ctrl+R
Uncomment	Ctrl+T
Smart Indent	Ctrl+I
Decrease Indent	Ctrl+[
Increase Indent	Ctrl+]
Find and Replace...	Ctrl+F
Evaluate Selection	F9
Open Selection	
Data Scope	
Real-Time Workshop	Navigate to Code...

The code location for line 21 appears highlighted in `eml_fire.c`.

```
61     for (eml_y = 0; eml_y < 60; eml_y += 2) {
62         /* '<S2>:1:18' */
63         for (eml_x = 0; eml_x < 256; eml_x++) {
64             /* '<S2>:1:19' */
65             /* '<S2>:1:21' */
66             eml_yb = eml_y + 3;
```

Highlighted line of code

Enhancing Readability of Generated Code for Embedded MATLAB Function Blocks

In this section...

“Requirements for Using Readability Optimizations” on page 30-148

“Converting If-Elseif-Else Code to Switch-Case Statements” on page 30-148

“Example of Converting Code for If-Elseif-Else Decision Logic to Switch-Case Statements” on page 30-151

Requirements for Using Readability Optimizations

To use readability optimizations in your code, you must have a Real-Time Workshop 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 M-files that you call from an Embedded MATLAB Function block.

For more information, see “Choosing and Configuring an Embedded Real-Time Target” and “Controlling Code Style” in the Real-Time Workshop 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 an Embedded 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 Embedded 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 30-153
2	Enable the conversion.	“Enabling the Conversion” on page 30-154
3	Generate code for your model.	“Generating Code for Your Model” on page 30-155

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
Embedded 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 30-150.</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 an Embedded 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.

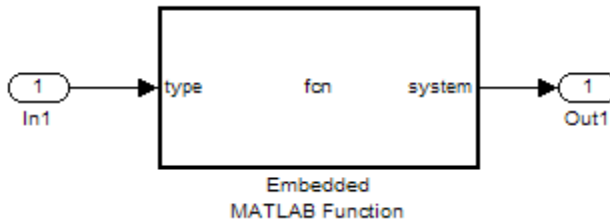
The following examples show how the conversion handles duplicate conditions.

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 an Embedded MATLAB Function block. Assume that the output data type is `double` and the input data type is `Controller`, an enumerated type that you define. (See “How to Define Enumerated Data Types for Embedded MATLAB Function Blocks” on page 30-130 for more information.)



The block contains the following code:

```
function system = fcn(type)
%#eml

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 “Using Traceability in Embedded MATLAB Function Blocks” on page 30-141.

Verifying the Contents of the Block

Check that the block follows all the rules in “Rules for Conversion” on page 30-149.

Construct	How the Construct Follows the Rules
Embedded 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 **Real-Time Workshop** pane, select `ert.tlc` for the **System target file**.

This step specifies an embedded real-time target for your model.

- 3** In the **Real-Time Workshop > 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 Embedded MATLAB Function blocks in a model
- Embedded MATLAB functions in all Stateflow charts of that model
- Flow graphs in all Stateflow charts of that model

For more information, see “Enhancing Readability of Generated Code for Flow Graphs” in the Stateflow documentation.

Generating Code for Your Model

In the **Real-Time Workshop** pane of the Configuration Parameters dialog box, click **Build** in the lower right corner.

The code for the Embedded 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.

Speeding Up Simulation with the Basic Linear Algebra Subprograms (BLAS) Library

In this section...

“How Embedded MATLAB Function Blocks Use the BLAS Library” on page 30-157

“When to Disable BLAS Library Support” on page 30-157

“How to Disable BLAS Library Support” on page 30-158

“Supported Compilers” on page 30-158

How Embedded MATLAB Function Blocks Use the BLAS Library

The Basic Linear Algebra Subprograms (BLAS) Library is a library of external linear algebra routines optimized for fast computation of low-level matrix operations. By default, Embedded MATLAB Function blocks call BLAS library routines to speed simulation whenever possible, based on heuristics implemented in the Embedded MATLAB subset. These heuristics are described in “How Embedded MATLAB Functions Use the BLAS Library” in the Embedded MATLAB documentation.

When to Disable BLAS Library Support

Consider disabling BLAS library support for Embedded MATLAB Function blocks when:

- You want your simulation results to more closely agree with code generated by Real-Time Workshop for your Embedded MATLAB Function block.
- You are executing code on a 64-bit platform and the number of elements in a matrix exceeds 32 bits.

In this case, Embedded MATLAB automatically truncates the matrix size to 32 bits.

- Your platform does not provide a robust implementation of BLAS routines.

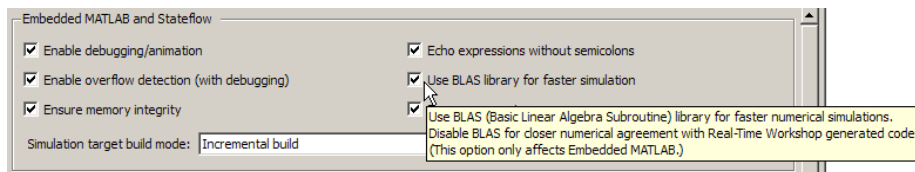
How to Disable BLAS Library Support

Embedded MATLAB Function blocks enable BLAS library support by default, but you can disable this feature explicitly for all Embedded MATLAB Function blocks in your Simulink model. Follow these steps:

- 1 Open your Embedded MATLAB Function block.
- 2 In the Embedded MATLAB Editor, select **Tools > Open Simulation Target**.

The Configuration Parameters dialog box opens with Simulation Target selected.

- 3 Under the **Embedded MATLAB and Stateflow** panel, clear the **Use BLAS library for faster simulation** check box and click **Apply**.



Supported Compilers

Embedded MATLAB function blocks use the BLAS library on all C compilers **except**:

- Watcom
- Intel
- Borland

The default MATLAB compiler, `lcc`, supports the BLAS library. To install a different C compiler, use the `mex -setup` command, as described in “Building MEX-Files” in the MATLAB External Interfaces documentation.

Controlling Runtime Checks

In this section...
“Types of Runtime Checks” on page 30-159
“When to Disable Runtime Checks” on page 30-160
“How to Disable Runtime Checks” on page 30-160

Types of Runtime Checks

In simulation, the code generated for your Embedded MATLAB Function block includes the following runtime checks:

- Memory integrity checks

These checks detect violations of memory integrity in code generated for Embedded MATLAB Function blocks and stop execution with a diagnostic message.

Caution For safety, these checks are enabled by default. Without memory integrity checks, violations will result in unpredictable behavior.

- Responsiveness checks in code generated for Embedded MATLAB Function blocks

These checks enable periodic checks for Ctrl+C breaks in code generated for Embedded MATLAB functions. 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 Runtime Checks

Generally, generating code with runtime checks enabled results in more lines of generated code and slower simulation than generating code with the checks disabled. Disabling runtime 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 Runtime Checks

Embedded MATLAB Function blocks enable runtime checks by default, but you can disable them explicitly for all Embedded MATLAB Function blocks in your Simulink model. Follow these steps:

- 1 Open your Embedded MATLAB Function block.
- 2 In the Embedded MATLAB Editor, select **Tools > Open Simulation Target**.

The Configuration Parameters dialog box opens with Simulation Target selected.

- 3 Under the **Embedded MATLAB and Stateflow** panel, clear the **Ensure memory integrity** or **Ensure responsiveness** check boxes, as applicable, and click **Apply**.

PrintFrame Editor

- “PrintFrame Editor Overview” on page 31-2
- “Designing the Print Frame” on page 31-8
- “Specifying the Print Frame Page Setup” on page 31-9
- “Creating Borders (Rows and Cells)” on page 31-11
- “Adding Information to Cells” on page 31-14
- “Changing Information in Cells” on page 31-18
- “Saving and Opening Print Frames” on page 31-22
- “Printing Block Diagrams with Print Frames” on page 31-23
- “Example” on page 31-26

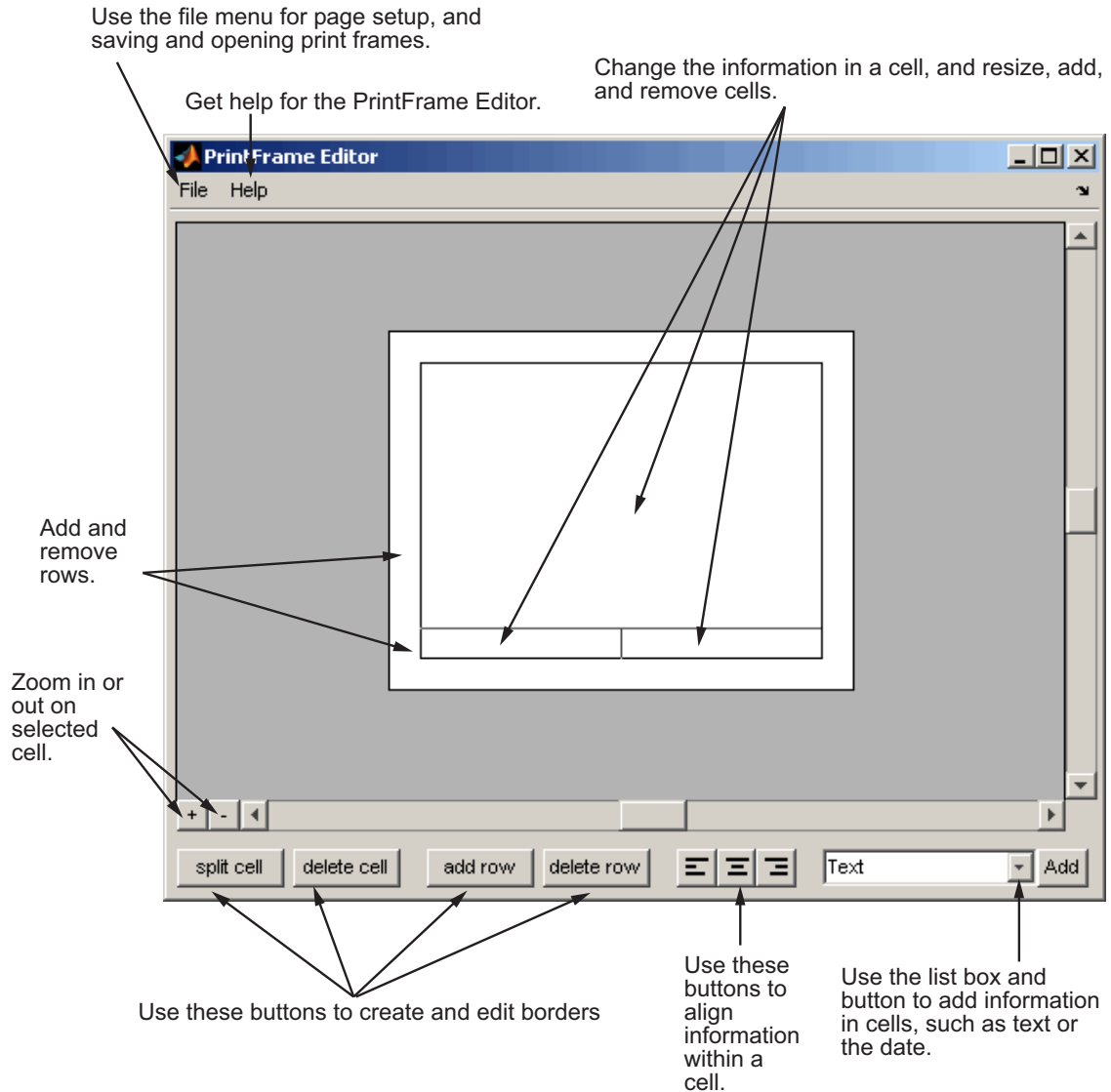
PrintFrame Editor Overview

In this section...
“About the Print Frame Editor” on page 31-2
“What PrintFrames Are” on page 31-3
“Starting the PrintFrame Editor” on page 31-6
“Getting Help for the PrintFrame Editor” on page 31-7
“Closing the PrintFrame Editor” on page 31-7
“Print Frame Process” on page 31-7

About the Print Frame Editor

The PrintFrame Editor is a graphical user interface you use to create and edit print frames for block diagrams created with the Simulink software and the Stateflow product. This chapter outlines the PrintFrame Editor, accessible with the `frameedit` command.

The following figure describes the general layout of the PrintFrame Editor.

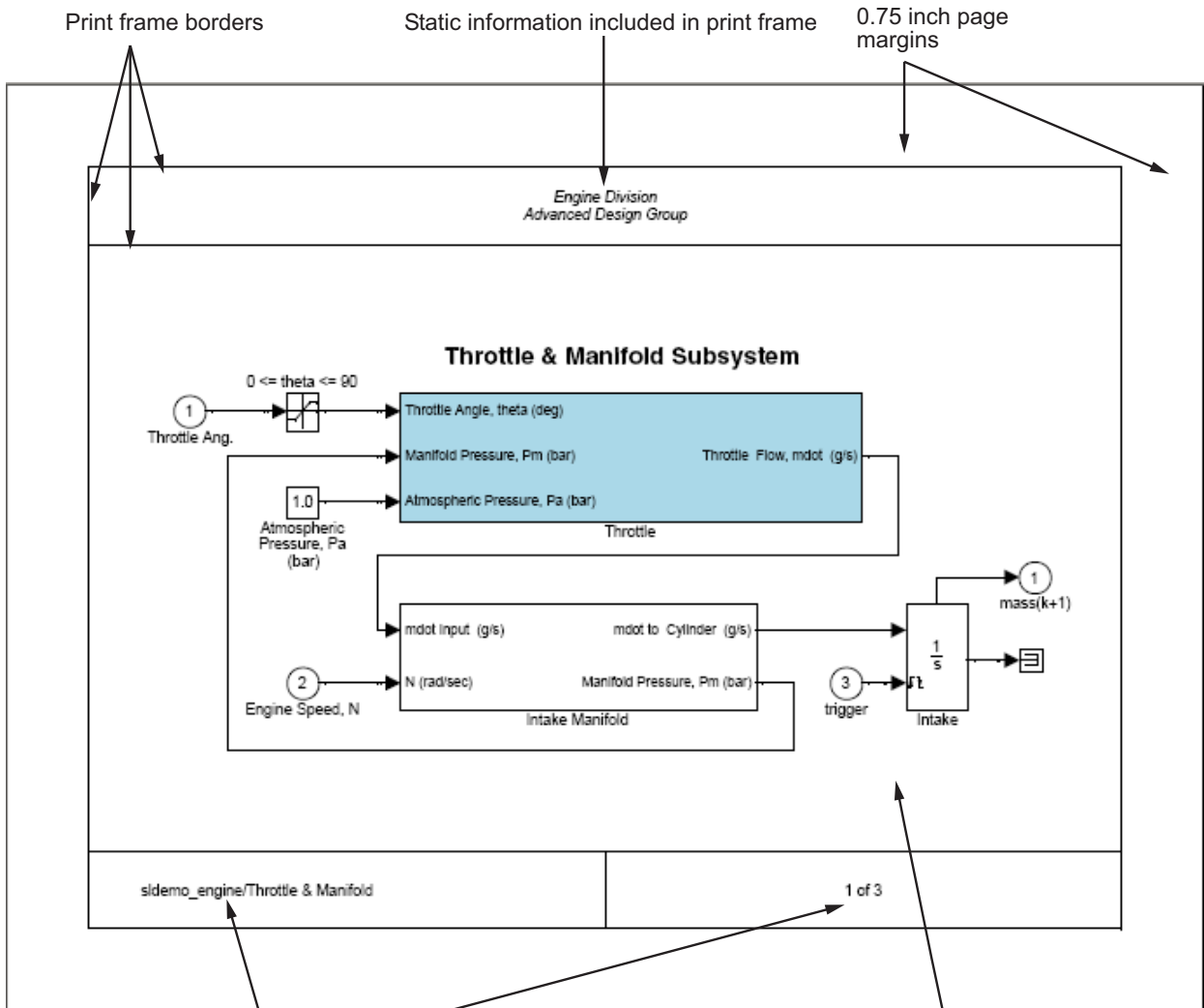


What PrintFrames Are

Print frames are borders containing information relevant to the block diagram, for example, the name of the block diagram. After creating a print

frame, you can use the Simulink software or the Stateflow product to print a block diagram with a print frame.

This illustration shows an example of a print frame with the major elements labeled.



Variable information included in print frame

Simulink block diagram

See the "Example" on page 31-26 for specific instructions to create this print frame.

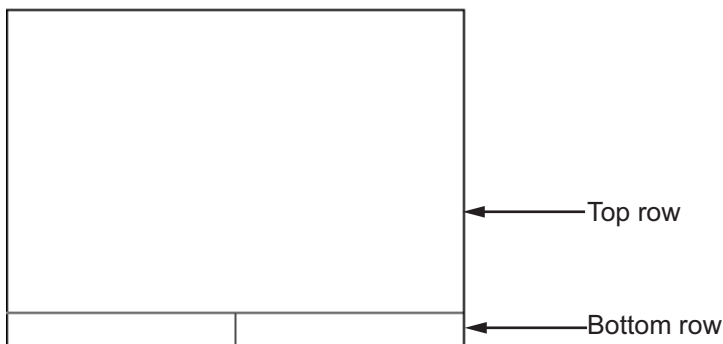
Starting the PrintFrame Editor

Type `frameedit` at the MATLAB prompt. The **PrintFrame Editor** window appears. The **PrintFrame Editor** window opens with the default print frame.

You can use `frameeditfilename` to open the **PrintFrame Editor** window with the specified filename, where `filename` is a figure file you previously created and saved using `frameedit`.

Default Print Frame

The default print frame has two rows. The top row consists of one cell and the bottom row has two cells.



You can add information entries to these cells. You can also add new rows and cells and add information in them, or change entries to different ones.

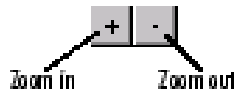
Zooming In and Out

While using the PrintFrame Editor, you might need to zoom in on an area to better see the information or cell.

- 1 Click in the area you want to zoom in on.

This selects a cell.

- 2 Click the zoom in button.



The area is magnified.

3 Click the zoom in button repeatedly to continue zooming in.

To zoom out, reducing magnification in an area, click the zoom out button. Click the zoom out button repeatedly to continue zooming out.

Getting Help for the PrintFrame Editor

Select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor window to access this online help.

Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Print Frame Process

These are the basic steps for creating and using print frames:

- “Designing the Print Frame” on page 31-8
- “Specifying the Print Frame Page Setup” on page 31-9
- “Creating Borders (Rows and Cells)” on page 31-11
- “Adding Information to Cells” on page 31-14
- “Changing Information in Cells” on page 31-18
- “Saving and Opening Print Frames” on page 31-22
- “Printing Block Diagrams with Print Frames” on page 31-23

See also the “Example” on page 31-26.

Designing the Print Frame

In this section...
“Before You Begin” on page 31-8
“Variable and Static Information” on page 31-8
“Single Use or Multiple Use Print Frames” on page 31-8

Before You Begin

Before you create a print frame using the PrintFrame Editor, consider the type of information you want to include in it and how you want the information to appear. You might want to make a sketch of how you want the print frame to look, and note the wording you want to use.

Variable and Static Information

In a print frame, you can include variable and static information. Variable information is automatically supplied at the time of printing, for example, the date the block diagram is being printed. Static information always prints exactly as you entered it, for example, the name and address of your organization.

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 with different block diagrams.

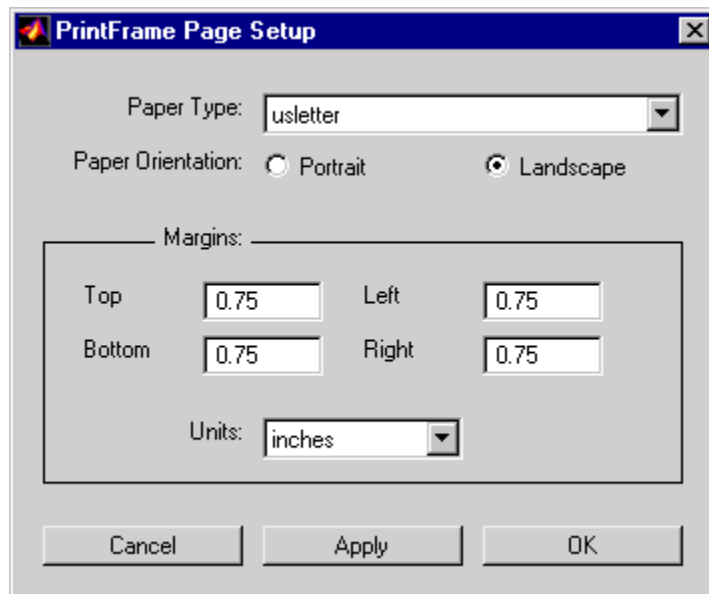
Specifying the Print Frame Page Setup

After you have an idea of the design of your print frame, specify the page setup for the print frame.

Note Always begin creating a new print frame with **PrintFrame Page Setup**. If, instead, you begin by creating borders and adding information, and then later change the page setup, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the page setup paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

- 1 In the **PrintFrame Editor** window, select **Page Setup** from the **File** menu.

The **PrintFrame Page Setup** dialog box appears.



- 2 In the dialog box, specify:

- **Paper Type** – for example, usletter
 - **Paper Orientation** – portrait or landscape
 - **Margins** for the print frame and the **Units** in which to specify the margins
- 3** Click **Apply** to see the effects of the changes you made. Then click **OK** to close the dialog box.

Creating Borders (Rows and Cells)

In this section...

“First Steps” on page 31-11
“Adding and Removing Rows” on page 31-11
“Adding and Removing Cells” on page 31-12
“Resizing Rows and Cells” on page 31-12
“Print Frame Size” on page 31-12

First Steps

Once you have set up the page, use the PrintFrame Editor to specify borders (cells) in which the block diagram and information will be placed.

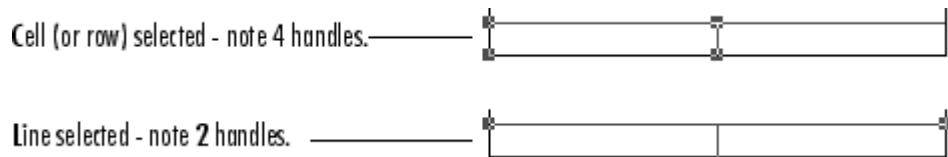
Important Always specify the PrintFrame page setup before creating borders and adding information (see “Specifying the Print Frame Page Setup” on page 31-9). Otherwise, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

Adding and Removing Rows

You can add and remove rows in a print frame.

- 1 Click within an existing row to select it. If a row consists of multiple cells, click in any of the cells in the row to select that row.

When a row is selected, handles appear on all four corners. If handles appear on only two corners, you clicked on and only selected the line, not the row.



- 2 Click the **add row** button to create a new row.

The new row appears above the row you selected.

- 3 To remove a row, select the row and click the **delete row** button.

Adding and Removing Cells

You can create multiple cells within a row.

- 1 Select the row in which you want multiple cells.
- 2 Click the **split cell** button.

The row splits into two cells. If the row already consists of more than one cell, the selected cell splits into two cells.

- 3 To remove a cell, select the cell and click the **delete cell** button.

Resizing Rows and Cells

You can change the dimensions of a row or cell.

- 1 Click on the line you want to move.

A handle appears on both ends of the line.

- 2 Drag the line to the new location.

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.

Print Frame Size

Note that the overall size of the print frame is based on the options you specify using the page setup feature. Therefore, when you change the dimensions

of one row or cell, the dimensions of the row or cell next to it change in an inverse direction. For example, if you drag the top line of a row to make it taller, the row above it becomes shorter by the same amount.

To change the overall dimensions of the print frame, use the page setup feature. See “Specifying the Print Frame Page Setup” on page 31-9.

Adding Information to Cells

In this section...

“Procedure for Adding Information to Cells” on page 31-14

“Text Information” on page 31-15

“Variable Information” on page 31-15

“Multiple Entries in a Cell” on page 31-16

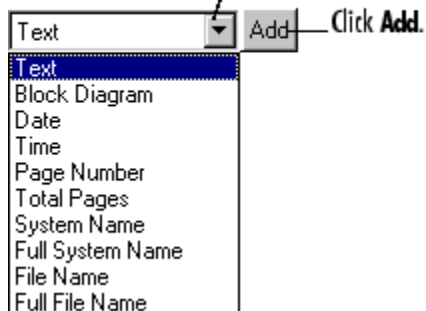
Procedure for Adding Information to Cells

Use the following steps to add information to cells.

- 1 Select the cell where you want to add information.
- 2 From the list box, select the type of information you want to add.
- 3 Click the **Add** button.

An edit box containing that information appears in the cell. (The edit boxes for your platform might look slightly different from those in the figure below.)

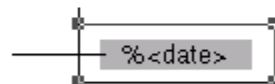
Select an item from the list.



For **Text**, an empty edit box appears.



For variable information, the variable's name appears in the edit box. In this example, the variable information is **Date**.



- 4 Click outside of the edit box to end editing mode.

Note If you click the **Add** button and nothing happens, it might be because you did not select a cell first.

Text Information

For **Text**, type the text you want to include in that cell, for example, the name of your organization. Press the **Enter** key if you want to type additional text on a new line. Note that you can type special characters, for example, superscripts and subscripts, Greek letters, and mathematical symbols. For special characters, use embedded TeX sequences (see the `text` command `String` property (in Text Properties of the online documentation) for a list of allowable sequences). Click outside of the edit box when you are finished to end editing mode.

Variable Information

All of the items in the information list box, except for the **Text** item, are for adding variable information, which is supplied at the time of printing. When you print a block diagram with a print frame that contains variable information, the information for that particular block diagram prints in those fields.

Types of Variable Information

The variable entries you can include are:

- **Block Diagram** — This entry indicates where the block diagram is to be printed. **Block Diagram** is a mandatory entry. If **Block Diagram** is not in one of the cells, you cannot save the print frame and therefore cannot print a block diagram with it.
- **Date** — The date that the block diagram and print frame are printed, in `dd-mmm-yyyy` format, for example, `05-Dec.-1997`.
- **Time** — The time that the block diagram and print frame are printed, in `hh:mm` format, for example, `14:22`.
- **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 filename of the block diagram, for example, sldemo_engine.mdl.
- **Full File Name** — The full path and filename for the block diagram, for example, \\matlab\toolbox\simulink\simdemos\automotive\sldemo_engine.mdl.

Note: Adding the system name or filename does not mean that you can then specify a filename for the Simulink software or the StateFlow product in the PrintFrame Editor. It means that when you print a block diagram and specify that it print with a print frame, the system name or filename of the Simulink software or the Stateflow product block diagram is printed in the specified cell of the print frame.

Format for Variable Information

When you add a variable entry, a percent sign, %, is automatically included to identify the entry as variable information rather than a text string. In addition, the type of entry, for example, page, appears in angle brackets, <>. The entry consists of the entire string, for example, %<page>, for **Page Number**.

Multiple Entries in a Cell

You can include multiple entries in one cell.

- 1 Select the cell.
- 2 Add another item from the list box.

The new entry is added after the last entry in that cell.

You can also type descriptive text to any of the variable entries without using the **Text** item in the information list box.

1 Double-click in the cell.

An edit box appears around the entry.

2 Type text in the edit box before or after the entry.

3 Click anywhere outside of the edit box to end editing mode.

Note You cannot include multiple entries or text in the cell that contains the block diagram entry. `%<blockdiagram>` must be the only information in that cell. If there is any other information in that cell, you cannot save the print frame and therefore cannot print it with a block diagram.

Changing Information in Cells

In this section...

“Aligning the Information in a Cell” on page 31-18

“Editing Text Strings” on page 31-18

“Removing and Copying Entries” on page 31-19

“Changing the Font Characteristics” on page 31-20

Aligning the Information in a Cell

To align the information within a cell:

- 1 Click within the cell to select it.
- 2 Click on one of the **Align** buttons for left, center, or right alignment.



The information aligns within the cell.

Alignment does not apply to the cell that contains the `%<blockdiagram>` entry. The block diagram is automatically scaled and centered to fit in that cell at the time of printing.

Editing Text Strings

You can change text you typed in a cell:

- 1 Double-click the information you want to edit.

An edit box appears around all of the information in that cell.

- 2 Click at the start of the text you want to change and drag to the end of the text to be changed.

This highlights the text.

- 3 Type the replacement text.

It automatically replaces the highlighted text.

- 4 Click anywhere outside of the edit box to end editing mode.

Note Be careful not to edit the text of a variable entry, because then the variable information will not print. For example, if you accidentally remove the % from the %<page> entry, the text <page> will print instead of the actual page number.

Removing and Copying Entries

You can cut, copy, paste, or delete an entry:

- 1 Double-click the information you want to remove or copy.

An edit box appears around all of the information in that cell.

- 2 Click at the start of the entry you want to edit and drag to the end of that entry. This highlights the entry.

For variable information, be sure to include the entire string, for example, %<page>.

Note that for computers running the Microsoft Windows operating system, you can select all of the entries in a cell by right-clicking the information and choosing **Select All** from the pop-up menu.

- 3 Use the standard editing techniques for your platform to cut, copy, or delete the highlighted information.
 - For computers running the Microsoft Windows operating system, right-click in the edit box and select **Cut**, **Copy**, or **Delete** from the pop-up menu.
 - For UNIX based systems, highlighting the information automatically copies it to the clipboard. If you want to remove it, press the **Delete** key.

If you make a mistake, use your platform's standard undo technique. For example, for computers running the Microsoft Windows operating system, right-click in the edit box and select **Undo** from the pop-up menu.

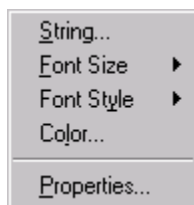
- 4** If you cut or copied the information to the clipboard and want to paste it, double-click the entry where you want to paste it and position the cursor at the new location in that edit box. Then use the standard paste technique for your platform.
 - For computers running the Microsoft Windows operating system, right-click at the new location and select **Paste** from the pop-up menu.
 - For UNIX based systems, click at the new location and then click the middle mouse button.
- 5** Click somewhere outside of the edit box to end editing mode.

Changing the Font Characteristics

You can change the font characteristics for the information in any cell. Specifically, you can specify the font size, style, color, and family.

- 1** Right-click the information in the cell.

The information in the cell is selected and the pop-up menu for changing font characteristics appears.



If this pop-up menu does not appear, it is because you were in edit mode. To get the font pop-up menu, click somewhere outside of the edit box surrounding the information and then right-click.

- 2** Select an item from the pop-up menu. Choose **Properties** if you want to change the font family or if you want to change multiple characteristics at once.

Note that you can also select **String** from the pop-up menu, which allows you to edit the text string.

- 3** Select the new font characteristic(s) for that cell. For example, for **Font Size**, select the new size from its pop-up menu.

Note that changing the font characteristics for the %<blockdiagram> entry is not relevant and does nothing.

Saving and Opening Print Frames

In this section...
“Saving a Print Frame” on page 31-22
“Opening a Print Frame” on page 31-22

Saving a Print Frame

You must save a print frame to print a block diagram with that print frame. To save a print frame:

- 1 Select **Save As** from the **File** menu.

The **Save As** dialog box appears.

- 2 Type a name for the print frame in the **File name** edit box.

- 3 Click the **Save** button.

The print frame is saved as a figure file, which has the `.fig` extension. A figure file is a binary file used for print frames.

Opening a Print Frame

You can open a saved print frame in the PrintFrame Editor, make changes to it, and save it under the same or a different name. To open an existing print frame:

- 1 Select **Open** from the **File** menu.

- 2 Select the print frame you want to open.

All print frames are figure files.

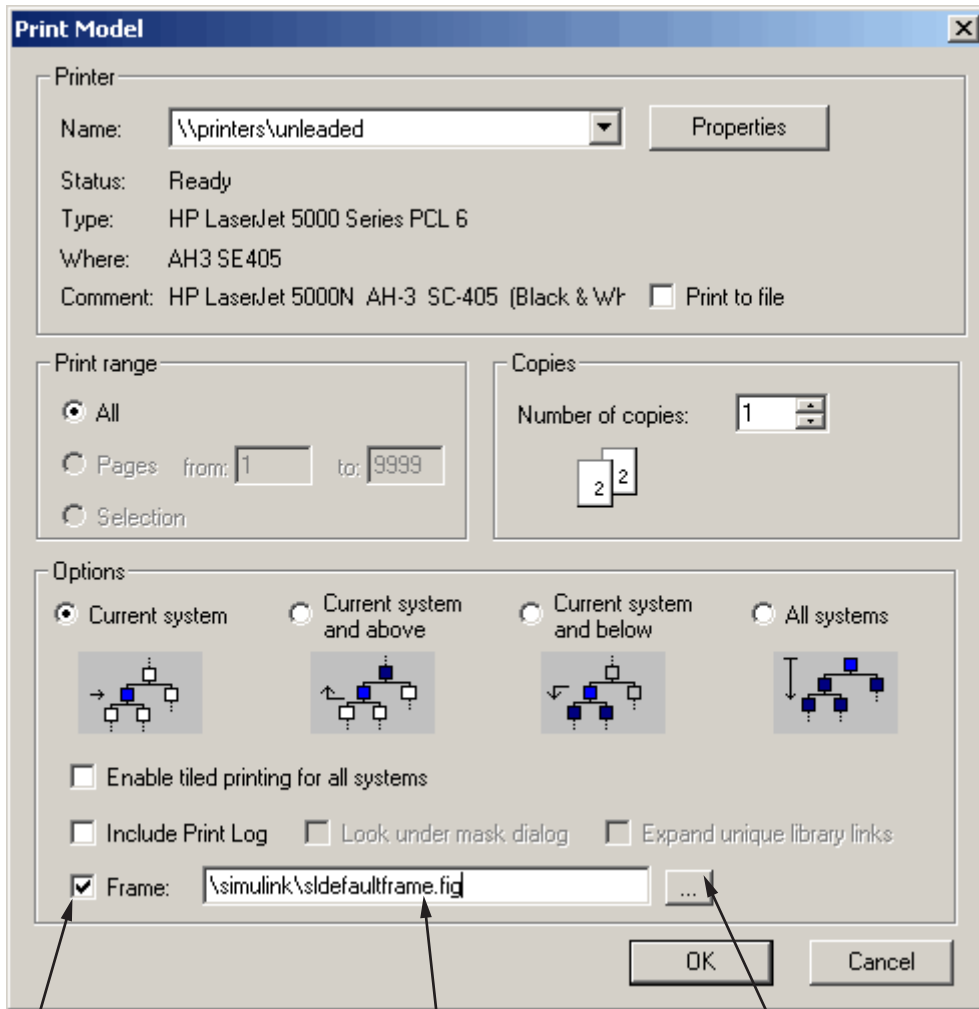
Alternatively, you can open a print frame from the MATLAB prompt. Type `frameedit filename` and the PrintFrame Editor opens with the print frame file you specified.

Printing Block Diagrams with Print Frames

When using the Simulink software or Stateflow product, you can print a block diagram with the print frame.

- 1 Select **Print** from the **File** menu.

The **Print Model** dialog box appears. The dialog box shown below is for computers running the Microsoft Windows operating system. The dialog box for your platform might look slightly different.



Check this box to print the block diagram with a print frame.

Type the path and filename of the print frame you want the block diagram to print with,

or click the ... button and then select the print frame file.

In the **Print Model** dialog box:

- 1 Select the **Frame** check box.

- 2 Supply the filename for the print frame you want to use. Either type the path and filename directly in the edit box, or click the ... button and select the print frame file you saved using the PrintFrame Editor.

Note that the default print frame filename, `sldefaultframe.fig`, appears in the filename edit box until you specify a different filename.

- 3 Specify other printing options in the **Print** dialog box. For example, for computers running the Microsoft Windows operating system, specify options under **Properties**.

Note Specify the paper orientation for printing the way you normally would. The paper orientation you specified in the PrintFrame Editor's **PrintFrame Page Setup** dialog box is not the same as the paper orientation used for printing. For example, assume you specified a landscape-oriented print frame in the PrintFrame Editor. If you want the printed page to have a landscape orientation, you must specify that at the time of printing. For example, for computers running the Microsoft Windows operating system, click the **Properties** button in the **Print Model** dialog box, and for **Page Setup**, specify the **Orientation** as **Landscape**.

- 4 Click **OK** in the **Print** dialog box.

The block diagram prints with the print frame you specified.

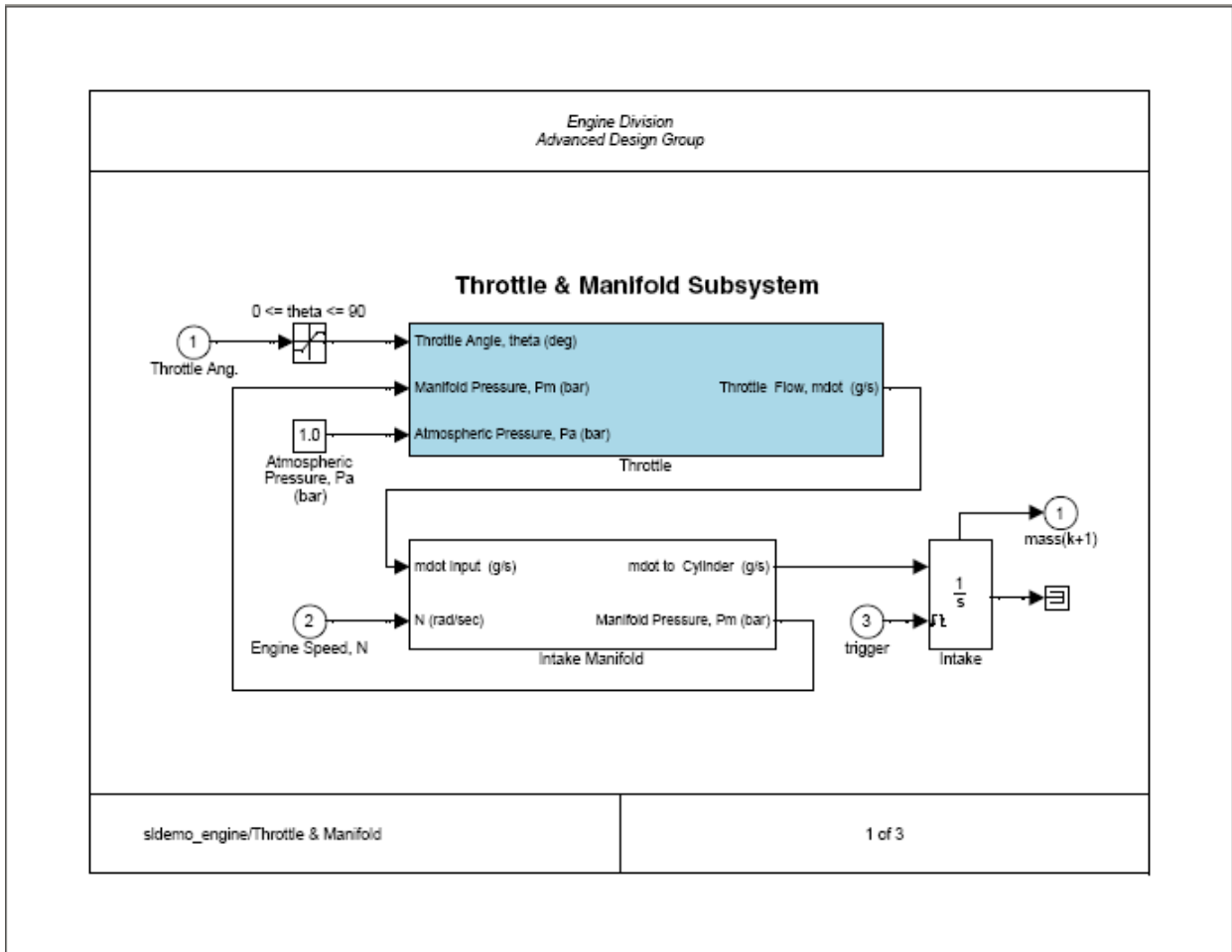
See also the example, "Print the Block Diagram with the Print Frame" on page 31-30.

Example

In this section...
“About the Example” on page 31-26
“Create the Print Frame” on page 31-27
“Print the Block Diagram with the Print Frame” on page 31-30

About the Example

This example uses a Simulink software demo engine model. It involves two parts — first creating a print frame, and then printing the engine model with that print frame. The result looks similar to the figure below.



Create the Print Frame

1 At the MATLAB prompt, type `frameedit`.

The **PrintFrame Editor** window appears.

2 Set up the page:

- a** Select **Page Setup** from the **File** menu.

The **PrintFrame Page Setup** dialog box opens.

- b** For **Paper Type**, select a size that is appropriate for your printer.
- c** For this example, keep the **Paper Orientation** as **Landscape** and the **Margins** set to **0.75 inches**.
- d** Click the **OK** button.

The dialog box closes. The print frame you see in the **PrintFrame Editor** window will reflect your changes.

- 3** Add the information entries `%<blockdiagram>`, `%<fullsystem>`, and `%<page>`.
 - a** Click within the upper row in the print frame.
 - b** From the bottom right list box, select **Block Diagram**, then click **Add**. The `%<blockdiagram>` appears in the row.
 - c** Click within the lower left cell in the print frame.
 - d** From the bottom right list box, select **Full System Name**, then click **Add**. The `%<fullsystem>` appears in the cell.
 - e** Click within the lower right cell in the print frame.
 - f** From the bottom right list box, select **Page Number**, then click **Add**. The `%<page>` appears in the cell.
- 4** Add a row at the top.
 - a** Click within the upper row in the print frame, the row that contains the `%<blockdiagram>` entry.

Be sure that handles appear on all four corners of the row.
 - b** Click the **add row** button.

A new row appears at the top, above the row you selected.
- 5** Make the new row shorter.
 - a** Click on the horizontal line that separates the top row (the row you just added) from the row beneath it (the row containing the `%<blockdiagram>` entry).

Be sure that only two handles appear, one at each end of the line. If you see four handles in either row, click directly on the horizontal line and the other two handles disappear.

- b** Drag the line up until the top row is about the same height as the row at the bottom of the print frame.

6 Add information in the top row.

- a** Click anywhere within the top row (the row you just added).
- b** Select **Text** from the information list box.
- c** Click the **Add** button.

An edit box appears in the cell.

- d** Type **Engine Division**, press the **Enter** key to advance the cursor to the next line, and then type **Advanced Design Group**.

Click the zoom in button if you need to magnify the entry.

- e** Click outside of the edit box to end editing mode.

7 In the left cell of the bottom row, align the information on the left.

- a** Click the zoom out button if you need to.
- b** Click within the left cell of the bottom row to select it.
- c** Click the left alignment button.

The entry moves to the left.

8 Make the information in the top row appear in italics.

- a** Right-click on the entry in the row.
- b** Select **Font Style** from the pop-up menu.

If the pop-up menu for font properties does not appear, you are in editing mode. Click outside of the edit box to end editing mode and then right-click the text to access the pop-up menu.

- c** From the **Font Style** pop-up menu, select **italic**.

The entry in the cell appears in italics and the information will appear in italics when the print frame is printed with a Simulink diagram.

- 9 Add the total number of pages to the right cell in the bottom row.
 - a Click within the cell to select it.
 - b Add the total pages entry: select **Total Pages** from the information list box and click the **Add** button.

The %<npages> entry appears after the %<page> entry. If you need to, zoom in to see the entry.

- c Add the text **of** after the page number entry. Click the cursor after the %<page> entry, and then type **of** (type a space before and after the word).

The information in the cell now is: %<page> of %<npages>.

- 10 Save the print frame: select **Save As** from the **File** menu. In the **Save Frame** dialog box, type engdiv1 for the **File name**. Click the **Save** button.

The print frame is saved as a figure file.

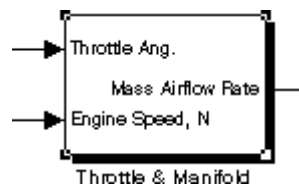
- 11 You can close the **PrintFrame Editor** window by clicking the close box.

Print the Block Diagram with the Print Frame

- 1 To view the Simulink engine model, type `sldemo_engine` at the MATLAB prompt.

The engine model appears in a Simulink window.

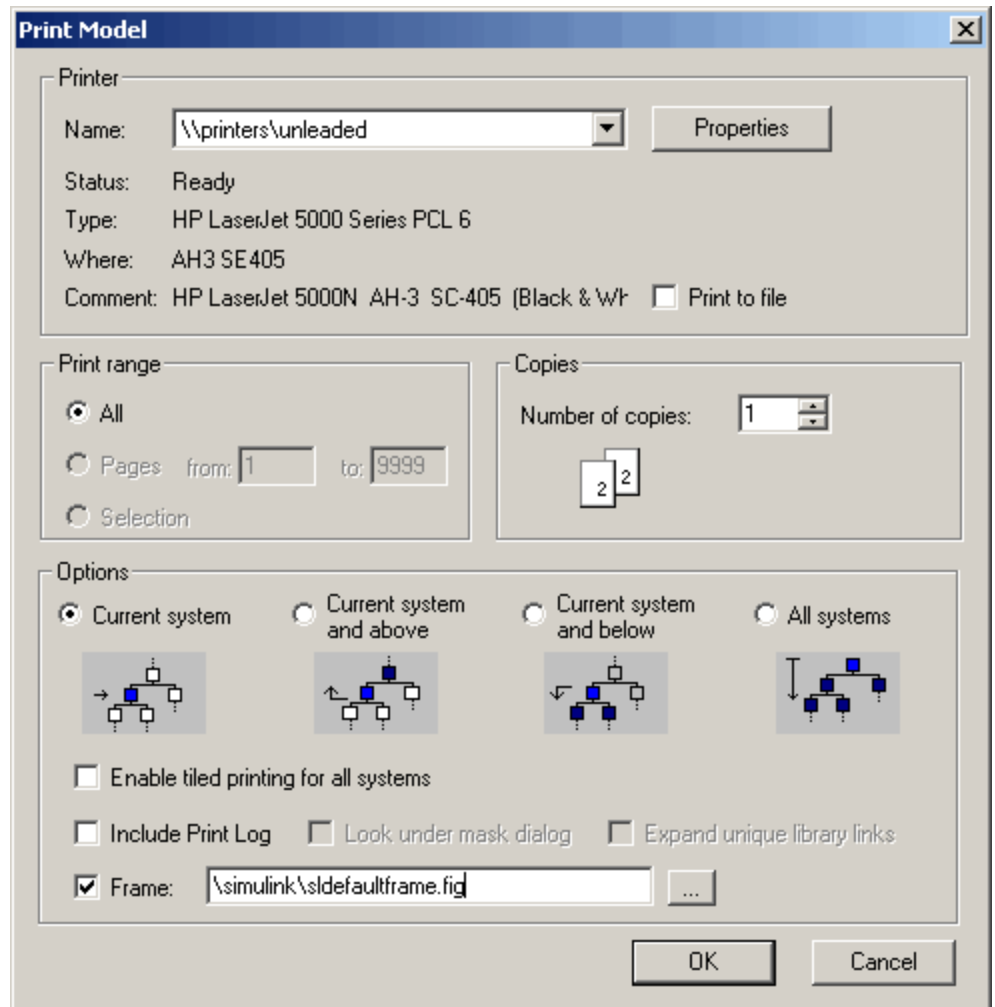
- 2 Double-click the Throttle & Manifold block.



The Throttle & Manifold subsystem opens in a new window.

- 3 In the **Throttle & Manifold** window, select **Print** from the **File** menu.

The **Print Model** dialog box opens with the default settings as shown here.



- 4 In the **Print Model** dialog box, set the page orientation to landscape. This example uses the techniques for computers running the Microsoft Windows

operating system. Use the methods for your own platform to change the page orientation for printing.

- a** Click the **Properties** button.

The **Document Properties** dialog box opens.

- b** Go to the **Page Setup** tab.
- c** For **Orientation**, select **Landscape**.
- d** Click **OK**.

The **Document Properties** dialog box closes.

- 5** In the **Print Model** dialog box, under **Options**, select **Current system and below**.

This specifies that the Throttle & Manifold block diagram and its subsystems will print.

- 6** Check the **Frame** check box.

- 7** Specify the print frame to use.

- a** Click the ... button.
- b** In the **Frame File Selection** dialog box, find the filename of the print frame you just created, `engdiv1.fig`, and select it.
- c** Click the **Open** button.

The path and filename appear in the **Frame** edit box.

- 8** Click **OK** in the **Print Model** dialog box.

The Throttle & Manifold block diagram prints with the print frame; it should look similar to the figure shown at the start of this example.

In addition, the Throttle block diagram and the Intake Manifold block diagram print because you specified printing of the current system and its subsystems. These block diagrams also print with the `engdiv1` print frame, but note that their variable information in the print frame is different.

library

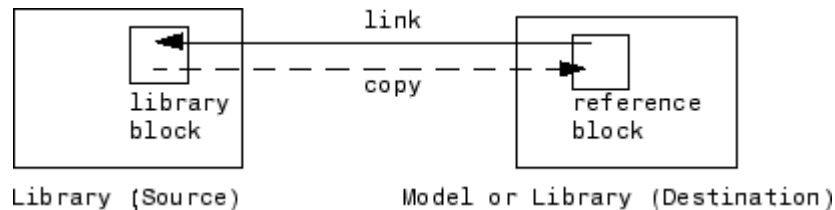
A collection of blocks intended to serve as prototypes for creating instances of block types in models. Simulink software uses a special type of model file to store block libraries.

library block

A block in a library that serves as a prototype for creating instances of the block's type in models. You create an instance of the block type by copying the library block into a model.

library link

A connection between a library reference block and a library block that allows Simulink to update the reference block when the library block changes. Simulink creates the link when you create the reference block.



parameterized link

A link to a library block from a reference block whose parameter values differ from those of the library block. Parameterized links reference block parameter values to differ from library block parameter values without losing the advantage of automatic updating of reference blocks when library blocks change.

reference block

An instance of a block type in a model linked to a library block that serves as the block type's prototype.

sublibrary

A block library included in another library.

Examples

Use this list to find examples in the documentation.

Simulink Basics

- “Updating a Block Diagram” on page 1-27
- “Positioning and Sizing a Diagram” on page 1-31
- “Print Command” on page 1-36
- “Adding Items to Model Editor Menus” on page 28-2
- “Disabling and Hiding Model Editor Menu Items” on page 28-13
- “Disabling and Hiding Dialog Box Controls” on page 28-15

How Simulink Works

- “Algebraic Loops” on page 2-35
- “How Propagation Affects Inherited Sample Times” on page 3-31

Creating a Model

- “Creating a Model Template” on page 4-2
- “Creating Annotations Programmatically” on page 4-35
- “Creating a Subsystem by Grouping Existing Blocks” on page 4-38
- “How to Discretize Blocks from the Simulink Model” on page 4-127
- “Enabled Subsystems” on page 5-4
- “Triggered Subsystems” on page 5-14
- “Triggered and Enabled Subsystems” on page 5-19
- “Conditional Execution Behavior” on page 5-25
- “Data Store Examples” on page 16-13

Executing Commands From Models

- “Loading Variables Automatically When Opening a Model” on page 4-63
- “Executing an M-file by Double-Clicking a Block” on page 4-64
- “Executing Commands Before Starting Simulation” on page 4-65

Working with Blocks

“Making Backward-Compatible Changes to Libraries” on page 8-16

Creating Block Masks

“Masked Subsystem Example” on page 9-5

“Setting Masked Block Dialog Parameters” on page 9-52

“Self-Modifying Mask Example” on page 9-56

Working with Lookup Tables

“Example Output for Lookup Methods” on page 17-26

“Example of a Logarithm Lookup Table” on page 17-39

Creating Custom Simulink Blocks

“Tutorial: Creating a Custom Block” on page 29-18

Symbols and Numerics

% 31-16
< > 31-16

A

Abs block
 zero crossings 2-32
absolute tolerance
 definition 21-17
 simulation accuracy 23-10
accelbuild command
 building Accelerator MEX-file 27-24
Acceleration
 code regeneration 27-7
 compilation overhead 23-4
 debugger advantages 27-27
 decision tree 27-12
 designing for 27-14
 how to run debugger with 27-27
 inhibiting 27-17
 numerical precision 27-15
 verses normal mode 27-15
 trade-offs 27-10
Accelerator
 description 27-2
 determining why it rebuilds 27-7
 how it works 27-3
 making run time changes 27-23
 Simulink blocks whose performance is not
 improved by 27-14
 switching back to normal mode 27-28
 using with Simulink debugger 27-27
Accelerator and Rapid Accelerator
 choosing between 27-10
 comparing 27-10
Accelerator mode
 keywords 27-18
Accelerators
 customizing build process 27-20

 interacting programmatically 27-24
 running 27-21
Action Port block
 in subsystem 4-45
activating
 configuration references 20-24
Adams-Bashforth-Moulton PECE solver 21-16
Add button 31-14
adding
 cells 31-12
 rows 31-11
 text to cells 31-14
 variable information to cells 31-14
algebraic loops
 direct feedthrough blocks 2-35
 highlighting 26-41
 identifying blocks in 26-38
 simulation speed 23-3
aligning
 information in cells 31-18
aligning blocks 7-11
annotations
 changing font 4-27
 creating 4-26
 definition 4-26
 deleting 4-27
 editing 4-26
 moving 4-26
 using symbols and Greek letters in 4-33
 using TeX formatting commands in 4-33
 using to document models 18-7
Assignment block
 and For Iterator block 4-51
associating
 bus objects
 with bus creator blocks 12-11
 with inports and outports 12-12
 with model entities 12-11
asynchronous sample time 3-19
atomic subsystem 2-13

- attaching
 - configuration references 20-17
 - to additional models 20-22
- attributes format string 4-23
- AttributesFormatString block parameter 7-33
- avoiding
 - mux/bus mixtures 12-63
- B**
- Backlash block
 - zero crossings 2-32
- backpropagating sample time 3-33
- Backspace key
 - deleting annotations 4-27
 - deleting blocks 7-14
 - deleting labels 10-8
- Band-Limited White Noise block
 - simulation speed 23-3
- block
 - library Glossary-1
 - reference Glossary-1
 - simulation terminated or suspended 21-5
- block callback parameters 4-58
- block callbacks
 - adding custom functionality 29-37
- Block data tips 7-2
- block diagram
 - updating 1-27
- block diagram entry 31-15
- block diagrams
 - panning 1-23
 - printing 1-29
 - zooming 1-22
- block libraries
 - adding to Library Browser 8-24
 - creating 8-14
 - locking 8-16
 - modifying 8-15
 - searching 4-5
- block library
 - definition Glossary-1
- block names
 - changing location 7-40
 - copied blocks 7-10
 - editing 7-39
 - flipping location 7-40
 - generated for copied blocks 7-10
 - hiding and showing 7-40
 - location 7-39
 - rules 7-39
- block parameters
 - about 7-15
 - displaying beneath a block 7-52
 - modifying during simulation 21-19
 - scalar expansion 10-28
 - setting 7-16
- block priorities
 - assigning 7-52
- Block Properties dialog box 7-27
- blocks
 - aligning 7-11
 - assigning priorities 7-52
 - associating user data with 13-57
 - autoconnecting 4-15
 - bus-capable 12-9
 - callback routines 4-54
 - changing font 7-39
 - changing font names 7-39
 - changing location of names 7-40
 - checking connections 2-19
 - connecting automatically 4-15
 - connecting manually 4-18
 - copying from Library Browser 4-6
 - copying into models 7-9
 - custom, Simulink 29-2
 - categories 29-3
 - code generation requirements 29-8
 - designing 29-18
 - examples 29-44

- incorporating legacy code 29-6
 - modeling requirements 29-7
 - simulation requirements 29-8
 - tutorial 29-18
 - using block callbacks 29-37
 - using MATLAB functions 29-3
 - using S-functions 29-4
 - using Subsystems 29-4
 - deleting 7-14
 - disconnecting 4-23
 - displaying sorted order on 7-46
 - drop shadows 7-38
 - duplicating 7-13
 - grouping to create subsystem 4-38
 - hiding block names 7-40
 - input ports with direct feedthrough 2-35
 - moving between windows 7-11
 - moving in a model 7-10
 - names
 - editing 7-39
 - orientation 7-34
 - resizing 7-37
 - reversing signal flow through 18-8
 - rotating 7-34
 - showing block names 7-40
 - updating 2-19
 - <>blocks 7-39
 - See also* block names
 - Bogacki-Shampine formula 21-16
 - borders
 - creating 31-11
 - important note about 31-11
 - bounding box
 - grouping blocks for subsystem 4-38
 - selecting objects 4-7
 - branch lines 4-19
 - Break Library Link menu item 8-11
 - breaking links to library block 8-11
 - breakpoints
 - setting 26-27
 - setting at end of block 26-30
 - setting at timesteps 26-31
 - setting on nonfinite values 26-31
 - setting on step-size-limiting steps 26-31
 - setting on zero crossings 26-31
 - Browser 19-26
 - building models
 - tips 18-6
 - bus creator blocks
 - associating
 - with bus signals 12-11
 - Bus Editor 12-10
 - opening 12-14
 - bus objects
 - associating
 - with bus creator blocks 12-11
 - with inports and outports 12-12
 - with model entities 12-11
 - creating
 - with the API 12-47
 - with the Bus Editor 12-17
 - using 12-10
 - bus signals
 - created by Mux blocks 12-58
 - used as vectors 12-59
 - Bus to Vector block
 - backward compatibility 12-63
 - bus-capable blocks 12-9
 - buses 12-51
 - connecting to inports 12-51
 - mixing with muxes 12-57
 - nesting 12-7
 - busses used as muxes
 - correcting 12-62
- ## C
- callback routines 4-54
 - callback routines, referencing mask parameters
 - in 4-58

- callback tracing 4-55
- canvas, editor 1-19
- cells
 - adding 31-12
 - adding text to 31-15
 - adding variable information to 31-15
 - changing information in 31-18
 - deleting 31-12
 - resizing 31-12
 - selecting 31-11
 - splitting 31-12
- center alignment button 31-18
- changing
 - configuration references 20-23
 - fonts 31-20
 - information in cells 31-18
 - signal labels font 10-8
 - values using configuration references 20-25
- characters, special 31-15
- classes
 - enumerated 14-2
- Clear menu item 7-14
- Clipboard block callback parameter 4-58
- Clock block
 - example 25-3
- CloseFcn block callback parameter 4-58
- CloseFcn model callback parameter 4-56
- closing PrintFrame Editor 31-7
- color of text 31-20
- colors for sample times 3-33
- command line debugger for Embedded MATLAB
 - Function block 30-32
- commands
 - undoing 1-22
- Compare To Constant block
 - zero crossings 2-32
- Compare To Zero block
 - zero crossings 2-32
- compilation report keyboard shortcuts
 - Embedded MATLAB function block
 - compilation report 30-77
- compilation reports 30-66
 - description 30-66
- compiled sample time 3-20
- Compiled Size property for Embedded MATLAB
 - Function block variables 30-93
- compilers
 - supported for Embedded MATLAB Function blocks (code generation) 30-15
 - supported for Embedded MATLAB Function blocks (simulation) 30-15
- composite signals 12-2
- conditional execution behavior 5-25
- conditionally executed subsystem 2-12
- conditionally executed subsystems 5-2
- configurable subsystem 4-121
- Configuration Parameters dialog box
 - increasing Accelerator performance 23-8
- configuration references
 - activating 20-24
 - and building models 20-28
 - and generating code 20-28
 - attaching 20-17
 - to additional models 20-22
 - changing 20-23
 - changing values with 20-25
 - creating 20-17
 - limitations 20-28
 - obtaining handles 20-21
 - obtaining values with 20-25
- configuration sets
 - copying 20-16
 - creating 20-16
 - freestanding 20-12
 - reading from an M-file 20-17
 - referencing 20-12
 - using 20-2
- connecting
 - buses to inports 12-51

- buses to root-level inports 12-51
- connecting blocks 4-18
- ConnectionCallback
 - port callback parameters 4-62
- constant sample time 3-17
- context menu 1-20
- continuous sample time 3-16
- control flow diagrams
 - do-while 4-49
 - for 4-50
 - if-else 4-44
 - switch 4-46
 - while 4-47
- control flow subsystem 5-2
- control input 5-2
- control signal 5-2 10-18
- Control System Toolbox
 - linearization 25-5
- controlling runtime checks
 - Embedded MATLAB Function block 30-159
- Copy menu item 7-9
- CopyFcn block callback parameter 4-59
- copying
 - blocks 7-9
 - configuration sets 20-16
 - signal labels 10-7
- copying information among cells 31-19
- correcting
 - buses used as muxes 12-62
- Created model parameter 4-112
- creating
 - bus objects
 - with the API 12-47
 - with the Bus Editor 12-17
 - configuration references 20-17
 - configuration sets 20-16
 - custom Simulink blocks 29-2
 - freestanding configuration sets 20-16
- creating print frames 31-7
- Creator model parameter 4-112

- custom blocks
 - creating 29-2
- Customizing Accelerator Build
 - AccelVerboseBuild 27-26
 - SimCompilerOptimization 27-26
- Cut menu item 7-11

D

- dash-dot lines 10-18
- data
 - enumerated 14-2
- data range checking
 - Embedded MATLAB Function blocks 30-34
- data types
 - displaying 13-23
 - enumerated 14-2
 - propagation definition 13-24
 - specifying 13-6
- data types of Embedded MATLAB Function variables 30-86
- data, adding to Embedded MATLAB Function blocks 30-49
- date entry 31-15
- Dead Zone block
 - zero crossings 2-32
- debugger
 - running incrementally 26-19
 - setting breakpoints 26-27
 - setting breakpoints at time steps 26-31
 - setting breakpoints at zero crossings 26-31
 - setting breakpoints on nonfinite values 26-31
 - setting breakpoints on step-size-limiting steps 26-31
 - skipping breakpoints 26-24
 - starting 26-11
 - stepping by time steps 26-22
- debugging
 - breakpoints in Embedded MATLAB Function block function 30-25

- display variable values in Embedded MATLAB Function block function 30-31
 - displaying Embedded MATLAB Function block variables in MATLAB 30-32
 - Embedded MATLAB Function block
 - example 30-24
 - Embedded MATLAB Function block function 30-23 to 30-24
 - operations for debugging Embedded MATLAB functions 30-34
 - stepping through Embedded MATLAB Function block function 30-26
 - decimation factor
 - saving simulation output 15-32
 - default print frame 31-6
 - Delete key
 - deleting blocks 7-14
 - deleting signal labels 10-8
 - DeleteChildFcn block callback parameter 4-59
 - DeleteFcn block callback parameter 4-59
 - deleting
 - cells 31-12
 - rows 31-11
 - dependency analysis 19-29
 - best practices 19-47
 - comparing manifests 19-41
 - editing manifests 19-38
 - exporting manifests 19-42
 - file manifests 19-44
 - generating manifests 19-30
 - viewing dependencies 19-52
 - Derivative block
 - linearization 25-8
 - Description model parameter 4-113
 - Description property
 - Embedded MATLAB Function blocks 30-49
 - designing
 - custom Simulink blocks 29-18
 - designing print frames 31-8
 - DestroyFcn block callback parameter 4-59
 - diagnosing simulation errors 21-27
 - diagnostics
 - for mux/bus mixtures 12-58
 - diagonal line segments 4-20
 - diagonal lines 4-19
 - direct feedthrough blocks 2-35
 - direct-feedthrough ports 7-51
 - disabled subsystem
 - output 5-7
 - disconnecting blocks 4-23
 - discrete blocks
 - in enabled subsystem 5-9
 - in triggered systems 5-18
 - discrete sample time 3-15
 - discrete states
 - initializing 10-53
 - discretization methods 4-119
 - discretizing a Simulink model 4-115
 - dlinmod function
 - extracting linear models 25-4
 - do-while control flow diagram 4-49
 - Docking Scope Viewer 24-13
 - Document link property
 - Embedded MATLAB Function blocks 30-49
 - Dormand-Prince
 - pair 21-16
 - drop shadows 7-38
 - duplicating blocks 7-13
- ## E
- editing
 - Embedded MATLAB Function block function
 - code 30-37
 - mode 31-14
 - text in cells 31-18
 - editing look-up tables 17-28
 - editor 1-14
 - canvas 1-19
 - toolbar 1-15

- either trigger event 5-14
- Embedded MATLAB
 - comparing to M-file S-functions 29-5
 - when to disable runtime checks 30-160
- Embedded MATLAB blocks 30-66
 - and Embedded MATLAB Language 30-3
 - description 30-3 30-66
- Embedded MATLAB Editor
 - description 30-10
- Embedded MATLAB function block
 - compilation report keyboard shortcuts 30-77
- Embedded MATLAB Function block
 - controlling runtime checks 30-159
 - how to disable runtime checks 30-160
 - resolving signal objects 30-98
- Embedded MATLAB Function block fimath
 - Embedded MATLAB Function blocks 30-48
- Embedded MATLAB Function blocks
 - adding data using the Ports and Data Manager 30-49
 - adding frame-based data 30-135
 - adding function call outputs using the Ports and Data Manager 30-62
 - adding input triggers using the Ports and Data Manager 30-58
 - and embedded applications 30-6
 - and standalone executables 30-6
 - breakpoints in function 30-25
 - builtin data types 30-86
 - calling extrinsic functions 30-6
 - calling MATLAB functions 30-6 30-13
 - creating model with 30-8
 - data range checking 30-34
 - debugging 30-24
 - debugging example 30-24
 - debugging function for 30-23
 - debugging operations 30-34
 - description 30-3
 - Description property 30-49
 - diagnostic errors 30-15
 - display variable value 30-31
 - displaying variable values in MATLAB 30-32
 - Document link property 30-49
 - Embedded MATLAB Editor 30-10 30-37
 - Embedded MATLAB Function block
 - fimath 30-48
 - Embedded MATLAB runtime library of functions 30-6
 - example containing structures 30-101
 - example model with 30-8
 - example program 30-10
 - function library 30-12
 - implicitly declared variables 30-12
 - inherited data types and sizes 30-6
 - inheriting variable size 30-93
 - Lock Editor property 30-47
 - Model Explorer 30-19
 - Name property 30-45
 - names and ports 30-8
 - parameter arguments 30-97
 - Ports and Data Manager 30-42
 - Saturate on integer overflow property 30-46
 - setting properties 30-44
 - simulating function 30-24
 - Simulink input signal properties 30-47
 - sizing variables 30-93
 - sizing variables by expression 30-95
 - stepping through function 30-26
 - subfunctions 30-5 30-14
 - supported compilers for code
 - generation 30-15
 - supported compilers for simulation 30-15
 - typing variables 30-81
 - typing with other variables 30-86
 - Update method property 30-45
 - variable type by inheritance 30-84
 - variables for 30-19
 - why use them? 30-6
 - working with frame-based signals 30-134

- working with structures and bus signals 30-100
 - Embedded MATLAB Language 30-3
 - Embedded MATLAB runtime library
 - functions 30-6
 - Enable block
 - creating enabled subsystems 5-5
 - outputting enable signal 5-9
 - states when enabling 5-8
 - zero crossings 2-32
 - enabled subsystems 5-4
 - initializing output of 10-57
 - setting states 5-8
 - ending Simulink session 1-42
 - enlarging display 31-6
 - entries for information 31-16
 - enumerated data types 14-2
 - error checking
 - Embedded MATLAB Function blocks 30-15
 - error tolerance 21-17
 - simulation accuracy 23-10
 - simulation speed 23-2
 - ErrorFcn block callback parameter 4-60
 - example using print frames 31-26
 - examples
 - Clock block 25-3
 - continuous system 18-8
 - converting Celsius to Fahrenheit 18-15
 - equilibrium point determination 25-10
 - linearization 25-4
 - multirate discrete model 3-22
 - Outport block 25-2
 - return variables 25-2
 - structures in an Embedded MATLAB Function block 30-101
 - To Workspace block 25-3
 - Transfer Function block 18-9
 - working with frame-based signals in Embedded MATLAB Function blocks 30-136
 - execution context
 - defined 5-27
 - displaying 5-28
 - propagating 5-27
 - Exit MATLAB menu item 1-42
 - extrinsic functions
 - calling in Embedded MATLAB Function block functions 30-6
- F**
- falling trigger event 5-14
 - Fcn block
 - simulation speed 23-2
 - figure file 31-6
 - saving 31-22
 - file
 - .fig 31-22
 - opening 31-22
 - saving 31-22
 - file name
 - maximum length 1-10
 - filename entry 31-15
 - files
 - writing to 21-5
 - Final State check box 15-29
 - fixed in minor step 3-16
 - fixed-point data 13-4
 - properties 30-87
 - properties in Simulink model 13-18
 - fixed-step solvers
 - definition 2-23
 - Flip Name menu item 7-40
 - floating Display block 21-19
 - floating Scope block 21-19
 - font
 - annotations 4-27
 - block 7-39
 - block names 7-39
 - signal labels 10-8

- special characters 31-15
 - symbols 31-15
- Font menu item
 - changing block name font 7-39
 - changing the font of a signal label 10-8
- font size, setting for Model Explorer 19-19
- font size, setting for Simulink dialog boxes 19-19
- fonts, changing 31-20
- for control flow diagram 4-50
- For Iterator block
 - and Assignment block 4-51
 - in subsystem 4-50
 - output iteration number 4-51
 - specifying number of iterations 4-51
- frame-based signals
 - adding frame-based data to Embedded MATLAB Function blocks 30-135
 - examples of use in Embedded MATLAB Function blocks 30-136
 - using in Embedded MATLAB Function blocks 30-134
- frameedit
 - opening print frame 31-22
 - starting 31-6
- frames 31-2
- freestanding configuration sets 20-12
 - creating 20-16
- From Workspace block
 - zero crossings 2-32
- full filename entry 31-15
- full system name entry 31-15
- function call outputs, adding to Embedded MATLAB Function blocks 30-62
- functions
 - Embedded MATLAB Function block runtime library 30-12
- fundamental sample time 21-10

G

- Gain block
 - algebraic loops 2-35
- general code appearance options
 - Maximum identifier length 27-18
 - Reserved names 27-18
- Generator
 - attaching 24-24
 - performing common tasks 24-24
 - removing 24-24
- get_param command
 - checking simulation status 22-5
- Go To Library Link menu item 8-5
- Greek letters 31-15
 - using in annotations 4-33
- grouping blocks 4-37

H

- handles on selected object 4-7
- held output of enabled subsystem 5-7
- held states of enabled subsystem 5-8
- Hide Name menu item
 - hiding block names 7-40
 - hiding port labels 4-41
- Hide Port Labels menu item 4-41
- hiding block names 7-40
- hierarchy of model
 - advantage of subsystems 18-7
 - replacing virtual subsystems 2-19
- Hit Crossing block
 - notification of zero crossings 2-34
- how to disable runtime checks
 - Embedded MATLAB Function block 30-160
- hybrid systems
 - integrating 3-25

I

- If block

- connecting outputs 4-45
 - data input ports 4-45
 - data output ports 4-45
 - zero crossings
 - and Disable zero crossing detection option 2-32
 - if-else control flow diagram 4-44
 - important note about print frames 31-9
 - information
 - adding to cells 31-14
 - copying among cells 31-19
 - in print frames 31-8
 - mandatory 31-15
 - multiple entries in a cell 31-16
 - removing 31-19
 - information list box 31-14
 - inherited sample time 3-16
 - inheriting Embedded MATLAB Function block
 - variable size 30-93
 - inheriting Embedded MATLAB Function
 - variable types 30-84
 - InitFcn block callback parameter 4-59
 - InitFcn model callback parameter 4-56
 - initial conditions
 - specifying 15-29
 - Initial State check box 15-30
 - initial states
 - loading 15-30
 - initial step size
 - simulation accuracy 23-10
 - initial values
 - tuning 10-55
 - inlining S-functions using the TLC
 - and Accelerator performance 27-15
 - Inport block
 - in subsystem 4-38
 - linearization 25-5
 - supplying input to model 15-16
 - inports
 - associating
 - with bus objects 12-12
 - connecting to buses 12-51
 - root-level
 - connecting to buses 12-51
 - input triggers, adding to Embedded MATLAB Function blocks 30-58
 - inputs
 - loading from a workspace 15-16
 - mixing vector and scalar 10-28
 - scalar expansion 10-28
 - integer overflow and underflow
 - and Real-Time Workshop targets in Embedded MATLAB Function blocks 30-47
 - and simulation targets in Embedded MATLAB Function blocks 30-46
 - in Embedded MATLAB Function blocks 30-46
 - Integrator block
 - algebraic loops 2-35
 - example 18-8
 - sample time colors 3-33
 - simulation speed 23-3
 - zero crossings 2-32
 - invalid loops, avoiding 18-2
 - invalid loops, detecting 18-3
- J**
- Jacobian matrices 21-17
- K**
- keyboard actions summary 1-43
 - Keywords
 - acceleration 27-18
- L**
- labeling signals 10-7
 - labeling subsystem ports 4-41

- LastModifiedBy model parameter 4-113
 - LastModifiedDate model parameter 4-113
 - left alignment button 31-18
 - libinfo command 8-5
 - libraries. *See* block libraries
 - library block
 - definition Glossary-1
 - library blocks
 - breaking links to 8-11
 - finding 8-5
 - getting information about 8-10
 - Library Browser
 - adding libraries to 8-24
 - copying blocks from 4-6
 - library link
 - definition Glossary-1
 - library links
 - disabling 8-7
 - displaying 8-6
 - self-modifying 8-5
 - showing in Model Browser 19-28
 - status of 8-10
 - unresolved 8-12
 - line segments 4-20
 - diagonal 4-20
 - moving 4-20
 - line vertices
 - moving 4-21
 - linear models
 - extracting
 - example 25-4
 - linearization 25-4
 - lines
 - branch 4-19
 - connecting blocks 4-15
 - diagonal 4-19
 - dragging 31-12
 - moving 7-10
 - selecting 31-11
 - signals carried on 21-19
 - links
 - breaking 8-11
 - LinkStatus block parameter 8-10
 - linmod function
 - example 25-4
 - LoadFcn block callback parameter 4-60
 - loading from a workspace 15-16
 - loading initial states 15-30
 - location of block names 7-39
 - Lock Editor property
 - Embedded MATLAB Function blocks 30-47
 - logging signals 15-3
 - look-up tables, editing 17-28
 - Lookup Table Editor 17-28
 - lookup tables
 - blocks 17-6
 - components 17-5
 - data characteristics 17-18
 - data entry 17-11
 - definition 17-2
 - estimation 17-23
 - examples for Prelookup and Interpolation
 - Using Prelookup blocks 17-43
 - extrapolation 17-24
 - interpolation 17-23
 - rounding 17-25
 - selection guidelines 17-8
 - terminology 17-44
 - loops, algebraic. *See* algebraic loops
 - loops, avoiding invalid 18-2
 - loops, detecting invalid 18-3
- M**
- M-file S-functions
 - customized saturation block example 29-21
 - simulation speed 23-2
 - magnifying display 31-6
 - manifest 19-44
 - margins 31-9

- masked blocks
 - parameters
 - referencing in callbacks 4-58
 - showing in Model Browser 19-28
- masked subsystems
 - showing in Model Browser 19-28
- Math Function block
 - algebraic loops 2-35
- mathematical symbols 31-15
 - using in annotations 4-33
- MATLAB
 - in Embedded MATLAB Function
 - blocks 30-13
 - terminating 1-42
- MATLAB Fcn block
 - simulation speed 23-2
- MATLAB functions
 - calling in Embedded MATLAB Function
 - block functions 30-6
- mdl files 1-9
- Memory block
 - simulation speed 23-2
- memory issues 18-7
- menus 1-19
 - context 1-20
- MinMax block
 - zero crossings 2-32
- mixed continuous and discrete systems 3-25
- mixing
 - muxes, and buses 12-57
- model
 - editor 1-14
- Model Advisor
 - and mux/bus mixtures 12-60
- Model Browser 19-26
 - showing library links in 19-28
 - showing masked subsystems in 19-28
- model callback parameters 4-55
- model configuration preferences 19-5
- model dependencies 19-29
 - best practices 19-47
 - comparing manifests 19-41
 - editing manifests 19-38
 - exporting manifests 19-42
 - file manifests 19-44
 - generating manifests 19-30
 - viewing 19-52
- model dependency viewer 19-52
- model discretization
 - configurable subsystems 4-121
 - discretizing a model 4-115
 - overview 4-114
 - specifying the discretization method 4-119
 - starting the model discretizer 4-116
- Model Explorer
 - Apply Changes 19-12
 - Auto Apply/Ignore Dialog Changes 19-12
 - Dialog pane 19-11
 - Embedded MATLAB Function blocks 30-19
 - font size 19-19
- model file name, maximum size of 1-10
- model files
 - mdl file 1-9
- model navigation commands 4-40
- model parameters for version control 4-112
- ModelCloseFcn block callback parameter 4-60
- modeling strategies 18-7
- models
 - callback routines 4-54
 - creating 4-2
 - creating change histories for 4-111
 - creating templates 4-2
 - editing 1-4
 - navigating 4-40
 - organizing and documenting 18-7
 - printing 1-29
 - properties of 4-103
 - saving 1-8
 - selecting entire 4-8
 - tips for building 18-6

- version control properties of 4-112
- ModelVersion model parameter 4-113
- ModelVersionFormat model parameter 4-113
- ModifiedBy model parameter 4-113
- ModifiedByFormat model parameter 4-113
- ModifiedComment model parameter 4-113
- ModifiedDate model parameter 4-113
- ModifiedDateFormat model parameter 4-113
- ModifiedHistory> model parameter 4-113
- Monte Carlo analysis 22-2
- mouse actions summary 1-43
- MoveFcn block callback parameter 4-60
- multiple entries in a cell 31-16
- multirate discrete systems
 - example 3-23
- Mux blocks
 - used to create bus signals 12-58
- mux/bus mixtures
 - and Model Advisor 12-60
 - avoiding 12-63
 - diagnostics for 12-58
- muxes
 - correcting busses used as 12-62
 - mixing with buses 12-57

N

- Name property
 - Embedded MATLAB Function blocks 30-45
- NameChangeFcn block callback parameter 4-60
- names
 - blocks 7-39
 - copied blocks 7-10
- nesting
 - buses 12-7
- New menu item 4-2
- nonvirtual buses
 - compared with virtual 12-48
 - specifying 12-48
- nonvirtual subsystem 2-12

- number of pages entry 31-15
- numerical differentiation formula 21-16
- numerical integration 2-20

O

- objects
 - selecting more than one 4-7
 - selecting one 4-7
- obtaining
 - configuration reference handles 20-21
 - values using configuration references 20-25
- ode113 solver
 - advantages 21-16
 - hybrid systems 2-9
 - Memory block
 - and simulation speed 23-2
- ode15s solver
 - advantages 21-16
 - and stiff problems 23-3
 - hybrid systems 2-9
 - Memory block
 - and simulation speed 23-2
 - unstable simulation results 23-10
- ode23 solver 21-16
 - hybrid systems 2-8
- ode23s solver
 - advantages 21-16
 - simulation accuracy 23-10
- ode45 solver
 - hybrid systems 2-8
- Open menu item 1-4
- OpenFcn block callback parameter
 - purpose 4-61
- opening
 - print frame file 31-22
 - PrintFrame Editor 31-6
 - Subsystem block 4-39
 - the Bus Editor 12-14
- orientation of blocks 7-34

- Output block
 - example 25-2
 - in subsystem 4-38
 - linearization 25-5
 - outputs
 - associating
 - with bus objects 12-12
 - output
 - additional 15-34
 - between trigger events 5-16
 - disabled subsystem 5-7
 - enable signal 5-9
 - options 15-33
 - saving to workspace 15-24
 - smoother 15-33
 - specifying for simulation 15-34
 - subsystem initial output 5-5
 - trajectories
 - viewing 25-2
 - trigger signal 5-17
 - writing to file
 - when written 21-5
 - writing to workspace 15-24
 - when written 21-5
 - output ports
 - Enable block 5-9
 - Trigger block 5-17
- P**
- page number entry 31-15
 - page setup 31-9
 - panning block diagrams 1-23
 - paper orientation and type 31-9
 - PaperOrientation model parameter 1-31
 - PaperPosition model parameter 1-31
 - PaperPositionMode model parameter 1-31
 - PaperType model parameter 1-31
 - parameter arguments for Embedded MATLAB Function blocks 30-97
 - Parameter values
 - checking
 - using get_param 7-19
 - parameterized link
 - definition Glossary-1
 - parameters
 - block 7-15
 - setting values of 7-16
 - tunable
 - definition 2-9
 - ParentCloseFcn block callback parameter 4-61
 - Paste menu item 7-9
 - performance
 - comparing Acceleration to Normal Mode 23-6
 - ports
 - default port rotation 7-35
 - labeling in subsystem 4-41
 - physical port rotation 7-35
 - Ports and Data Manager
 - adding data to Embedded MATLAB Function blocks 30-49
 - adding function call outputs to Embedded MATLAB Function blocks 30-62
 - adding input triggers to Embedded MATLAB Function blocks 30-58
 - Ports and Data Manager, Embedded MATLAB Function Block 30-42
 - PostLoadFcn model callback parameter 4-56
 - PostSaveFcn block callback parameter 4-61
 - PostSaveFcn model callback parameter 4-56
 - PostScript files
 - printing to 1-37
 - PreCopyFcn block callback parameter 4-61
 - PreDeleteFcn block callback parameter 4-61
 - preferences, model configuration 19-5
 - PreLoadFcn model callback parameter 4-57
 - PreSaveFcn block callback parameter 4-62
 - PreSaveFcn model callback parameter 4-57
 - print command 1-29
 - print frames

- defined 31-2
 - designing 31-8
 - important note about 31-9
 - process for creating 31-7
 - size of 31-12
 - Print menu item 1-29
 - PrintFrame Editor
 - closing 31-7
 - interface for 31-6
 - starting 31-6
 - printing to PostScript file 1-37
 - printing with print frames
 - block diagrams 31-23
 - multiple use 31-8
 - Priority block parameter 7-52
 - process for creating print frames 31-7
 - produce additional output option 15-34
 - produce specified output only option 15-34
 - Product block
 - algebraic loops 2-35
 - profiler
 - enabling 27-31
 - Profiler
 - how it works 27-29
 - properties
 - setting for Embedded MATLAB Function blocks 30-44
 - properties of Scope Viewer 24-13
 - purely discrete systems 3-22
- Q**
- Quit MATLAB menu item 1-42
- R**
- Random Number block
 - simulation speed 23-3
 - Rapid Accelerator
 - description 27-2
 - determining why it rebuilds 27-7
 - how to run 27-20
 - Simulink blocks not supported 27-14 to 27-15
 - visualization 27-16
 - Rapid Accelerator mode
 - how to run 27-20
 - keywords 27-18
 - rate transitions 3-30
 - reading
 - configuration sets from an M-file 20-17
 - Redo menu item 1-22
 - reference block
 - definition Glossary-1
 - reference blocks
 - about 8-3
 - creating 8-3
 - modifying 8-4
 - updating 8-4
 - referencing
 - configuration sets 20-12
 - refine factor
 - smoothing output 15-33
 - Relational Operator block
 - zero crossings 2-32
 - relative tolerance
 - definition 21-17
 - simulation accuracy 23-10
 - Relay block
 - zero crossings 2-32
 - removing information 31-19
 - Reserved
 - accelerator mode keywords 27-18
 - reset
 - output of enabled subsystem 5-7
 - states of enabled subsystem 5-8
 - resizing blocks 7-37
 - resizing rows and cells 31-12
 - resolving

- signal objects for Embedded MATLAB
 - Function blocks 30-98
 - return variables
 - example 25-2
 - reversing direction of signal flow 18-8
 - right alignment button 31-18
 - rising trigger event 5-14
 - root-level inports
 - connecting to buses 12-51
 - Rosenbrock formula 21-16
 - Rotate Block>Clockwise menu item 7-34
 - Rotate Block>Counter-clockwise menu item 7-34
 - rotating a block 7-34
 - rows
 - adding 31-11
 - deleting 31-11
 - resizing 31-12
 - selecting 31-11
 - splitting 31-12
 - Runge-Kutta (2,3) pair 21-16
 - Runge-Kutta (4,5) formula 21-16
- S**
- sample time
 - backward propagating 3-33
 - colors 3-33
 - fundamental 21-10
 - simulation speed 23-3
 - Sample Time Colors menu item
 - updating coloring 4-13
 - Sample Time Legend 3-10
 - sample time propagation 3-31
 - saturate
 - on integer overflow in Embedded MATLAB
 - Function blocks 30-46
 - Saturate on integer overflow property
 - Embedded MATLAB Function blocks 30-46
 - saturation block
 - zero crossings
 - how used 2-33
 - Saturation block
 - zero crossings 2-33
 - Save As menu item 1-9
 - Save menu item 1-9
 - Save options area 15-24
 - save_system command
 - breaking links 8-12
 - saving 31-22
 - scalar expansion 10-28
 - Scope block
 - example of a continuous system 18-9
 - Scope blocks and viewers
 - differences between 24-3
 - Scope Viewer 24-1
 - adding multiple axes 24-20
 - adding multiple signals to 24-19
 - attaching 24-18
 - axes 24-13
 - context menu 24-17
 - data markers 24-14
 - displaying signals 24-9
 - how to attach 24-6
 - how to display 24-7
 - legends 24-19
 - limiting data 24-15
 - line styles 24-9
 - performance 24-10
 - performing common tasks 24-18
 - properties dialog 24-13
 - refresh 24-16
 - saving axes settings
 - gui 24-13
 - scroll 24-14
 - signal logging 24-15
 - simulation speed 23-3
 - things to know 24-9
 - toolbar 24-12
 - trace colors 24-9
 - zooming 24-19

- Scopes
 - using with Rapid Accelerator 27-16
- Select All menu item 4-8
- selecting cells, rows, and lines 31-11
- Set Font dialog box 7-39
- set_param command
 - breaking link 8-11
 - controlling model execution 27-24
 - running a simulation 21-3 22-5
 - setting block parameters within M-file
 - S-functions 29-33
 - setting simulation mode 27-24
- setting breakpoints 26-27
- shadowed files 18-4
- Shampine, L. F. 21-17
- Shape preservation
 - improving solver accuracy 2-24
- Show Name menu item 7-40
- show output port
 - Enable block 5-9
 - Trigger block 5-17
- showing block names 7-40
- Sign block
 - zero crossings 2-33
- Signal Builder
 - snap grid 10-80
- Signal Builder block
 - zero crossings 2-33
- Signal Builder dialog box 10-71
- Signal Builder time range
 - about 10-82
 - changing 10-82
- signal groups 10-69
 - activating 10-84
 - copying and pasting 10-74
 - creating a custom waveform in 10-74
 - creating a set of 10-69
 - creating and deleting 10-73
 - creating signals in 10-73
 - deleting 10-75
 - discrete 10-87
 - editing 10-71
 - exporting to workspace 10-83
 - final values 10-85
 - hiding waveforms 10-73
 - moving 10-73
 - renaming 10-73
 - renaming signals in 10-84
 - running all 10-84
 - simulating with 10-84
 - specifying final values for 10-85
 - specifying sample time of 10-87
 - time range of 10-82
- signal labels
 - changing font 10-8
 - copying 10-7
 - creating 10-7
 - deleting 10-8
 - editing 10-7
 - moving 10-7
 - using to document models 18-7
- signal logging, enabling 15-4
- signal objects
 - resolving for Embedded MATLAB Function blocks 30-98
 - using to initialize signals 10-54
- signal propagation 10-13
- signals
 - composite 12-2
 - initializing 10-53
 - labeling 10-7
 - labels 10-7
 - names 10-3
 - reversing direction of flow 18-8
 - virtual 10-12
- signals, creating 10-3
- signals, logging 15-3
- sim command
 - comparing performance 23-6
 - simulating an accelerated model 27-25

- syntax 22-3
- simulation
 - accuracy 23-10
 - checking status of 22-5
 - displaying information about
 - algebraic loops 26-35
 - block execution order 26-38
 - block I/O 26-33
 - debug settings 26-41
 - integration 26-36
 - nonvirtual blocks 26-40
 - nonvirtual systems 26-39
 - system states 26-36
 - zero crossings 26-40
 - Embedded MATLAB Function block
 - function 30-24
 - execution phase 2-20
 - parameters
 - specifying 21-27
 - running incrementally 26-19
 - running nonstop 26-24
 - speed 23-2
 - status bar 1-20
 - stepping by breakpoints 26-27
 - stepping by time steps 26-22
 - unstable results 23-10
- Simulation Diagnostics Viewer 21-27
- simulation errors
 - diagnosing 21-27
- Simulation Options dialog box 10-85
- simulation time
 - compared to clock time 21-8
 - writing to workspace 15-24
- Simulink
 - custom blocks, creating 29-2
 - ending session 1-42
 - icon 1-2
 - menus 1-19
 - starting 1-2
 - terminating 1-42
 - Simulink block library. *See* block libraries
 - `simulink` command
 - starting Simulink 1-2
 - Simulink dialog boxes
 - font size 19-19
 - Simulink input signal properties
 - Embedded MATLAB Function blocks 30-47
 - Simulink Profiler
 - purpose 27-29
 - Simulink, printing with print frames 31-2
 - `Simulink.BlockDiagram.getChecksum` command
 - determining why the Accelerators rebuilds with 27-8
 - size of
 - cells 31-12
 - print frame 31-9
 - print frame page setup 31-12
 - rows 31-12
 - text 31-20
 - size of block
 - changing 7-37
 - sizing Embedded MATLAB Function block
 - variables by expression 30-95
 - sizing Embedded MATLAB Function block
 - variables by inheritance 30-93
 - sizing Embedded MATLAB Function
 - variables 30-93
 - `sldebug` command
 - starting the Simulink debugger 26-14
 - smart guides 7-12
 - snap grid, Signal Builder's 10-80
 - solvers
 - fixed-step
 - definition 2-23
 - `ode113`
 - advantages 21-16
 - and simulation speed 23-2
 - `ode15s`
 - advantages 21-16
 - and simulation speed 23-2

- and stiff problems 23-3
- simulation accuracy 23-10
- ode23 21-16
- ode23s
 - advantages 21-16
 - simulation accuracy 23-10
- sorted order
 - displaying 7-46
- Source Control menu item 4-99
- special characters 31-15
- specifying
 - nonvirtual buses 12-48
- speed of simulation 23-2
- splitting rows and cells 31-12
- stairs function 3-24
- Start menu item 18-16
- start time 21-8
- StartFcn block callback parameter 4-62
- StartFcn model callback parameter 4-57
- starting
 - PrintFrame Editor 31-6
- starting Simulink 1-2
- starting the model discretizer 4-116
- State-Space block
 - algebraic loops 2-35
- states
 - between trigger events 5-16
 - loading initial 15-30
 - when enabling 5-8
 - writing to workspace 15-24
- states, discrete
 - initializing 10-53
- static information in cells 31-8
- status
 - checking simulation 22-5
- status bar 1-20
- Step block
 - zero crossings 2-33
- step size
 - simulation speed 23-2
- stiff problems 21-17
- stiff systems
 - simulation speed 23-3
- stop time 21-8
- StopFcn block callback parameter 4-62
- StopFcn model callback parameter 4-57
- structures
 - using in Embedded MATLAB Function blocks 30-100
- style of text 31-20
- subfunctions
 - in Embedded MATLAB Function block functions 30-5
 - in Embedded MATLAB Function blocks 30-14
- sublibrary 8-15
 - definition Glossary-1
- subscript 31-15
- subsystem
 - atomic 2-13
 - conditionally executed 2-12
 - initial output 5-5
 - nonvirtual 2-12
 - virtual 2-12
- Subsystem block
 - adding to create subsystem 4-38
 - opening 4-39
- Subsystem Examples block library 18-2
- subsystem ports
 - labeling 4-41
- subsystem sample time
 - sample time 3-21
- subsystems
 - controlling access to 4-42
 - creating 4-37
 - displaying parent of 4-40
 - labeling ports 4-41
 - model hierarchy 18-7
 - opening 4-40
 - triggered and enabled 5-19

- underlying blocks 4-39
- undoing creation of 4-40
- Sum block
 - algebraic loops 2-35
- summary of mouse and keyboard actions 1-43
- superscript 31-15
- Switch block
 - zero crossings 2-33
- Switch Case block
 - zero crossings
 - and Disable zero-crossing detection option 2-33
- switch control flow diagram 4-46
- SwitchCase block
 - adding cases 4-46
 - connecting to Action subsystem 4-46
 - data input 4-46
- symbols 31-15
- system name entry 31-15

T

- terminating MATLAB 1-42
- terminating Simulink 1-42
- terminating Simulink session 1-42
- test point icons 10-62 15-5
- test points 10-61
- TeX commands
 - using in annotations 4-33
- TeX sequences 31-15
- text
 - adding to cells 31-15
 - editing 31-18
 - editing mode 31-14
 - size of 31-20
 - special characters 31-15
 - style of 31-20
- tic command
 - comparing performance 23-6
- time entry 31-15
- time interval
 - simulation speed 23-2
- time range
 - of a Signal Builder block 10-82
- tips for building models 18-6
- To Workspace block
 - example 25-3
- toc command
 - comparing performance 23-6
- toolbar
 - editor 1-15
- total pages entry 31-15
- Trace colors
 - Scope Viewer 24-9
- Transfer Fcn block
 - algebraic loops 2-35
 - example 18-9
- Transport Delay block
 - linearization 25-8
- Trigger and Enabled Subsystem
 - zero crossings 2-33
- Trigger block
 - creating triggered subsystem 5-15
 - outputting trigger signal 5-17
 - showing output port 5-17
 - zero crossings 2-33
- triggered and enabled subsystems 5-19
- triggered sample time 3-19
- triggered subsystems 5-14
- triggers
 - control signal
 - outputting 5-17
 - either 5-14
 - events 5-14
 - falling 5-14
 - input 5-14
 - rising 5-14
 - type parameter 5-15
- tunable parameters
 - definition 2-9

- tutorial
 - creating a customized saturation block 29-18
 - typing Embedded MATLAB Function block
 - variables with other variables 30-86
 - typing Embedded MATLAB Function
 - variables 30-81
- U**
- Undo menu item 1-22
 - UndoDeleteFcn block callback parameter 4-62
 - undoing commands 1-22
 - units for margins 31-9
 - unstable simulation results 23-10
 - Update Diagram menu item
 - fixing bad link 8-13
 - out-of-date reference block 8-4
 - recoloring model 4-13
 - Update method property
 - Embedded MATLAB Function blocks 30-45
 - updating a block diagram 1-27
 - updating a diagram programmatically 22-5
 - user
 - specifying current 4-101
 - user data 13-57
 - UserData 13-57
 - UserDataPersistent 13-57
 - using
 - bus objects 12-10
- V**
- variable information
 - adding to cells 31-15
 - defined 31-8
 - format for 31-16
 - variable sample time 3-18
 - variables
 - creating for Embedded MATLAB Function blocks 30-19
 - vector length
 - checking 2-19
 - vectors
 - bus signals used as 12-59
 - version control model parameters 4-112
 - vertices
 - moving 4-21
 - Viewers
 - using with Rapid Accelerator 27-16
 - Viewers and generators
 - when to use 24-4
 - viewing output trajectories 25-2
 - viewing sample time information
 - Sample Time Display 3-9
 - virtual blocks 7-2
 - virtual buses
 - compared with nonvirtual 12-48
 - virtual signals 10-12
 - virtual subsystem 2-12
 - visualization
 - Rapid Accelerator limitations 27-16
 - Visualizing
 - simulation results 24-1
- W**
- when to disable runtime checks
 - Embedded MATLAB 30-160
 - while control flow diagram 4-47
 - While Iterator block
 - changing to do-while 4-49
 - condition input 4-49
 - in subsystem 4-48
 - initial condition input 4-49
 - iterator number output 4-49
 - window reuse 4-40
 - workspace
 - loading from 15-16
 - saving to 15-24
 - writing to

simulation terminated or
suspended 21-5

Z

zero

zero-crossing threshold 2-30

zero crossing detection 2-24

zero crossings

disabled by non-double data types 13-25

saturation block 2-33

zero-crossing

adaptive 2-29

algorithms 2-29

blocks that register 2-31

demonstrating effects 2-25

missing events 2-27

nonadaptive 2-29

preventing excessive 2-27

threshold 2-30

zero-crossing detection 2-24

zero-crossing slope method 5-5

Zero-Pole block

algebraic loops 2-35

zooming 31-6

zooming block diagrams 1-22