

# SIMULINK<sup>®</sup>

Dynamic System Simulation for MATLAB<sup>®</sup>

Modeling

Simulation

Implementation



Using Simulink

*Version 3*

## How to Contact The MathWorks:



508-647-7000 Phone



508-647-7001 Fax



The MathWorks, Inc. Mail  
24 Prime Park Way  
Natick, MA 01760-1500



<http://www.mathworks.com> Web  
<ftp.mathworks.com> Anonymous FTP server  
<comp.soft-sys.matlab> Newsgroup



[support@mathworks.com](mailto:support@mathworks.com) Technical support  
[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[subscribe@mathworks.com](mailto:subscribe@mathworks.com) Subscribing user registration  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information

### *Using Simulink*

© COPYRIGHT 1990 - 1999 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.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks and the Target Language Compiler is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: 1990 First printing  
December 1996 Revised for Simulink 2  
May 1997 Revised for Simulink 2.1 (online version)  
January 1998 Revised for Simulink 2.2 (online version)  
January 1999 Revised for Simulink 3 (Release 11)

## Getting Started

### 1

<b>To the Reader</b> .....	1-2
What Is Simulink? .....	1-2
How to Use This Manual .....	1-3
<b>Application Toolboxes</b> .....	1-5
<b>The Simulink Real-Time Workshop</b> .....	1-10
Key Features .....	1-10
<b>The Real-Time Workshop Ada Extension</b> .....	1-12
Key Features .....	1-12
<b>Blocksets</b> .....	1-14
The DSP Blockset .....	1-14
The Fixed-Point Blockset .....	1-14
The Nonlinear Control Design Blockset .....	1-16
The Power System Blockset .....	1-16

## Quick Start

### 2

<b>Running a Demo Model</b> .....	2-2
Description of the Demo .....	2-3
Some Things to Try .....	2-4
What This Demo Illustrates .....	2-5
Other Useful Demos .....	2-5

<b>Building a Simple Model</b> .....	<b>2-6</b>
--------------------------------------	------------

## **Creating a Model**

# **3**

<b>Starting Simulink</b> .....	<b>3-2</b>
Creating a New Model .....	<b>3-3</b>
Editing an Existing Model .....	<b>3-3</b>
Entering Simulink Commands .....	<b>3-3</b>
Simulink Windows .....	<b>3-5</b>
Zooming Block Diagrams .....	<b>3-6</b>
<b>Selecting Objects</b> .....	<b>3-7</b>
Selecting One Object .....	<b>3-7</b>
Selecting More than One Object .....	<b>3-7</b>
<b>Blocks</b> .....	<b>3-9</b>
Block Data Tips .....	<b>3-9</b>
Virtual Blocks .....	<b>3-9</b>
Copying and Moving Blocks from One Window to Another ..	<b>3-10</b>
Moving Blocks in a Model .....	<b>3-12</b>
Duplicating Blocks in a Model .....	<b>3-12</b>
Specifying Block Parameters .....	<b>3-12</b>
Block Properties Dialog Box .....	<b>3-13</b>
Deleting Blocks .....	<b>3-14</b>
Changing the Orientation of Blocks .....	<b>3-15</b>
Resizing Blocks .....	<b>3-15</b>
Manipulating Block Names .....	<b>3-16</b>
Displaying Parameters Beneath a Block's Icon .....	<b>3-17</b>
Disconnecting Blocks .....	<b>3-18</b>
Vector Input and Output .....	<b>3-18</b>
Scalar Expansion of Inputs and Parameters .....	<b>3-18</b>
Assigning Block Priorities .....	<b>3-19</b>
Using Drop Shadows .....	<b>3-20</b>
<b>Libraries</b> .....	<b>3-21</b>
Terminology .....	<b>3-21</b>

Creating a Library .....	3-21
Modifying a Library .....	3-22
Copying a Library Block into a Model .....	3-22
Updating a Linked Block .....	3-23
Breaking a Link to a Library Block .....	3-23
Finding the Library Block for a Reference Block .....	3-24
Getting Information About Library Blocks .....	3-24
Browsing Block Libraries .....	3-25
<b>Lines .....</b>	<b>3-27</b>
Drawing a Line Between Blocks .....	3-27
Drawing a Branch Line .....	3-28
Drawing a Line Segment .....	3-28
Displaying Line Widths .....	3-31
Inserting Blocks in a Line .....	3-31
Signal Labels .....	3-32
Setting Signal Properties .....	3-34
Signal Properties Dialog Box .....	3-35
<b>Annotations .....</b>	<b>3-37</b>
<b>Working with Data Types .....</b>	<b>3-38</b>
Data Types Supported by Simulink .....	3-38
Block Support for Data and Numeric Signal Types .....	3-39
Specifying Block Parameter Data Types .....	3-43
Creating Signals of a Specific Data Type .....	3-43
Displaying Port Data Types .....	3-43
Data Type Propagation .....	3-43
Data Typing Rules .....	3-44
Enabling Strict Boolean Type Checking .....	3-45
Typecasting Signals .....	3-45
Typecasting Parameters .....	3-45
<b>Working with Complex Signals .....</b>	<b>3-47</b>
<b>Summary of Mouse and Keyboard Actions .....</b>	<b>3-48</b>
<b>Creating Subsystems .....</b>	<b>3-51</b>
Creating a Subsystem by Adding the Subsystem Block .....	3-51

Creating a Subsystem by Grouping Existing Blocks .....	3-52
Labeling Subsystem Ports .....	3-53
Using Callback Routines .....	3-53
<b>Tips for Building Models .....</b>	<b>3-57</b>
<b>Modeling Equations .....</b>	<b>3-58</b>
Converting Celsius to Fahrenheit .....	3-58
Modeling a Simple Continuous System .....	3-59
<b>Saving a Model .....</b>	<b>3-61</b>
<b>Printing a Block Diagram .....</b>	<b>3-62</b>
Print Dialog Box .....	3-62
Print Command .....	3-63
Specifying Paper Size and Orientation .....	3-64
Positioning and Sizing a Diagram .....	3-64
<b>The Model Browser .....</b>	<b>3-66</b>
Using the Model Browser on Windows .....	3-66
Using the Model Browser on UNIX .....	3-67
<b>Tracking Model Versions .....</b>	<b>3-70</b>
Specifying the Current User .....	3-70
Model Properties Dialog .....	3-72
Creating a Model Change History .....	3-76
Version Control Properties .....	3-77
<b>Ending a Simulink Session .....</b>	<b>3-79</b>

## Running a Simulation

# 4

<b>Introduction .....</b>	<b>4-2</b>
Using Menu Commands .....	4-2
Running a Simulation from the Command Line .....	4-3

<b>Running a Simulation Using Menu Commands</b> .....	4-4
Setting Simulation Parameters and Choosing the Solver .....	4-4
Applying the Simulation Parameters .....	4-4
Starting the Simulation .....	4-4
Simulation Diagnostics Dialog Box .....	4-6
<b>The Simulation Parameters Dialog Box</b> .....	4-8
The Solver Page .....	4-8
The Workspace I/O Page .....	4-17
The Diagnostics Page .....	4-24
<b>Improving Simulation Performance and Accuracy</b> .....	4-27
Speeding Up the Simulation .....	4-27
Improving Simulation Accuracy .....	4-28
<b>Running a Simulation from the Command Line</b> .....	4-29
Using the sim Command .....	4-29
Using the set_param Command .....	4-29
sim .....	4-30
simset .....	4-32
simget .....	4-36

## Analyzing Simulation Results

# 5

<b>Viewing Output Trajectories</b> .....	5-2
Using the Scope Block .....	5-2
Using Return Variables .....	5-2
Using the To Workspace Block .....	5-3
<b>Linearization</b> .....	5-4
<b>Equilibrium Point Determination</b> .....	5-7
linfun .....	5-9
trim .....	5-13

<b>Introduction</b> .....	<b>6-2</b>
<b>A Sample Masked Subsystem</b> .....	<b>6-3</b>
Creating Mask Dialog Box Prompts .....	<b>6-4</b>
Creating the Block Description and Help Text .....	<b>6-6</b>
Creating the Block Icon .....	<b>6-6</b>
Summary .....	<b>6-8</b>
<b>The Mask Editor: An Overview</b> .....	<b>6-9</b>
<b>The Initialization Page</b> .....	<b>6-10</b>
Prompts and Associated Variables .....	<b>6-10</b>
Control Types .....	<b>6-12</b>
Default Values for Masked Block Parameters .....	<b>6-14</b>
Tunable Parameters .....	<b>6-14</b>
Initialization Commands .....	<b>6-15</b>
<b>The Icon Page</b> .....	<b>6-18</b>
Displaying Text on the Block Icon .....	<b>6-18</b>
Displaying Graphics on the Block Icon .....	<b>6-20</b>
Displaying Images on Masks .....	<b>6-21</b>
Displaying a Transfer Function on the Block Icon .....	<b>6-22</b>
Controlling Icon Properties .....	<b>6-23</b>
<b>The Documentation Page</b> .....	<b>6-26</b>
The Mask Type Field .....	<b>6-26</b>
The Block Description Field .....	<b>6-26</b>
The Mask Help Text Field .....	<b>6-27</b>
<b>Creating Dynamic Dialogs for Masked Blocks</b> .....	<b>6-28</b>
Setting Masked Block Dialog Parameters .....	<b>6-28</b>
Predefined Masked Dialog Parameters .....	<b>6-29</b>



## Conditionally Executed Subsystems

# 7

<b>Introduction</b> .....	7-2
<b>Enabled Subsystems</b> .....	7-3
Creating an Enabled Subsystem .....	7-3
Blocks an Enabled Subsystem Can Contain .....	7-5
<b>Triggered Subsystems</b> .....	7-8
Creating a Triggered Subsystem .....	7-9
Function-Call Subsystems .....	7-10
Blocks That a Triggered Subsystem Can Contain .....	7-10
<b>Triggered and Enabled Subsystems</b> .....	7-11
Creating a Triggered and Enabled Subsystem .....	7-11
A Sample Triggered and Enabled Subsystem .....	7-12
Creating Alternately Executing Subsystems .....	7-12

## Block Reference

# 8

<b>What Each Block Reference Page Contains</b> .....	8-2
<b>Simulink Block Libraries</b> .....	8-3
Abs .....	8-11
Algebraic Constraint .....	8-12
Backlash .....	8-14
Band-Limited White Noise .....	8-18
Bus Selector .....	8-20
Chirp Signal .....	8-22
Clock .....	8-24
Combinatorial Logic .....	8-25
Complex to Magnitude-Angle .....	8-28
Complex to Real-Imag .....	8-29
Configurable Subsystem .....	8-30
Constant .....	8-34

Coulomb and Viscous Friction . . . . .	8-35
Data Store Memory . . . . .	8-36
Data Store Read . . . . .	8-38
Data Store Write . . . . .	8-39
Data Type Conversion . . . . .	8-41
Dead Zone . . . . .	8-43
Demux . . . . .	8-45
Derivative . . . . .	8-49
Digital Clock . . . . .	8-51
Discrete Filter . . . . .	8-52
Discrete Pulse Generator . . . . .	8-54
Discrete State-Space . . . . .	8-56
Discrete-Time Integrator . . . . .	8-58
Discrete Transfer Fcn . . . . .	8-65
Discrete Zero-Pole . . . . .	8-67
Display . . . . .	8-69
Dot Product . . . . .	8-72
Enable . . . . .	8-74
Fcn . . . . .	8-76
First-Order Hold . . . . .	8-78
From . . . . .	8-80
From File . . . . .	8-82
From Workspace . . . . .	8-85
Function-Call Generator . . . . .	8-88
Gain . . . . .	8-89
Goto . . . . .	8-91
Goto Tag Visibility . . . . .	8-94
Ground . . . . .	8-95
Hit Crossing . . . . .	8-96
IC . . . . .	8-98
Inport . . . . .	8-99
Integrator . . . . .	8-103
Logical Operator . . . . .	8-108
Look-Up Table . . . . .	8-110
Look-Up Table (2-D) . . . . .	8-113
Magnitude-Angle to Complex . . . . .	8-116
Manual Switch . . . . .	8-118
Math Function . . . . .	8-119
MATLAB Fcn . . . . .	8-121
Matrix Gain . . . . .	8-123

Memory .....	8-124
Merge .....	8-126
MinMax .....	8-129
Model Info .....	8-131
Multiport Switch .....	8-134
Mux .....	8-136
Outport .....	8-139
Product .....	8-143
Probe .....	8-145
Pulse Generator .....	8-146
Quantizer .....	8-148
Ramp .....	8-149
Random Number .....	8-150
Rate Limiter .....	8-152
Real-Imag to Complex .....	8-154
Relational Operator .....	8-156
Relay .....	8-158
Repeating Sequence .....	8-160
Rounding Function .....	8-161
Saturation .....	8-162
Scope .....	8-163
Selector .....	8-173
S-Function .....	8-175
Sign .....	8-177
Signal Generator .....	8-178
Sine Wave .....	8-180
Slider Gain .....	8-183
State-Space .....	8-185
Step .....	8-187
Stop Simulation .....	8-189
Subsystem .....	8-190
Sum .....	8-191
Switch .....	8-194
Terminator .....	8-196
To File .....	8-197
To Workspace .....	8-199
Transfer Fcn .....	8-203
Transport Delay .....	8-206
Trigger .....	8-208
Trigonometric Function .....	8-210

Uniform Random Number .....	8-212
Unit Delay .....	8-214
Variable Transport Delay .....	8-216
Width .....	8-218
XY Graph .....	8-219
Zero-Order Hold .....	8-221
Zero-Pole .....	8-222

## Additional Topics

# 9

<b>How Simulink Works</b> .....	<b>9-2</b>
Zero Crossings .....	9-3
Algebraic Loops .....	9-7
Invariant Constants .....	9-11
<b>Discrete-Time Systems</b> .....	<b>9-13</b>
Discrete Blocks .....	9-13
Sample Time .....	9-13
Purely Discrete Systems .....	9-13
Multirate Systems .....	9-14
Sample Time Colors .....	9-15
Mixed Continuous and Discrete Systems .....	9-17

## Model Construction Commands

# 10

<b>Introduction</b> .....	<b>10-2</b>
How to Specify Parameters for the Commands .....	10-3
How to Specify a Path for a Simulink Object .....	10-3
add_block .....	10-4
add_line .....	10-5
bdclose .....	10-6
bdroot .....	10-7

close_system .....	10-8
delete_block .....	10-10
delete_line .....	10-11
find_system .....	10-12
gcb .....	10-14
gcbh .....	10-15
gcs .....	10-16
get_param .....	10-17
new_system .....	10-19
open_system .....	10-20
replace_block .....	10-21
save_system .....	10-23
set_param .....	10-24
simulink .....	10-26

## Simulink Debugger

# 11

<b>Introduction .....</b>	<b>11-2</b>
<b>Using the Debugger .....</b>	<b>11-3</b>
Starting the Debugger .....	11-3
Getting Help .....	11-4
Entering Commands .....	11-4
About Block Indexes .....	11-4
Accessing the MATLAB Workspace .....	11-4
<b>Running a Simulation Incrementally .....</b>	<b>11-6</b>
Stepping by Blocks .....	11-6
Stepping by Time Steps .....	11-7
Stepping by Breakpoints .....	11-8
Running a Simulation Nonstop .....	11-8
<b>Setting Breakpoints .....</b>	<b>11-9</b>
Breaking at Blocks .....	11-9
Breaking at Time Steps .....	11-11
Breaking on Nonfinite Values .....	11-11

Breaking on Step-Size Limiting Steps .....	11-12
Breaking at Zero-Crossings .....	11-12
<b>Displaying Information About the Simulation .....</b>	<b>11-13</b>
Displaying Block I/O .....	11-13
Displaying Algebraic Loop Information .....	11-14
Displaying System States .....	11-15
Displaying Integration Information .....	11-15
<b>Displaying Information About the Model .....</b>	<b>11-17</b>
Displaying a Model's Block Execution Order .....	11-17
Displaying a Block .....	11-17
Displaying a Model's Nonvirtual Systems .....	11-18
Displaying a Model's Nonvirtual Blocks .....	11-18
Displaying Blocks with Potential Zero-Crossings .....	11-20
Displaying Algebraic Loops .....	11-20
Displaying Debug Settings .....	11-21
<b>Debugger Command Reference .....</b>	<b>11-22</b>
ashow .....	11-24
atrace .....	11-25
bafter .....	11-26
break .....	11-27
bshow .....	11-28
clear .....	11-29
continue .....	11-30
disp .....	11-31
help .....	11-32
ishow .....	11-33
minor .....	11-34
nanbreak .....	11-35
next .....	11-36
probe .....	11-37
quit .....	11-38
run .....	11-39
slist .....	11-40
states .....	11-41
systems .....	11-42
status .....	11-43
step .....	11-44

stop .....	11-45
tbreak .....	11-46
trace .....	11-47
undisp .....	11-48
untrace .....	11-49
xbreak .....	11-50
zcbreak .....	11-51
zclist .....	11-52

## Model and Block Parameters

---

### A

<b>Introduction</b> .....	<b>A-2</b>
<b>Model Parameters</b> .....	<b>A-3</b>
<b>Common Block Parameters</b> .....	<b>A-7</b>
<b>Block-Specific Parameters</b> .....	<b>A-10</b>
<b>Mask Parameters</b> .....	<b>A-24</b>

## Model File Format

---

### B

<b>Model File Contents</b> .....	<b>B-2</b>
Model Section .....	<b>B-3</b>
BlockDefaults Section .....	<b>B-3</b>
AnnotationDefaults Section .....	<b>B-3</b>
System Section .....	<b>B-3</b>
<b>A Sample Model File</b> .....	<b>B-4</b>





# Getting Started

---

<b>To the Reader</b> . . . . .	1-2
What Is Simulink? . . . . .	1-2
How to Use This Manual . . . . .	1-3
<b>Application Toolboxes</b> . . . . .	1-5
<b>The Simulink Real-Time Workshop</b> . . . . .	1-10
Key Features . . . . .	1-10
<b>The Real-Time Workshop Ada Extension</b> . . . . .	1-12
Key Features . . . . .	1-12
<b>Blocksets</b> . . . . .	1-14
The DSP Blockset . . . . .	1-14
The Fixed-Point Blockset . . . . .	1-14
The Nonlinear Control Design Blockset . . . . .	1-16
The Power System Blockset . . . . .	1-16

## To the Reader

Welcome to Simulink®! In the last few years, Simulink has become the most widely used software package in academia and industry for modeling and simulating dynamical systems.

Simulink encourages you to try things out. You can easily build models from scratch, or take an existing model and add to it. Simulations are interactive, so you can change parameters “on the fly” and immediately see what happens. You have instant access to all of the analysis tools in MATLAB®, so you can take the results and analyze and visualize them. We hope that you will get a sense of the *fun* of modeling and simulation, through an environment that encourages you to pose a question, model it, and see what happens.

With Simulink, you can move beyond idealized linear models to explore more realistic nonlinear models, factoring in friction, air resistance, gear slippage, hard stops, and the other things that describe real-world phenomena. It turns your computer into a lab for modeling and analyzing systems that simply wouldn't be possible or practical otherwise, whether the behavior of an automotive clutch system, the flutter of an airplane wing, the dynamics of a predator-prey model, or the effect of the monetary supply on the economy.

Simulink is also practical. With thousands of engineers around the world using it to model and solve real problems, knowledge of this tool will serve you well throughout your professional career.

We hope you enjoy exploring the software.

## What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamical systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block

---

library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks. For information on creating your own blocks, see the separate *Writing S-Functions* guide.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

## **How to Use This Manual**

Because Simulink is graphical and interactive, we encourage you to jump right in and try it.

For a useful introduction that will help you start using Simulink quickly, take a look at "Running a Demo Model" in Chapter 2. Browse around the model, double-click on blocks that look interesting, and you will quickly get a sense of how Simulink works. If you want a quick lesson in building a model, see "Building a Simple Model" in Chapter 2.

Chapter 3 describes in detail how to build and edit a model. It also discusses how to save and print a model and provides some useful tips.

Chapter 4 describes how Simulink performs a simulation. It covers simulation parameters and the integration solvers used for simulation, including some of

the strengths and weaknesses of each solver that should help you choose the appropriate solver for your problem. It also discusses multirate and hybrid systems.

Chapter 5 discusses Simulink and MATLAB features useful for viewing and analyzing simulation results.

Chapter 6 discusses methods for creating your own blocks and using masks to customize their appearance and use.

Chapter 7 describes subsystems whose execution depends on triggering signals.

Chapter 8 provides reference information for all Simulink blocks.

Chapter 9 provides information about how Simulink works, including information about zero crossings, algebraic loops, and discrete and hybrid systems.

Chapter 10 provides reference information for commands you can use to create and modify a model from the MATLAB command window or from an M-file.

Chapter 11 explains how to use the Simulink debugger to debug Simulink models. It also documents debugger commands.

Appendix A lists model and block parameters. This information is useful with the `get_param` and `set_param` commands, described in Chapter 10.

Appendix B describes the format of the file that stores model information.

Although we have tried to provide the most complete and up-to-date information in this manual, some information may have changed after it was completed. Please check the *Known Software and Documentation Problems* delivered with your Simulink system, for the latest release notes.

## Application Toolboxes

One of the key features of Simulink is that it is built on top of MATLAB. As a result, Simulink users have direct access to the wide range of MATLAB-based tools for generating, analyzing, and optimizing systems implemented in Simulink. These tools include MATLAB Application Toolboxes, specialized collections of M-files for working on particular classes of problems.

Toolboxes are more than just collections of useful functions; they represent the efforts of some of the world's top researchers in fields such as controls, signal processing, and system identification. MATLAB Application Toolboxes therefore let you “stand on the shoulders” of world class scientists.

All toolboxes are built using MATLAB. This has some very important implications for you:

- Every toolbox builds on the robust numerics, rock-solid accuracy, and years of experience in MATLAB.
- You get seamless and immediate integration with Simulink and any other toolboxes you may own.
- Because all toolboxes are written in MATLAB code, you can take advantage of MATLAB's open-system approach. You can inspect M-files, add to them, or use them for templates when creating your own functions.
- Every toolbox is available on any computer platform that runs MATLAB.

Here is a list of professional toolboxes currently available from The MathWorks. This list is by no means static— more are being created every year.

**The Communications Toolbox.** The Communications Toolbox provides an integrated set of tools for accelerating the design, analysis, and simulation of modern communications systems. It combines MATLAB's high-level language with the ease of use of Simulink's block diagram interface, and provides communications engineers with comprehensive communications system design and analysis capabilities. The toolbox is useful in such diverse industries as telecommunications, telephony, aerospace, and computer peripherals.

**The Control System Toolbox.** The Control System Toolbox, the foundation of the MATLAB control design toolbox family, contains functions for modeling, analyzing, and designing automatic control systems. The application of automatic control grows each year as sensors and computers become less expensive. As a result, automatic controllers are used not only in highly technical settings for automotive and aerospace systems, computer peripherals, and process control, but also in less obvious applications such as washing machines and cameras.

**The Financial Toolbox.** The Financial Toolbox operates with MATLAB to provide a robust set of financial functions essential to financial and quantitative analysis. Applications include pricing securities, calculating interest and yield, analyzing derivatives, and optimizing portfolios. The Financial Toolbox requires the Statistics and Optimization Toolboxes. The Simulink graphical interface is recommended for Monte Carlo and non-stochastic simulations for pricing fixed-income securities, derivatives, and other instruments.

**The Frequency-Domain System Identification Toolbox.** The Frequency-Domain System Identification Toolbox by István Kollár, in cooperation with Johan Schoukens and researchers at the Vrije Universiteit in Brussels, is a set of M-files for modeling linear systems based on measurements of the system's frequency response.

**The Fuzzy Logic Toolbox.** The Fuzzy Logic Toolbox provides a complete set of GUI-based tools for designing, simulating, and analyzing fuzzy inference systems. Fuzzy logic provides an easily understandable, yet powerful way to map an input space to an output space with arbitrary complexity, with rules and relationships specified in natural language. Systems can be simulated in MATLAB or incorporated into a Simulink block diagram, with the ability to generate code for stand-alone execution.

**The Higher-Order Spectral Analysis Toolbox.** The Higher-Order Spectral Analysis Toolbox, by Jerry Mendel, C. L. (Max) Nikias, and Ananthram Swami, provides tools for signal processing using higher-order spectra. These methods are particularly useful for analyzing signals originating from a nonlinear process or corrupted by non-Gaussian noise.

**The Image Processing Toolbox.** The Image Processing Toolbox contains tools for image processing and algorithm development. It includes tools for filter design

and image restoration; image enhancement; analysis and statistics; color, geometric, and morphological operations; and 2-D transforms.

**The LMI Control Toolbox.** The LMI Control Toolbox, authored by leading researchers: Pascal Gahinet, Arkadi Nemirovski, and Alan Laub, allows one to efficiently solve Linear Matrix Inequalities (LMIs). LMIs are special convex optimization problems that arise in many disciplines, including control, identification, filtering, structural design, graph theory, and linear algebra.

The LMI Control Toolbox also features a variety of LMI-based tools for control systems design and covers applications such as robust stability and performance analysis, robust gain scheduling, and multi-objective controller synthesis with a mix of H-infinity, LQG, and pole placement objectives.

**The Model Predictive Control Toolbox.** The Model Predictive Control Toolbox was written by Manfred Morari and N. Lawrence Ricker. Model predictive control is especially useful for control applications with many input and output variables, many of which have constraints. As a result, it has become particularly popular in chemical engineering and other process control applications.

**The Mu-Analysis and Synthesis Toolbox.** The Mu-Analysis and Synthesis Toolbox, by Gary Balas, Andy Packard, John Doyle, Keith Glover, and Roy Smith, contains specialized tools for  $H_\infty$  optimal control, and  $\mu$ -analysis and synthesis, an approach to advanced robust control design of multivariable linear systems.

**The NAG Foundation Toolbox.** The NAG Foundation Toolbox includes more than 200 numeric computation functions from the well-regarded NAG Fortran subroutine libraries. It provides specialized tools for boundary-value problems, optimization, adaptive quadrature, surface and curve-fitting, and other applications.

**The Neural Network Toolbox.** The Neural Network Toolbox by Howard Demuth and Mark Beale is a collection of MATLAB functions for designing and simulating neural networks. Neural networks are computing architectures, inspired by biological nervous systems, that are useful in applications where formal analysis is extremely difficult or impossible, such as pattern recognition and nonlinear system identification and control.

**The Optimization Toolbox.** The Optimization Toolbox contains commands for the optimization of general linear and nonlinear functions, including those with

constraints. An optimization problem can be visualized as trying to find the lowest (or highest) point in a complex, highly contoured landscape. An optimization algorithm can thus be likened to an explorer wandering through valleys and across plains in search of the topographical extremes.

**The Partial Differential Equation Toolbox.** The Partial Differential Equation Toolbox extends the MATLAB Technical Computing Environment for the study and solution of PDEs in two space dimensions (2-D) and time. The PDE Toolbox provides a set of command line functions and an intuitive graphical user interface for preprocessing, solving, and postprocessing generic 2-D PDEs using the Finite Element Method (FEM). The toolbox also provides automatic and adaptive meshing capabilities and a suite of eight application modes for common PDE application areas such as heat transfer, structural mechanics, electrostatics, magnetostatics, and diffusion. These application areas are common in the fields of engineering and physics.

**The QFT Control Design Toolbox.** The Quantitative Feedback Theory Toolbox by Yossi Chait, Craig Borghesani, and Oded Yaniv implements QFT, a frequency-domain approach to controller design for uncertain systems that provides direct insight into the trade-offs between controller complexity (hence the ability to implement it) and specifications.

**The Robust Control Toolbox.** The Robust Control Toolbox provides a specialized set of tools for the analysis and synthesis of control systems that are “robust” with respect to uncertainties that can arise in the real world. The Robust Control Toolbox was created by controls theorists Richard Y. Chiang and Michael G. Safonov.

**The Signal Processing Toolbox.** The Signal Processing Toolbox contains tools for signal processing. Applications include audio (e.g., compact disc and digital audio tape), video (digital HDTV, image processing, and compression), telecommunications (fax and voice telephone), medicine (CAT scan, magnetic resonance imaging), geophysics, and econometrics.

**The Spline Toolbox.** The Spline Toolbox by Carl de Boor, a pioneer in the field of splines, provides a set of M-files for constructing and using splines, which are piecewise polynomial approximations. Splines are useful because they can approximate other functions without the unwelcome side effects that result from other kinds of approximations, such as piecewise linear curves.



**The Statistics Toolbox.** The Statistics Toolbox provides a set of M-files for statistical data analysis, modeling, and Monte Carlo simulation, with GUI-based tools for exploring fundamental concepts in statistics and probability.

**The Symbolic Math Toolbox.** The Symbolic Math Toolbox gives MATLAB an integrated set of tools for symbolic computation and variable-precision arithmetic, based on Maple V®. The Extended Symbolic Math Toolbox adds support for Maple programming plus additional specialized functions.

**The System Identification Toolbox.** The System Identification Toolbox, written by Lennart Ljung, is a collection of tools for estimation and identification. System identification is a way to find a mathematical model for a physical system (like an electric motor, or even a financial market) based only on a record of the system's inputs and outputs.

**The Wavelet Toolbox.** The Wavelet Toolbox provides a comprehensive collection of routines for examining local, multiscale, or nonstationary phenomena. Wavelet methods offer additional insight and performance in any application where Fourier techniques have been used. The toolbox is useful in many signal and image processing applications, including speech and audio processing, communications, geophysics, finance, and medicine.

## The Simulink Real-Time Workshop

The Simulink Real-Time Workshop<sup>®</sup> automatically generates C code directly from Simulink block diagrams. This allows the execution of continuous, discrete-time, and hybrid system models on a wide range of computer platforms, including real-time hardware. Simulink is required.

The Real-Time Workshop can be used for:

- **Rapid Prototyping.** As a rapid prototyping tool, the Real-Time Workshop enables you to implement your designs quickly without lengthy hand coding and debugging. Control, signal processing, and dynamic system algorithms can be implemented by developing graphical Simulink block diagrams and automatically generating C code.
- **Embedded Real-Time Control.** Once a system has been designed with Simulink, code for real-time controllers or digital signal processors can be generated, cross-compiled, linked, and downloaded onto your selected target processor. The Real-Time Workshop supports DSP boards, embedded controllers, and a wide variety of custom and commercially available hardware.
- **Real-Time Simulation.** You can create and execute code for an entire system or specified subsystems for hardware-in-the-loop simulations. Typical applications include training simulators (pilot-in-the-loop), real-time model validation, and testing.
- **Stand-Alone Simulation.** Stand-alone simulations can be run directly on your host machine or transferred to other systems for remote execution. Because time histories are saved in MATLAB as binary or ASCII files, they can be easily loaded into MATLAB for additional analysis or graphic display.

### Key Features

Real-Time Workshop provides a comprehensive set of features and capabilities that provide the flexibility to address a broad range of applications:

- Automatic code generation handles continuous-time, discrete-time, and hybrid systems.
- Optimized code guarantees fast execution.

- Control framework Application Program Interface (API) uses customizable makefiles to build and download object files to target hardware automatically.
- Portable code facilitates usage in a wide variety of environments.
- Concise, readable, and well-commented code provides ease of maintenance.
- Interactive parameter downloading from Simulink to external hardware allows system tuning on the fly.
- A menu-driven, graphical user interface makes the software easy to use.

The Real-Time Workshop supports the following target environments:

- dSPACE DS1102, DS1002, DS1003 using TI C30/C31/C40 DSPs
- VxWorks, VME/68040
- 486 PC-based systems with Xycom, Matrix, Data Translation, or Computer Boards I/O devices and Quanser Multiq board

## The Real-Time Workshop Ada Extension

The Simulink Real-Time Workshop (RTW) Ada Extension automatically generates Ada code directly from Simulink block diagrams. This allows the execution of continuous, discrete-time, and hybrid system models on a wide range of computer platforms, including real-time hardware. Simulink is required.

RTW Ada Extension can be used for:

- **Rapid Prototyping.** As a rapid prototyping tool, the RTW Ada Extension enables you to implement your designs quickly without lengthy hand coding and debugging. Control and dynamic system algorithms can be implemented by developing graphical Simulink block diagrams and automatically generating Ada code.
- **Embedded Real-Time Control.** Once a system has been designed with Simulink, code for real-time controllers can be generated, cross-compiled, linked, and downloaded onto your selected target processor. The RTW Ada Extension generates Ada code, which can be run on a wide variety of custom and commercially available hardware.
- **Real-Time Simulation.** You can create and execute code for an entire system or specified subsystems for hardware-in-the-loop simulations. Typical applications include training simulators (pilot-in-the-loop), real-time model validation, and testing.
- **Stand-Alone Simulation.** Stand-alone simulations can be run directly on your host machine or transferred to other systems for remote execution. Because time histories are saved in MATLAB as binary or ASCII files, they can be easily loaded into MATLAB for additional analysis or graphic display.

### Key Features

RTW Ada Extension provides a comprehensive set of features and capabilities that provide the flexibility to address a broad range of applications:

- Automatic code generation handles continuous-time, discrete-time, and hybrid systems.
- Optimized code guarantees fast execution.

- Control framework Application Program Interface (API) uses customizable makefiles to build and download object files to target hardware automatically.
- Portable code facilitates usage in a wide variety of environments.
- Concise, readable, and well-commented code provides ease of maintenance.
- A menu-driven, graphical user interface makes it easy to use.

The RTW Ada Extension provides turnkey solutions for the following Ada 83 compilers:

- Rational VADS for UNIX platforms
- Thomson ActivAda for Microsoft Windows Professional Edition
- Thomson ActivAda for Windows NT

## Blocksets

Similar to MATLAB and its application toolboxes, The MathWorks offers blocksets for use with Simulink. Blocksets are collections of Simulink blocks that are grouped in a separate library from the main Simulink library.

### The DSP Blockset

The DSP Blockset extends Simulink for use in the rapid design and simulation of DSP-based devices and systems. With the DSP Blockset, Simulink provides an intuitive tool for interactive block-diagram simulation and evaluation of signal processing algorithms. Its graphical programming environment makes it easier for engineers to create, modify, and prototype DSP designs. Simulink is required.

Applications for the DSP Blockset include design and analysis of communications systems, computer peripherals, speech and audio processing, automotive and aerospace controls, and medical electronics. It is ideal for both time and frequency domain algorithms, including problems such as adaptive noise cancellation.

The Fixed-Point Blockset requires Simulink 3.0 and MATLAB 5.3 and is shipping on Microsoft Windows and UNIX.

### The Fixed-Point Blockset

The Fixed-Point Blockset includes a collection of block diagram components that extend the standard Simulink block library. With this new set of blocks, you can create discrete-time dynamic systems that utilize fixed-point arithmetic. As a result, Simulink can simulate effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering. Simulink is required.

The Fixed-Point Blockset allows you to simulate fixed-point effects in a convenient and productive environment. The new blocks provided by the Fixed-Point Blockset include blocks for:

- Addition and subtraction
- Multiplication and division
- Summation
- Gains and constants

- Conversion between floating-point and fixed-point signals
- One- and two-dimensional lookup tables
- Logical operators
- Relational operators
- Conversion/saturation of fixed-point signals
- Switch between two values
- Delay
- Delta-inverse operator
- Monitoring signals

Signal conversion blocks let you convert between floating-point and fixed-point signals. Using the conversion blocks, you can create Simulink block diagrams, which consist of both standard Simulink block library components and fixed-point blocks.

For example, you can create plant models using the standard Simulink blocks and model the controller with fixed-point blocks. Data range blocks provide maximum and minimum values encountered during simulation from any point in the block diagram.

The Fixed-Point Blockset lets you build models using unsigned or two's complement 8-, 16-, or 32-bit word lengths. A combination of blocks with differing word lengths may be used in the same block diagram. Scaling of fixed-point values is achieved by specifying the location of the binary-point within the fixed-point blocks. During simulation, data types can be changed allowing you to immediately see the effects of different word sizes, binary-point locations, rounding versus truncation, and overflow checking.

Another powerful feature of this blockset is automatic location of the binary-point to give maximum precision without overflow.

By using the data range blocks, you can fix binary point locations to appropriate values.

The Fixed-Point Blockset requires Simulink 3.0 and MATLAB 5.3 and is shipping on Microsoft Windows and UNIX.

## **The Nonlinear Control Design Blockset**

The Nonlinear Control Design (NCD) Blockset offers time domain-based, robust, nonlinear control design. Controller designs are developed as block diagrams in Simulink. You select a set of tunable model parameters and graphically place time response constraints on selected output signals. Successive simulation and optimization methods are applied automatically, thereby tuning the selected model parameters.

Simulink is required with the NCD Blockset.

## **The Power System Blockset**

The Power System Blockset allows scientists and engineers to build models that simulate power systems. The blockset uses the Simulink environment, allowing a model to be built using click and drag procedures. Not only can the circuit topology be drawn rapidly, but the analysis of the circuit can include its interactions with mechanical, thermal, control, and other disciplines. This is possible because all the electrical parts of the simulation interact with Simulink's extensive modeling library. Because Simulink uses MATLAB as the computational engine, MATLAB's toolboxes can also be used by the designer.

The blockset libraries contain models of typical power equipment such as transformers, lines, machines, and power electronics. These models are proven ones coming from textbooks, and their validity is based on the experience of the Power Systems Testing and Simulation Laboratory of Hydro-Quebec, a large North American utility located in Canada. The capabilities of the blockset for modeling a typical electrical grid are illustrated in demonstration files. For users who want to refresh their knowledge of power system theory, there are also case studies available.



# Quick Start

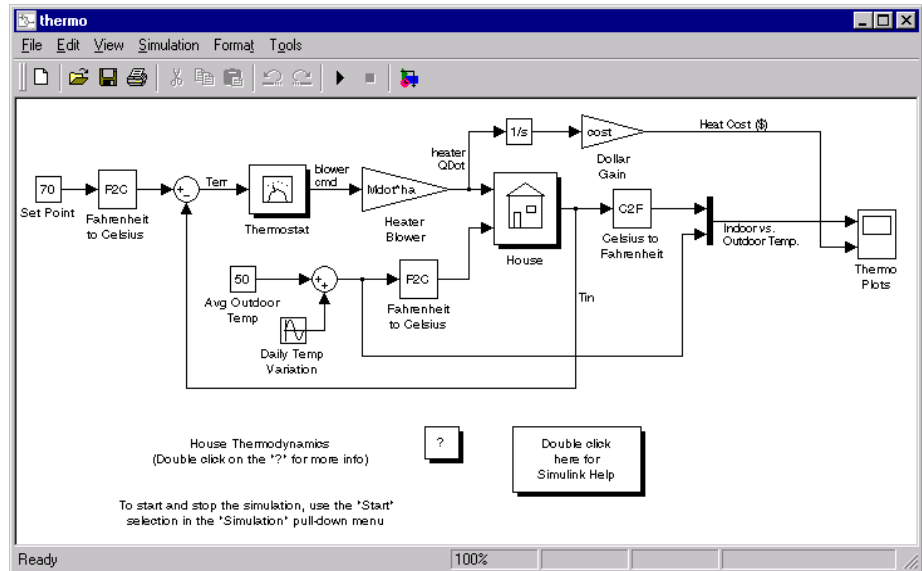
---

<b>Running a Demo Model</b> . . . . .	2-2
Description of the Demo . . . . .	2-3
Some Things to Try . . . . .	2-4
What This Demo Illustrates . . . . .	2-5
Other Useful Demos . . . . .	2-5
<b>Building a Simple Model</b> . . . . .	2-6

## Running a Demo Model

An interesting demo program provided with Simulink models the thermodynamics of a house. To run this demo, follow these steps:

- 1 Start MATLAB. See your MATLAB documentation if you're not sure how to do this.
- 2 Run the demo model by typing thermo in the MATLAB command window. This command starts up Simulink and creates a model window that contains this model.



When you open the model, Simulink opens a Scope block containing two plots labeled Indoor vs. Outdoor Temp and Heat Cost (\$), respectively.

- 3 To start the simulation, pull down the **Simulation** menu and choose the **Start** command (or, on Microsoft Windows, press the **Start** button on the Simulink toolbar). As the simulation runs, the indoor and outdoor temperatures appear in the Indoor vs. Outdoor Temp plot and the cumulative heating cost appears in the Heat Cost (\$) plot.

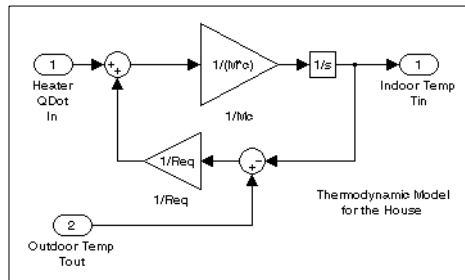
- 4 To stop the simulation, choose the **Stop** command from the **Simulation** menu (or press the **Pause** button on the toolbar). If you want to explore other parts of the model, look over the suggestions in “Some Things to Try” on page 2-4.
- 5 When you’re finished running the simulation, close the model by choosing **Close** from the **File** menu.

## Description of the Demo

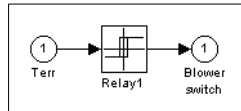
The demo models the thermodynamics of a house using a simple model. The thermostat is set to 70 degrees Fahrenheit and is affected by the outside temperature, which varies by applying a sine wave with amplitude of 15 degrees to a base temperature of 50 degrees. This simulates daily temperature fluctuations.

The model uses subsystems to simplify the model diagram and create reusable systems. A subsystem is a group of blocks that is represented by a Subsystem block. This model contains five subsystems: one named Thermostat, one named House, and three Temp Convert subsystems (two convert Fahrenheit to Celsius, one converts Celsius to Fahrenheit).

The internal and external temperatures are fed into the House subsystem, which updates the internal temperature. Double-click on the House block to see the underlying blocks in that subsystem.

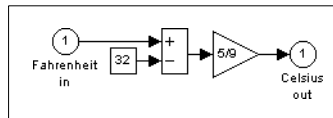


The Thermostat subsystem models the operation of a thermostat, determining when the heating system is turned on and off. Double-click on the block to see the underlying blocks in that subsystem.



Thermostat subsystem

Both the outside and inside temperatures are converted from Fahrenheit to Celsius by identical subsystems



Fahrenheit to Celsius conversion (F2C)

When the heat is on, the heating costs are computed and displayed on the Heat Cost (\$) plot on the Thermo Plots Scope. The internal temperature is displayed on the Indoor Temp Scope.

## Some Things to Try

Here are several things to try to see how the model responds to different parameters:

- Each Scope block contains one or more signal display areas and controls that enable you to select the range of the signal displayed, zoom in on a portion of the signal, and perform other useful tasks. The horizontal axis represents time and the vertical axis represents the signal value. For more information about the Scope block, see Chapter 8.
- The Constant block labeled Set Point (at the top left of the model) sets the desired internal temperature. Open this block and reset the value to 80 degrees while the simulation is running. See how the indoor temperature and heating costs change. Also, adjust the outside temperature (the Avg Outdoor Temp block) and see how it affects the simulation.
- Adjust the daily temperature variation by opening the Sine Wave block labeled Daily Temp Variation and changing the **Amplitude** parameter.

## What This Demo Illustrates

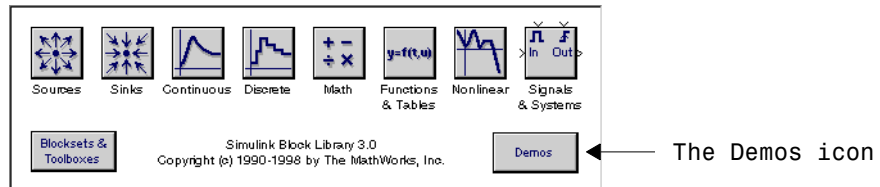
This demo illustrates several tasks commonly used when building models:

- Running the simulation involves specifying parameters and starting the simulation with the **Start** command, described in detail in Chapter 4.
- You can encapsulate complex groups of related blocks in a single block, called a subsystem. Creating subsystems is described in detail in Chapter 3.
- You can create a customized icon and design a dialog box for a block by using the masking feature, described in detail in Chapter 6. In the thermo model, all Subsystem blocks have customized icons created using the masking feature.
- Scope blocks display graphic output much as an actual oscilloscope does. Scope blocks are described in detail in Chapter 8.

## Other Useful Demos

Other demos illustrate useful modeling concepts. You can access these demos from the Simulink block library window:

- 1 Type `simulink3` in the MATLAB command window. The Simulink block library window appears.

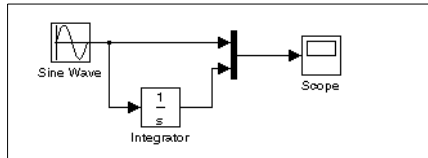


- 2 Double-click on the Demos icon. The MATLAB Demos window appears. This window contains several interesting sample models that illustrate useful Simulink features.

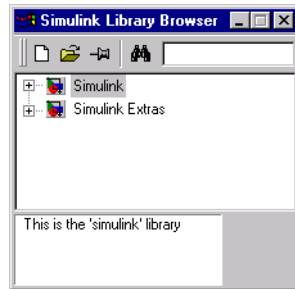
## Building a Simple Model

This example shows you how to build a model using many of the model building commands and actions you will use to build your own models. The instructions for building this model in this section are brief. All of the tasks are described in more detail in the next chapter.

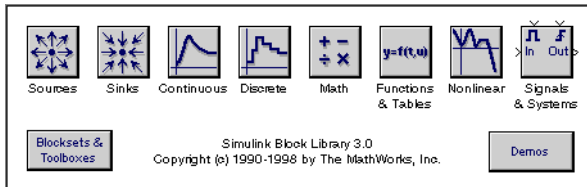
The model integrates a sine wave and displays the result, along with the sine wave. The block diagram of the model looks like this.



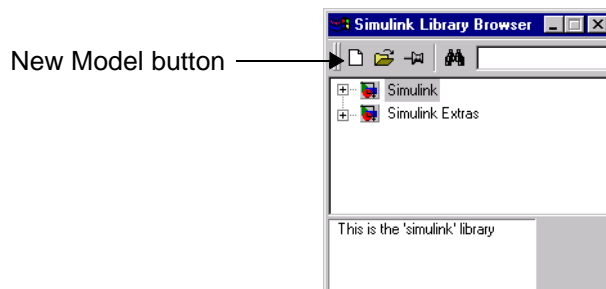
To create the model, first type `simulink` in the MATLAB command window. On Microsoft Windows, the Simulink Library Browser appears.



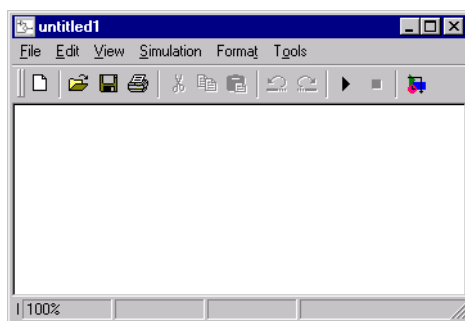
On UNIX, the Simulink library window appears.



To create a new model on UNIX, select **Model** from the **New** submenu of the Simulink library window's **File** menu. To create a new model on Windows, select the **New Model** button on the Library Browser's toolbar.



Simulink opens a new model window.



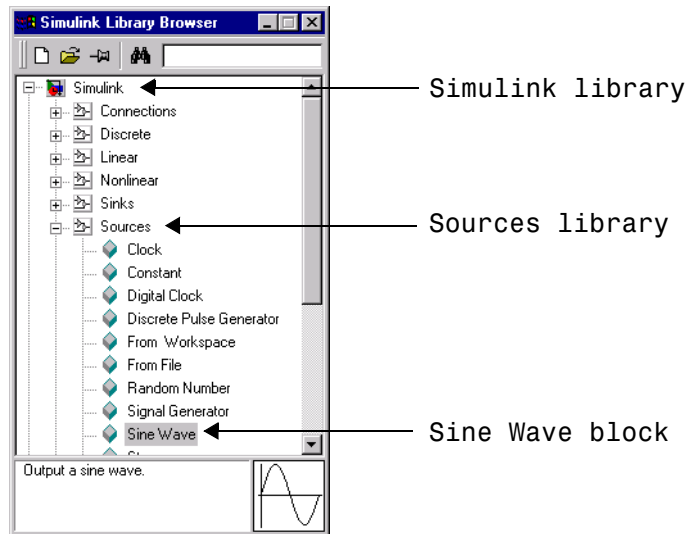
You might want to move the new model window to the right side of your screen so you can see its contents and the contents of block libraries at the same time.

To create this model, you will need to copy blocks into the model from the following Simulink block libraries:

- Sources library (the Sine Wave block)
- Sinks library (the Scope block)
- Continuous library (the Integrator block)
- Signals & Systems library (the Mux block)

You can copy a Sine Wave block from the Sources library, using the Library Browser (Windows only) or the Sources library window (UNIX or Windows).

To copy the Sine Wave block from the Library Browser, first expand the Library Browser tree to display the blocks in the Sources library. Do this by clicking first on the Simulink node to display the Sources node, then on the Sources node to display the Sources library blocks. Finally click on the Sine Wave node to select the Sine Wave block. Here is how the Library Browser should look after you have done this.

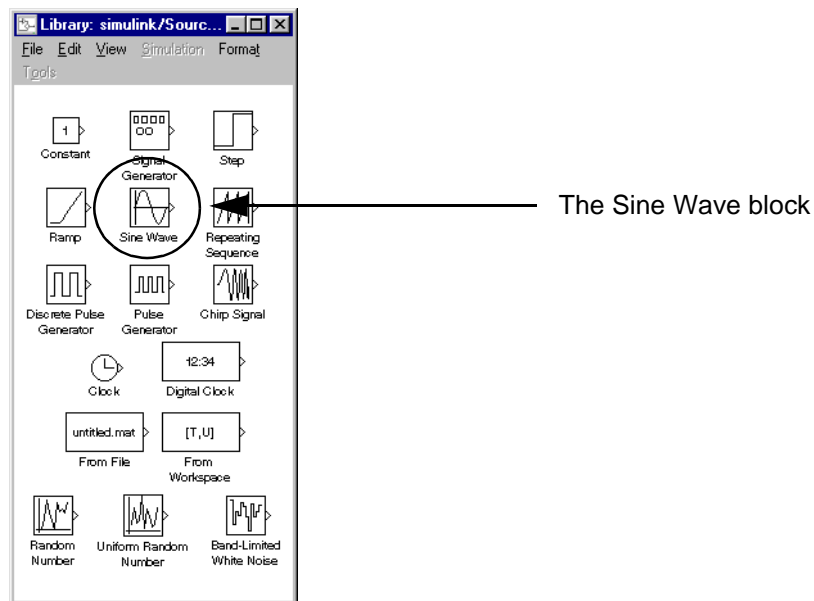


Now drag the Sine Wave node from the browser and drop it in the model window. Simulink creates a copy of the Sine Wave block at the point where you dropped the node icon.

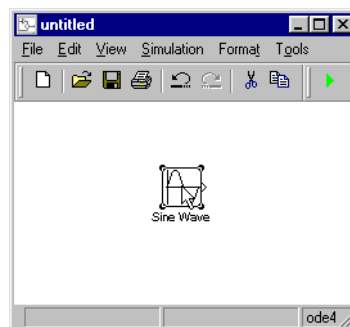
To copy the Sine Wave block from the Sources library window, open the Sources window by double-clicking on the Sources icon in the Simulink library window. (On Windows, you can open the Simulink library window by right-clicking the



Simulink node in the Library Browser and then clicking the resulting **Open Library** button.) Simulink displays the Sources library window.

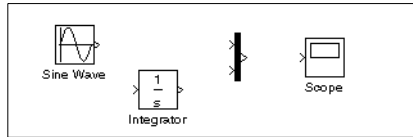


Now drag the Sine Wave block from the Sources window to your model window.

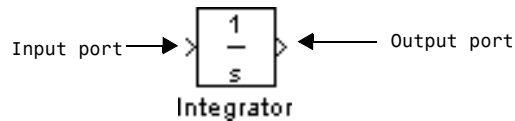


Copy the rest of the blocks in a similar manner from their respective libraries into the model window. You can move a block from one place in the model window to another by dragging the block. You can move a block a short distance by selecting the block, then pressing the arrow keys.

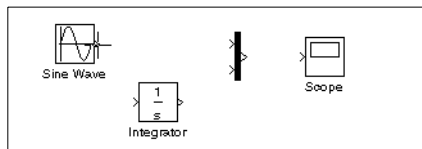
With all the blocks copied into the model window, the model should look something like this.



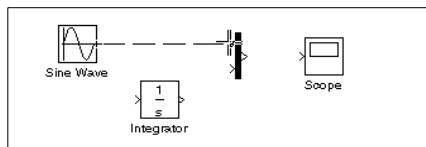
If you examine the block icons, you see an angle bracket on the right of the Sine Wave block and two on the left of the Mux block. The  $>$  symbol pointing out of a block is an *output port*; if the symbol points to a block, it is an *input port*. A signal travels out of an output port and into an input port of another block through a connecting line. When the blocks are connected, the port symbols disappear.



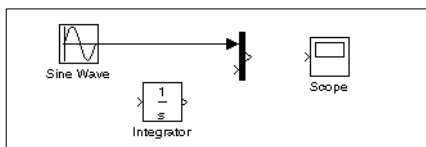
Now it's time to connect the blocks. Connect the Sine Wave block to the top input port of the Mux block. Position the pointer over the output port on the right side of the Sine Wave block. Notice that the cursor shape changes to cross hairs.



Hold down the mouse button and move the cursor to the top input port of the Mux block. Notice that the line is dashed while the mouse button is down and that the cursor shape changes to double-lined cross hairs as it approaches the Mux block.



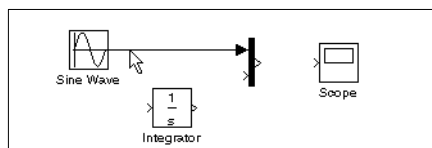
Now release the mouse button. The blocks are connected. You can also connect the line to the block by releasing the mouse button while the pointer is inside the icon. If you do, the line is connected to the input port closest to the cursor's position.



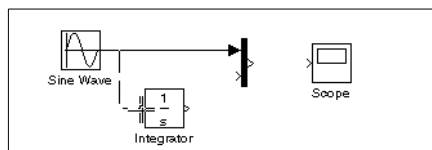
If you look again at the model at the beginning of this section (see “Building a Simple Model” on page 2-6), you’ll notice that most of the lines connect output ports of blocks to input ports of other blocks. However, one line connects a *line* to the input port of another block. This line, called a *branch line*, connects the Sine Wave output to the Integrator block, and carries the same signal that passes from the Sine Wave block to the Mux block.

Drawing a branch line is slightly different from drawing the line you just drew. To weld a connection to an existing line, follow these steps:

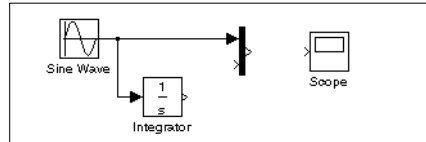
- 1 First, position the pointer *on the line* between the Sine Wave and the Mux block.



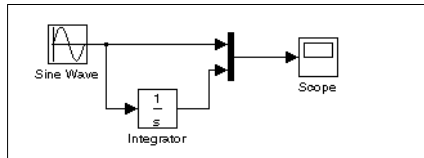
- 2 Press and hold down the **Ctrl** key. Press the mouse button, then drag the pointer to the Integrator block’s input port or over the Integrator block itself.



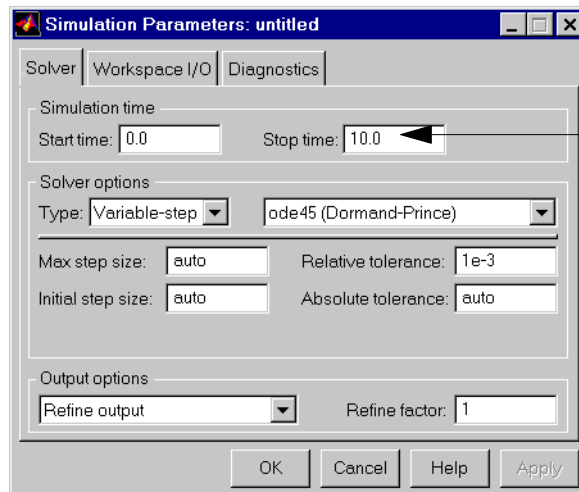
- 3 Release the mouse button. Simulink draws a line between the starting point and the Integrator block's input port.



Finish making block connections. When you're done, your model should look something like this.

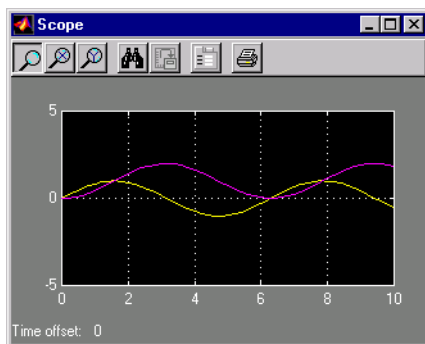


Now, open the Scope block to view the simulation output. Keeping the Scope window open, set up Simulink to run the simulation for 10 seconds. First, set the simulation parameters by choosing **Parameters** from the **Simulation** menu. On the dialog box that appears, notice that the **Stop time** is set to 10.0 (its default value).



Close the **Simulation Parameters** dialog box by clicking on the **Ok** button. Simulink applies the parameters and closes the dialog box.

Choose **Start** from the **Simulation** menu and watch the traces of the Scope block's input.



The simulation stops when it reaches the stop time specified in the **Simulation Parameters** dialog box or when you choose **Stop** from the **Simulation** menu.

To save this model, choose **Save** from the **File** menu and enter a filename and location. That file contains the description of the model.

To terminate Simulink and MATLAB, choose **Exit MATLAB** (on a Microsoft Windows system) or **Quit MATLAB** (on a UNIX system). You can also type `quit` in the MATLAB command window. If you want to leave Simulink but not terminate MATLAB, just close all Simulink windows.

This exercise shows you how to perform some commonly used model-building tasks. These and other tasks are described in more detail in Chapter 3.



# Creating a Model

---

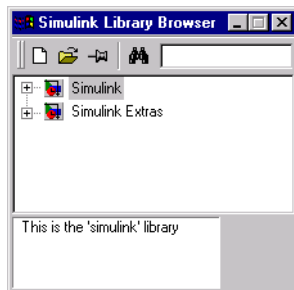
<b>Starting Simulink</b> . . . . .	3-2
<b>Selecting Objects</b> . . . . .	3-7
<b>Blocks</b> . . . . .	3-9
<b>Libraries</b> . . . . .	3-21
<b>Lines</b> . . . . .	3-27
<b>Annotations</b> . . . . .	3-37
<b>Working with Data Types</b> . . . . .	3-38
<b>Working with Complex Signals</b> . . . . .	3-47
<b>Summary of Mouse and Keyboard Actions</b> . . . . .	3-48
<b>Creating Subsystems</b> . . . . .	3-51
<b>Tips for Building Models</b> . . . . .	3-57
<b>Modeling Equations</b> . . . . .	3-58
<b>Saving a Model</b> . . . . .	3-61
<b>Printing a Block Diagram</b> . . . . .	3-62
<b>The Model Browser</b> . . . . .	3-66
<b>Tracking Model Versions</b> . . . . .	3-70
<b>Ending a Simulink Session</b> . . . . .	3-79

## Starting Simulink

To start Simulink, you must first start MATLAB. Consult your MATLAB documentation for more information. You can then start Simulink in two ways:

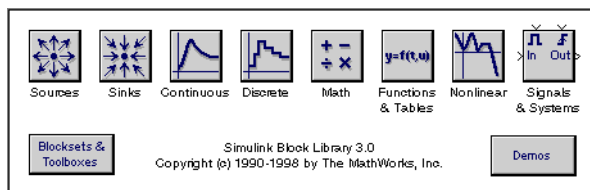
- Click on the Simulink icon  on the MATLAB toolbar.
- Enter the `simulink` command at the MATLAB prompt.

On Microsoft Windows platforms, starting Simulink displays the Simulink Library Browser.



The Library Browser displays a tree-structured view of the Simulink block libraries installed on your system. You can build models by copying blocks from the Library Browser into a model window (this procedure is described later in this chapter).

On UNIX platforms, starting Simulink displays the Simulink block library window.



The Simulink library window displays icons representing the block libraries that come with Simulink. You can create models by copying blocks from the library into a model window.



---

**Note** On Windows, you can display the Simulink library window by right-clicking the Simulink node in the Library Browser window.

---

## Creating a New Model

To create a new model, click the **New** button on the Library Browser's toolbar (Windows only) or choose **New** from the library window's **File** menu and select **Model**. You can move the window as you do other windows. Chapter 2 describes how to build a simple model. "Modeling Equations" on page 3–58 describes how to build systems that model equations.

## Editing an Existing Model

To edit an existing model diagram, either:

- Choose the **Open** button on the Library Browser's toolbar (Windows only) or the **Open** command from the Simulink library window's **File** menu and then choose or enter the model filename for the model you want to edit.
- Enter the name of the model (without the `.mdl` extension) in the MATLAB command window. The model must be in the current directory or on the path.

## Entering Simulink Commands

You run Simulink and work with your model by entering commands. You can enter commands by:

- Selecting items from the Simulink menu bar
- Selecting items from a context-sensitive Simulink menu (Windows only)
- Clicking buttons on the Simulink toolbar (Windows only)
- Entering commands in the MATLAB command window

### Using the Simulink Menu Bar to Enter Commands

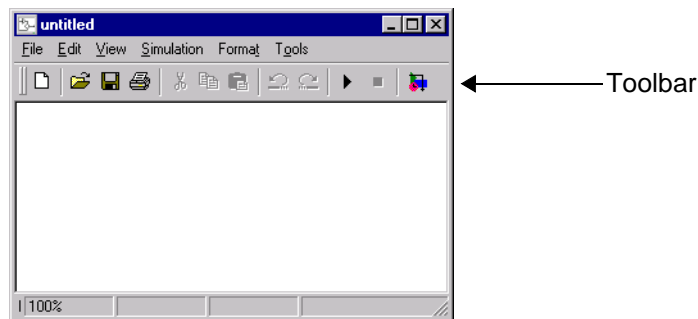
The Simulink menu bar appears near the top of each model window. The menu commands apply to the contents of that window.

#### Using Context-Sensitive Menus to Enter Commands

The Windows version of Simulink displays a context-sensitive menu when you click the right mouse button over a model or block library window. The contents of the menu depend on whether a block is selected. If a block is selected, the menu displays commands that apply only to the selected block. If no block is selected, the menu displays commands that apply to a model or library as a whole.

#### Using the Simulink Toolbar to Enter Commands

Model windows in the Windows version of Simulink optionally display a toolbar beneath the Simulink menu bar. To display the toolbar, check the **Toolbar** option on the Simulink **View** menu.



The toolbar contains buttons corresponding to frequently used Simulink commands, such as those for opening, running, and closing models. You can run such commands by clicking on the corresponding button. For example, to open a Simulink model, click on the button containing an open folder icon. You can determine which command a button executes by moving the mouse pointer over the button. A small window appears containing text that describes the button. The window is called a tooltip. Each button on the toolbar displays a tooltip when the mouse pointer hovers over it. You can hide the toolbar by unchecking the **Toolbar** option on the Simulink **View** menu.

#### Using the MATLAB Window to Enter Commands

When you run a simulation and analyze its results, you can enter MATLAB commands in the MATLAB command window. Running a simulation is discussed in Chapter 4, and analyzing simulation results is discussed in Chapter 5.

---

## 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 or deleting a block
- Adding or deleting a line
- Adding or deleting a model annotation
- Editing a block name

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

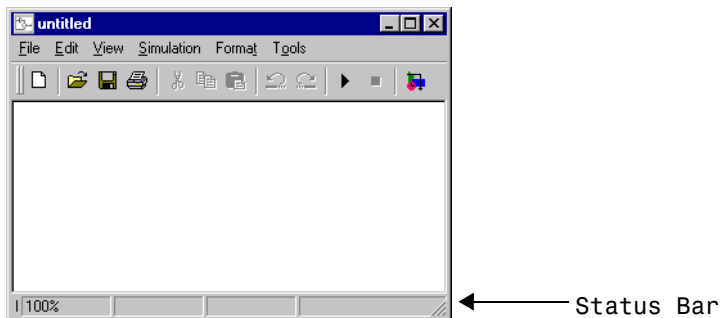
## Simulink Windows

Simulink uses separate windows to display a block library browser, a block library, a model, and graphical (scope) simulation output. These windows are not MATLAB figure windows and cannot be manipulated using Handle Graphics® commands.

Simulink windows are sized to accommodate the most common screen resolutions available. If you have a monitor with exceptionally high or low resolution, you may find the window sizes too small or too large. If this is the case, resize the window and save the model to preserve the new window dimensions.

## Status Bar

The Windows version of Simulink displays a status bar at the bottom of each model and library window.



When a simulation is running, the status bar displays the status of the simulation, including the current simulation time and the name of the current solver. You can display or hide the status bar by checking or unchecking the **Status Bar** item on the Simulink **View** menu.

### Zooming Block Diagrams

Simulink allows you to enlarge or shrink the view of the block diagram in the current Simulink 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 to view the diagram at actual size.

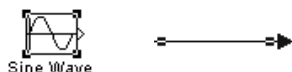
By default, Simulink fits a block diagram to view when you open the diagram either in the model browser's content pane or in a separate window. If you change a diagram's zoom setting, Simulink saves the setting when you close the diagram and restores the setting the next time you open the diagram. If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

## Selecting Objects

Many model building actions, such as copying a block or deleting a line, require that you first select one or more blocks and lines (objects).

### Selecting One Object

To select an object, click on 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 on it, any other selected objects become deselected.

### Selecting More than One Object

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.

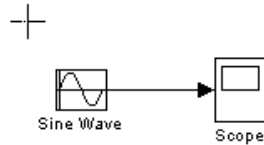
#### Selecting Multiple Objects One at a Time

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

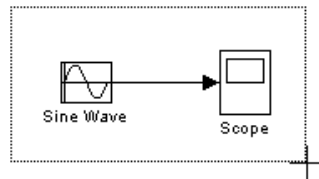
#### Selecting Multiple Objects Using a Bounding Box

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.

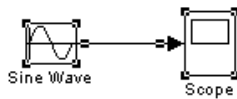
- 1 Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



- 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 the Entire Model

To select all objects in the active window, choose **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 3–51.

# Blocks

Blocks are the elements from which Simulink models are built. 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

On Microsoft Windows, Simulink displays information about a block 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** from the Simulink **View** menu.

## Virtual Blocks

When creating models, you need to be aware that Simulink blocks fall into two basic categories: nonvirtual 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 simply help to organize a model graphically. Some Simulink blocks can be virtual in some circumstances and nonvirtual in others. Such blocks are called conditionally virtual blocks. The following table lists Simulink's virtual and conditionally virtual blocks.

**Table 3-1: Virtual Blocks**

<b>Block Name</b>	<b>Condition Under Which Block Will Be Virtual</b>
Bus Selector	Always virtual.
Data Store Memory	Always virtual.
Demux	Always virtual.
Enable Port	Always virtual.
From	Always virtual.
Goto	Always virtual.
Goto Tag Visibility	Always virtual.

**Table 3-1: Virtual Blocks (Continued)**

<b>Block Name</b>	<b>Condition Under Which Block Will Be Virtual</b>
Ground	Always virtual.
Inport	Always virtual <i>unless</i> the block resides in a conditionally executed subsystem <i>and</i> has a direct connection to an output block.
Mux	Always virtual.
Output	Virtual if the block resides within any subsystem block (conditional or not), and does <i>not</i> reside in the root (top-level) Simulink window.
Selector	Always virtual.
Subsystem	Virtual if the block is not conditionally executed.
Terminator	Always virtual.
Test Point	Always virtual.
Trigger Port	Virtual if the output port is not present.

## 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, follow these steps:

- 1** Open the appropriate block library or model window.
- 2** Drag the block you want to copy into the target model window. To drag a block, position the cursor over the block icon, 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 3-25 for more information.



---

**Note** Simulink hides the names of Sum, Mux, Demux, and Bus Selector blocks 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 indicates 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 3–16.

When you copy a block, the new block inherits all the original block’s parameter values.

Simulink uses an invisible five-pixel grid to simplify the alignment of blocks. All blocks within a model snap to a line on the grid. You can move a block slightly up, down, left, or right by selecting the block and pressing the arrow keys.

You can display the grid in the model window by typing the following command in the MATLAB window:

```
set_param('<model name>', 'showgrid', 'on')
```

To change the grid spacing, type:

```
set_param('<model name>', 'gridspacing', <number of pixels>)
```

For example, to change the grid spacing to 20 pixels, type:

```
set_param('<model name>', 'gridspacing', 20)
```

For either of the above commands, you can also select the model, and then type `gcs` instead of `<model name>`.

You can copy or move blocks to compatible applications (such as word processing programs) using the **Copy**, **Cut**, and **Paste** commands. These commands copy only the graphic representation of the blocks, not their parameters.

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.

### 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 More than One Object” on page 3–7.
- 2 Drag the objects to their new location and release the mouse button.

### Duplicating Blocks in a Model

You can duplicate 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.

### Specifying Block Parameters

The Simulink user interface lets you assign values to block parameters. Some block parameters are common to all blocks. Use the **Block Properties** dialog box to set these parameters. To display the dialog box, select the block whose

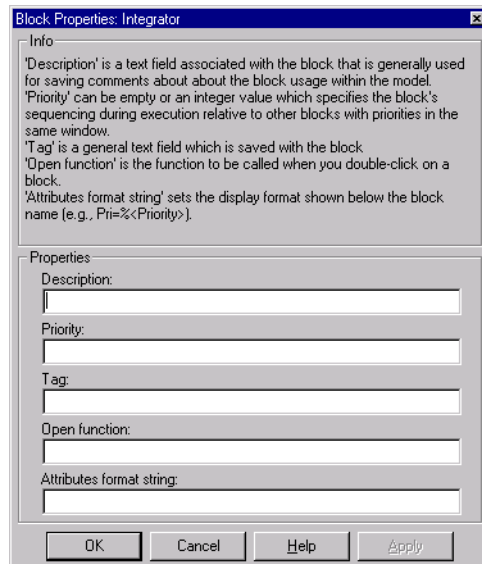
properties you want to set. Then select **Block Properties...** from Simulink's **Edit** menu. See “Block Properties Dialog Box” on page 3-13 for more information.

Other block parameters are specific to particular blocks. Use a block's block-specific parameter dialog to set these parameters. Double-click on the block to open its dialog box. You can accept the displayed values or change them. You can also use the `set_param` command to change block parameters. See `set_param` on page 10-24 for details.

Some block dialogs allow you to specify a data type for some or all of their parameters. The reference material that describes each block (in Chapter 8) shows the dialog box and describes the block parameters.

## Block Properties Dialog Box

The **Block Properties** dialog box lets you set some common block parameters.



The dialog box contains the following fields:

### 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 3-19 for more information.

#### Tag

A general text field that is saved with the block.

#### Open function

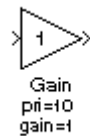
MATLAB (m-) function to be called when a user opens this block.

#### Attributes format string

Current value of the block’s AttributesFormatString parameter. This parameter specifies which parameters to display beneath a block’s icon. The attributes format string can be any text string with 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=%<prior y>\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 “???”.

#### 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.

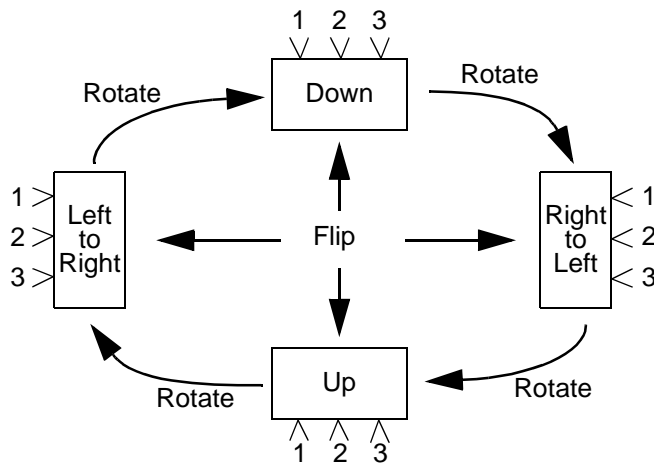
You can use the **Undo** command from the **Edit** menu to replace a deleted block.

## Changing the Orientation of Blocks

By default, signals flow through a block from left to right. Input ports are on the left, and output ports are on the right. You can change the orientation of a block by choosing one of these commands from the **Format** menu:

- The **Flip Block** command rotates the block 180 degrees.
- The **Rotate Block** command rotates a block clockwise 90 degrees.

The figure below shows how Simulink orders ports after changing the orientation of a block using the **Rotate Block** and **Flip Block** menu items. The text in the blocks shows their orientation.

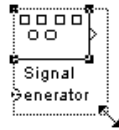


## Resizing Blocks

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 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.



## 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 this figure shows.



## Changing Block Names

You can edit a block name in one of these ways:

- To replace the block name on a Microsoft Windows or UNIX system, click on the block name, then 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 someplace 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 text on the block icon.

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 will 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 “Changing the Orientation of Blocks” on page 3–15.

## 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.

## Displaying Parameters Beneath a Block’s Icon

You can cause Simulink to display one or more of a block’s parameters beneath the block’s icon in a block diagram. 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 3-13)
- By setting the value of the block’s `AttributesFormatString` property to the format string, using `set_param` (see `set_param` on page 10-24)

### Disconnecting Blocks

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

### Vector Input and Output

Almost all Simulink blocks accept scalar or vector inputs, generate scalar or vector outputs, and allow you to provide scalar or vector parameters. These blocks are referred to in this manual as being *vectorized*.

You can determine which lines in a model carry vector signals by choosing **Wide Vector Lines** from the **Format** menu. When this option is selected, lines that carry vectors are drawn thicker than lines that carry scalars. The figures in the next section show scalar and vector lines.

If you change your model after choosing **Wide Vector Lines**, you must explicitly update the display by choosing **Update Diagram** from the **Edit** menu. Starting the simulation also updates the block diagram display.

Block descriptions in Chapter 8 discuss the characteristics of block inputs, outputs, and parameters.

### Scalar Expansion of Inputs and Parameters

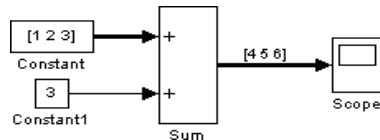
*Scalar expansion* is the conversion of a scalar value into a vector of identical elements. Simulink applies scalar expansion to inputs and/or parameters for most blocks. Block descriptions in Chapter 8 indicate whether Simulink applies scalar expansion to a block's inputs and parameters.

#### Scalar Expansion of Inputs

When using blocks with more than one input port (such as the Sum or Relational Operator block), you can mix vector and scalar inputs. When you do this, the scalar inputs are expanded into vectors of identical elements whose widths are equal to the width of the vector inputs. (If more than one block input is a vector, they must have the same number of elements.)



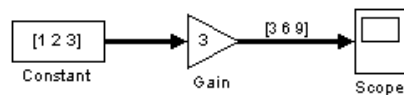
This model adds scalar and vector inputs. The input from block Constant1 is scalar expanded to match the size of the vector input from the Constant block. The input is expanded to the vector [3 3 3].



### Scalar Expansion of Parameters

You can specify the parameters for vectorized blocks as either vectors or scalars. When you specify vector parameters, each parameter element is associated with the corresponding element in the input vector(s). When you specify scalar parameters, Simulink applies scalar expansion to convert them automatically into appropriately sized vectors.

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.



### Assigning Block Priorities

You can assign evaluation priorities to nonvirtual blocks in a model. Higher priority blocks evaluate before lower priority blocks, though not necessarily before blocks that have no assigned priority.

You can assign block priorities interactively or programmatically. To set priorities programmatically, use the command

```
set_param(b, 'Priority', 'n')
```

where *b* is a block path and *n* is any valid integer. (Negative numbers and 0 are valid priority values.) The lower the number, the higher the priority; that is, 2 is higher priority than 3. To set a block's priority interactively, enter the priority in the **Priority** field of the block's **Block Properties** dialog box (see "Block Properties Dialog Box" on page 3-13).

### 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 figure below shows a Subsystem block with a drop shadow.



## Libraries

Libraries enable users to copy blocks into their models from external libraries and automatically update the copied blocks when the source blocks change. Using libraries allows users who develop their own block libraries, or who use those provided by others (such as blocksets), to ensure that their models automatically include the most recent versions of these blocks.

### Terminology

It is important to understand the terminology used with this feature.

*Library* – A collection of library blocks. A library must be explicitly created using **New Library** from the **File** menu.

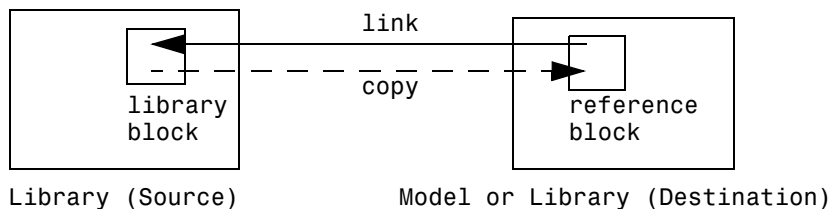
*Library block* – A block in a library.

*Reference block* – A copy of a library block.

*Link* – The connection between the reference block and its library block that allows Simulink to update the reference block when the library block changes.

*Copy* – The operation that creates a reference block from either a library block or another reference block.

This figure illustrates this terminology.



### Creating a Library

To create a library, select **Library** from the **New** submenu of the **File** menu. Simulink displays a new window, labeled **Library: untitled**. If an untitled window already appears, a sequence number is appended.

You can create a library from the command line using this command.

```
new_system('newlib', 'Library')
```

This command creates a new library named 'newlib'. To display the library, use the `open_system` command. These commands are described in Chapter 10.

The library must be named (saved) before you can copy blocks from it.

### 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.

### Copying a Library Block into a Model

You can copy a block from a library into a model by copying and pasting or dragging the block from the library window to the model window (see “Copying and Moving Blocks from One Window to Another” on page 3-10) or by dragging the block from the Library Browser (see “Browsing Block Libraries” on page 3-25) into the model window.

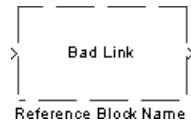
When you copy a library block into a model or another library, Simulink creates a link to the library block. The reference block is a copy of the library block. You can modify block parameters in the reference block but you cannot mask the block or, if it is masked, edit the mask. Also, you cannot set callback parameters for a reference block. If you look under the mask of a reference block, Simulink displays the underlying system for the library block.

The library and reference blocks are linked *by name*; that is, the reference block is linked to the specific block and library whose names are in effect at the time the copy is made.

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 is displayed like this (colored red).



To fix a bad link, you must either:

- 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 on the reference block. On the dialog box that appears, correct the pathname and click on **Apply** or **Close**.

All blocks have a `LinkStatus` parameter that indicates whether the block is a reference block. The parameter can have these values:

- 'none' indicates that the block is not a reference block.
- 'resolved' indicates that the block is a reference block and that the link is resolved.
- 'unresolved' indicates that the block is a reference block but that the link is unresolved.

## Updating a Linked 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 query the `LinkStatus` parameter of a block using the `get_param` command (see “Getting Information About Library Blocks” on page 3-24)
- When you use the `find_system` command

## 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 enables you to transport a model as a stand-alone model, without the libraries.

To break the link between a reference block and its library block, select the block, then choose **Break Library Link** from the **Edit** 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 save a system and break all links between reference blocks and library blocks using this command.

```
save_system('sys', 'newname', 'BreakLinks')
```

### Finding the Library Block for a Reference Block

To find the source library and block linked to a reference block, select the reference block, then choose **Go To Library Link** from the **Edit** menu. If the library is open, Simulink selects the library block (displaying selection handles on the 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

Use the `libinfo` command to get information about reference blocks in a system. The format for the command is

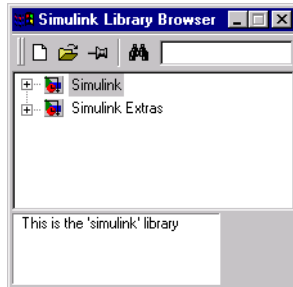
```
libdata = libinfo(sys)
```

where `sys` is the name of the system. The command returns a structure of size `n-by-1`, where `n` is the number of library blocks in `sys`. Each element of the structure has four fields:

- `Block`, the block path
- `Library`, the library name
- `ReferenceBlock`, the reference block path
- `LinkStatus`, the link status, either 'resolved' or 'unresolved'

## Browsing Block Libraries

The Library Browser lets you quickly locate and copy library blocks into a model.



---

**Note** The Library Browser is available only on Microsoft Windows platforms.

---

You can locate blocks either by navigating the Library Browser's library tree or by using the Library Browser's search facility.

### Navigating the Library Tree

The library tree displays a list of all the block libraries installed on the system. You can view or hide the contents of libraries by expanding or collapsing the tree using the mouse or keyboard. To expand/collapse the tree, click the +/- buttons next to library entries or select an entry and press the +/- or right/left arrow key on your keyboard. Use the up/down arrow keys to move up or down the tree.

### Searching Libraries

To find a particular block, enter the block's name in the edit field next to the Library Browser's **Find** button and then click the **Find** button.

### Opening a Library

To open a library, right-click the library's entry in the browser. Simulink displays an **Open Library** button. Select the **Open Library** button to open the library.

### Creating and Opening Models

To create a model, select the **New** button on the Library Browser's toolbar. To open an existing model, select the **Open** button on the toolbar.

### Copying Blocks

To copy a block from the Library Browser into a model, select the block in the browser, drag the selected block into the model window, and drop it where you want to create the copy.

### Displaying Help on a Block

To display help on a block, right-click the block in the Library Browser and select the button that subsequently pops up.

### Pinning the Library Browser

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



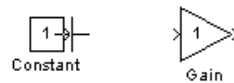
## Lines

Lines carry signals. Each line can carry a scalar or vector signal. A line can connect the output port of one block with the input port of another block. A line can also connect the output port of one block with input ports of many blocks by using branch lines.

### Drawing a Line Between Blocks

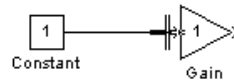
To connect the output port of one block to the input port of another block:

- 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 a cross hair.

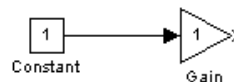


- 2 Press and hold down the mouse button.

- 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 a double cross hair.



- 4 Release the mouse button. Simulink replaces the port symbols 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 is drawn at the appropriate input port, and the signal is the same.

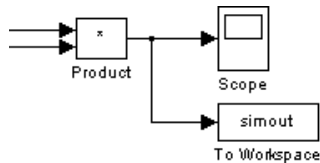


Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

## 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 carry the same signal. Using branch lines enables you to cause one signal to be carried to more than one block.

In this example, the output of the Product block goes to both the Scope block and the To Workspace block.



To add a branch line, follow these steps:

- 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

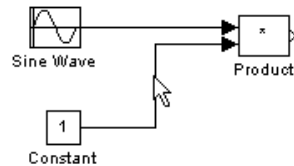
You may want to draw a line with segments exactly where you want them instead of where Simulink draws them. Or, you might want to draw a line before you copy the block to which the line is connected. You can do either by drawing line segments.

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. Simulink draws the segments as horizontal and vertical lines. To draw diagonal line segments, hold down the **Shift** key while you draw the lines.

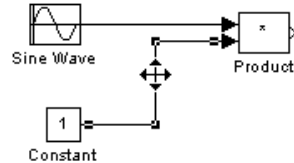
## Moving a Line Segment

To move a line segment, follow these steps:

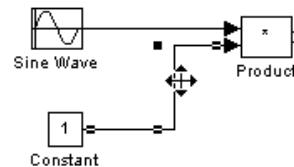
- 1 Position the pointer on the segment you want to move.



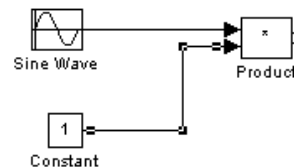
- 2 Press and hold down the left mouse button.



- 3 Drag the pointer to the desired location.



- 4 Release the mouse button.

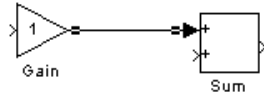


You cannot move the segments that are connected directly to block ports.

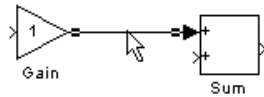
## Dividing a Line into Segments

You can divide a line segment into two segments, leaving the ends of the line in their original locations. Simulink creates line segments and a vertex that joins them. To divide a line into segments, follow these steps:

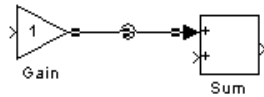
1 Select the line.



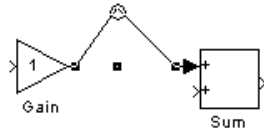
2 Position the pointer on the line where you want the vertex.



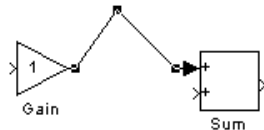
3 While holding down the **Shift** key, press and hold down the mouse button. The cursor shape changes to a circle that encloses the new vertex.



4 Drag the pointer to the desired location.



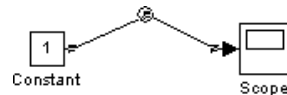
5 Release the mouse button and the **Shift** key.



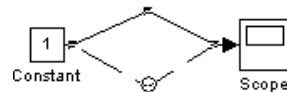
## Moving a Line Vertex

To move a vertex of a line, follow these steps:

- 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.



## Displaying Line Widths

You can display the widths of vector lines in a model by turning on **Vector Line Widths** from the **Format** menu. Simulink indicates the width of each signal at the block that originates the signal and the block that receives it. You can cause Simulink to use a thick line to display vector lines by selecting **Wide Vector Lines** from the **Format** menu.

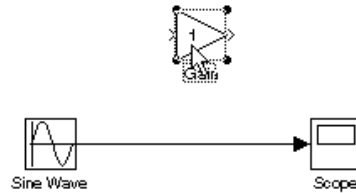
When you start a simulation or update the diagram and Simulink detects a mismatch of input and output ports, it displays an error message and shows line widths in the model.

## Inserting Blocks in a Line

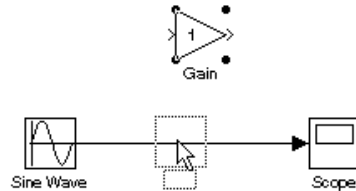
You can insert a block in a line by dropping the block on the line. Simulink inserts the block for you at the point where you drop the block. The block that you insert can have only one input and one output.

To insert a block in a line:

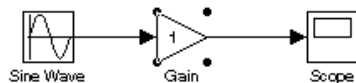
- 1 Position the pointer over the block and press the left mouse button.



- 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. Simulink inserts the block where you dropped it.



## Signal Labels

You can label signals to annotate your model. 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.

---

## Using Signal Labels

To create a signal label, double-click on the line segment and type the label at the insertion point. When you click on another part of the model, the label fixes its location.

---

**Note** When you create a signal label, take care to double-click *on* the line. If you click in an unoccupied area close to the line, you will create a model annotation instead.

---

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 a signal label, select it:

- To replace the label, click on the label, then 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 Label Propagation

Signal label propagation is the automatic labeling of a line emitting from a connection block. Blocks that support signal label propagation are the Demux, Enable, From, Inport, Mux, Selector, and Subsystem blocks. The labeled signal

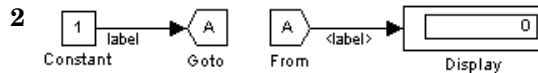
must be on a line feeding a connecting block and the propagated signal must be on a line coming from the same connecting block or one associated with it.

To propagate a signal label, create a signal label starting with the “<” character on the output of one of the listed connection blocks. When you run the simulation or update the diagram, the actual signal label appears, enclosed within angle brackets. The actual signal label is obtained by tracing back through the connection blocks until a signal label is encountered.

This example shows a model with a signal label and the propagated label both before and after updating the block diagram. In the first figure, the signal entering the Goto block is labeled label1 and the signal leaving the associated From block is labeled with a single <. The second figure shows the same model after choosing **Update Diagram** from the **Edit** menu.

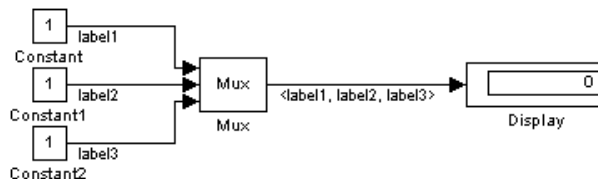


The signal label and propagated label before updating the diagram.



The same signal labels after updating the diagram.

In the next example, the propagated signal label shows the contents of a vector signal. This figure shows the label only *after* updating the diagram.



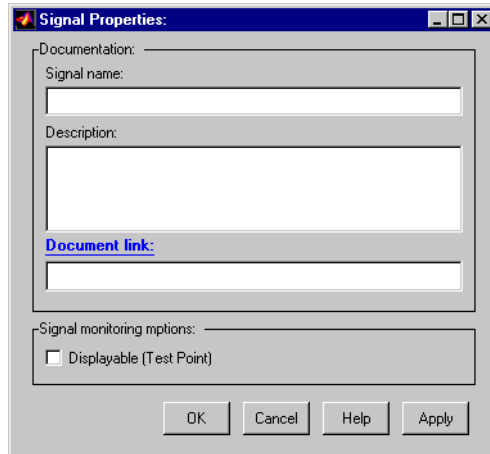
## Setting Signal Properties

Signals have properties. Use Simulink’s **Signal Properties** dialog box to view or set a signal’s properties. To display the dialog box, select the line that carries the signal and choose **Signal Properties** from the Simulink **Edit** menu.



## Signal Properties Dialog Box

The **Signal Properties** dialog box allows you to view and edit signal properties.



The dialog box includes the following controls.

### Signal Name

Name of signal.

### Description

Enter a description of the signal in this field.

### Document Link

Enter a MATLAB expression in the field that displays documentation for the signal. To display the documentation, click the field's label (that is, "Document Link"). For example, entering the expression

```
web(['file:/// ' which('foo_signal.html')])
```

in the field causes MATLAB's default Web browser to display `foo_signal.html` when you click the field's label.

### Displayable (Test Point)

Check this option to indicate that the signal can be displayed during simulation.

---

**Note** The next two controls are used to set properties used by the Real-Time Workshop to generate code from the model. You can ignore them if you do not plan to generate code from the model.

---

### **RTW storage class**

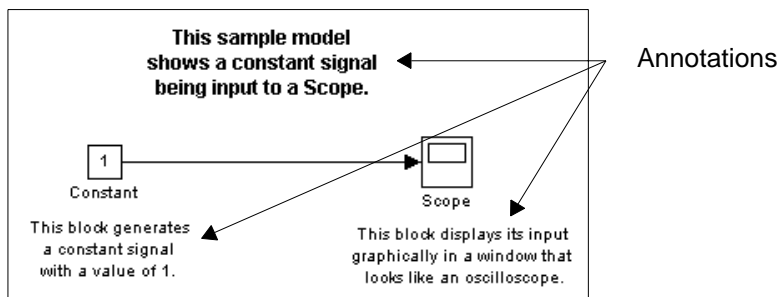
Select the storage class of this signal from the list. See the *Real-Time Workshop User's Guide* for an explanation of the listed options.

### **RTW storage type qualifier**

Select the storage type of this signal from the list. See the *Real-Time Workshop User's Guide* for more information.

## Annotations

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 on 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 within 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 on a Microsoft Windows or UNIX system, click on 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 the font of all or part of an annotation, select the text in the annotation you want to change, then choose **Font** from the **Format** menu. Select a font and size from the dialog box.

## Working with 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 enable you to take advantage of data typing to optimize the performance of MATLAB programs, MATLAB allows you to specify the data type of MATLAB variables. Simulink builds on this capability by allowing you to specify the data types of Simulink signals and block parameters.

The ability to specify the data types of a model's signals and block parameters is particularly useful in real-time control applications. For example, it allows a Simulink model to specify the optimal data types to use to represent signals and block parameters in code generated from a model by automatic code-generation tools, such as the Real-Time Workshop available from The MathWorks. By choosing the most appropriate data types for your model's signals and parameters, you can dramatically increase the 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 Simulink's 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 3-44). 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. 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 MATLAB's built-in data types.

<b>Name</b>	<b>Description</b>
double	Double-precision floating point
single	Single-precision floating point
int8	Signed eight-bit integer
uint8	Unsigned eight-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	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.

## 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. The following table lists Simulink blocks that prefer `boolean` or support multiple data types. The table also lists blocks that support complex signals.

<b>Block</b>	<b>Comments</b>
Abs	Inputs a real or complex signal of type <code>double</code> . Outputs a real signal of type <code>double</code> .
Combinatorial Logic	Input and output data type is <code>boolean</code> , if Boolean mode is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45); otherwise, <code>double</code> .
Constant	Outputs a real or complex signal of any data type.

<b>Block</b>	<b>Comments</b>
Data Type Conversion	Inputs and outputs any real or complex data type.
Demux	Accepts mixed-type signal vectors.
Display	Accepts signals of any complex or real data type.
Dot Product	Inputs and outputs real or complex values of type double.
Enable	The corresponding subsystem enable port accepts signals of type boolean or double.
From	Outputs the data type (or types) of the signal connected to the corresponding Goto block.
From Workspace	Outputs type of corresponding workspace values.
Gain	Input can be a real- or complex-valued signal or vector of any data type except boolean.
Goto	Input can be of any type.
Ground	Outputs a 0 signal of the same type as the port to which it is connected.
Hit Crossing	Inputs a double signal. Outputs boolean, if Boolean mode is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45); otherwise, double.
Inport	An inport accepts real- or complex-valued signals of any data type. The elements of an input signal vector must be of the same type if the inport is a root-level inport or the inport is directly connected to an outport of the same subsystem.
Integrator	An Integrator block accepts and outputs signals of type double on its data ports. Its external reset port accepts signals of type double or boolean.

<b>Block</b>	<b>Comments</b>
Logical Operator	Inputs and outputs real signals of type <code>boolean</code> , if Boolean mode is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45); otherwise, real signals of type <code>double</code> .
Manual Switch	Accepts real- or complex-valued signals of any type. All inputs must have the same signal and data type.
Math Function	Inputs and outputs real or complex values of type <code>double</code> .
MATLAB Function	Inputs and outputs real or complex values of type <code>double</code> .
Memory	Inputs real or complex signals of any data type.
Merge	Inputs and outputs any real or complex data type.
Multiport Switch	The control input of a Multiport Switch block accepts a real-valued signal of any type except <code>boolean</code> . The other inputs accept real- or complex-valued inputs of any type. All inputs must be of the same data and numeric type. The block outputs the type of signal on its inputs.
Mux	Accepts any supported Simulink data type, including mixed-type vectors, on each input.
Out	Accepts any Simulink data type as input. Accepts mixed-type vectors as input, if the output is in a subsystem and no initial condition is specified.
Product	Accepts real- or complex-valued signals of any data type except <code>boolean</code> . All inputs must be of the same data type.
Relational Operator	Accepts any supported data type as inputs. Both inputs must be of the same type. Outputs <code>boolean</code> , if Boolean mode is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45); otherwise, <code>double</code> .

<b>Block</b>	<b>Comments</b>
Rounding Function	Accepts and outputs real or complex values of type double.
Scope	Accepts real or complex signals of any data type.
Selector	Outputs the data types of the selected input signals.
Sum	Accepts any Simulink data type as input. All inputs must be of the same type. Outputs the same type as the input.
Switch	Accepts real- or complex-valued signals of any data type as switched inputs (inputs 1 and 3). Both switched inputs must be of the same type. The block output signal has the data type of the input. The data type of the threshold input must be boolean or double.
Terminator	Accepts any Simulink type.
To Workspace	Accepts any Simulink data type as input.
Trigger	The corresponding subsystem control port accepts signals of type boolean or double.
Trigonometric Function	Inputs and outputs real- or complex-valued signals of type double.
Unit Delay	Accepts and outputs real- or complex-valued signals of any data type.
Width	Accepts real- or complex-valued signals of any data type, including mixed-type signal vectors.
Zero-Order Hold	Accepts any Simulink data type as input.

See Chapter 8, “Block Reference” for more 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.



## Specifying Block Parameter Data Types

When entering block parameters whose data type is user-specifiable, use the syntax

```
type(value)
```

to specify the parameter, where `type` is the name of the data type and `value` is the parameter value. The following examples illustrate this syntax.

<code>single(1.0)</code>	Specifies a single-precision value of 1.0
<code>int8(2)</code>	Specifies an eight-bit integer of value 2
<code>int32(3+2i)</code>	Specifies a complex value whose real and imaginary parts are 32-bit integers

## 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 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.

## Displaying Port Data Types

To display the data types of ports in your model, select **Port Data Types** from Simulink's **Format** menu. Simulink does not update the port data type display when you change the data type of a diagram element. To refresh the display, type **Ctrl-D**.

## Data Type Propagation

Whenever you start a simulation, enable display of port data types, or refresh the port data type display, Simulink performs a processing step called data type propagation. This step involves determining the types of signals whose

type is not otherwise specified and checking the types of signals and input ports to ensure that they do not conflict. If type conflicts arise, Simulink displays an error dialog that specifies the signal and port whose data types conflict. Simulink also highlights the signal path that creates the type conflict.

---

**Note** You can insert typecasting (data type conversion) blocks in your model to resolve type conflicts. See “Typecasting Signals” on page 3-45 for more information.

---

### Data Typing Rules

Observing the following rules will help you to create models that are typesafe and therefore execute without error:

- Signal data types generally do not affect parameter data types, and vice versa.  
A significant exception to this rule is the Constant block whose output data type is determined by the data type of its parameter.
- If the output of a block is a function of an input and a parameter and the input and parameter differ in type, Simulink converts the parameter to the input type before computing the output.  
See “Typecasting Parameters” on page 3-45 for more information.
- 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 vector 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 boolean or double signals.

- Solver blocks accept only double signals.
- Connecting a nondouble signal to a block disables zero-crossing detection for that block.

## Enabling Strict Boolean Type Checking

By default, Simulink detects but does not signal an error when it detects that double signals are connected to block that prefer boolean input. This ensures compatibility with models created by earlier versions of Simulink that support only double data type. You can enable strict boolean type checking by unchecking the **Relax boolean type checking** option on the **Diagnostics** page of the **Simulation Parameters** dialog box (see “The Diagnostics Page” on page 4-24).

## Typecasting Signals

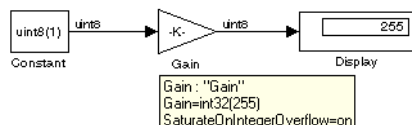
Simulink signals an error whenever it detects that a signal is connected to a block that does not accept the signal’s data type. If you want to create such a connection, you must explicitly typecast (convert) the signal to a type that the block does accept. You can use Simulink’s Data Type Conversion block to perform such conversions (see Data Type Conversion on page 8-41).

## Typecasting Parameters

In general, during simulation, Simulink silently converts parameter data types to signal data types (if they differ) when computing block outputs that are a function of an input signal and a parameter. The following exceptions occur to this rule:

- If the signal data type cannot represent the parameter value, Simulink halts the simulation and signals an error.

Consider, for example, the following model.

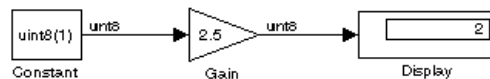


This model uses a Gain block to amplify a constant input signal. Computing the output of the Gain block requires computing the product of the input

signal and the gain. Such a computation requires that the two values be of the same data type. However, in this case, the data type of the signal, `uint8` (unsigned 8-bit word), differs from the data type of the gain parameter, `int32` (signed 32-bit integer). Thus computing the output of the gain block entails a type conversion.

When making such conversions, Simulink always casts the parameter type to the signal type. Thus, in this case, Simulink must convert the Gain block's gain value to the data type of the input signal. Simulink can make this conversion only if the input signal's data type (`uint8`) can represent the gain. In this case, Simulink can make the conversion because the gain is 255, which is within the range of the `uint8` data type (0 to 255). Thus, this model simulates without error. However, if the gain were slightly larger (for example, 256), Simulink would signal an out-of-range error if you attempted to simulate the model.

- If the signal data type can represent the parameter value but only at reduced precision, Simulink issues a warning message and continues the simulation. Consider, for example, the following model.



In this example, the signal type accommodates only integer values while the gain value has a fractional component. Simulating this model causes Simulink to truncate the gain to the nearest integral value (2) and issue a loss-of-precision warning. On the other hand, if the gain were 2.0, Simulink would simulate the model without complaint because in this case the conversion entails no loss of precision.

---

**Note** Conversion of an `int32` parameter to a `float` or `double` can entail a loss of precision. The loss can be severe if the magnitude of the parameter value is large. If an `int32` parameter conversion does entail a loss of precision, Simulink issues a warning message.

---

## Working with Complex Signals

By default, the values of Simulink signals are real numbers. However, models can create and manipulate signals that have complex numbers as values.

You can introduce a complex-valued signal into a model in any of the following ways:

- Load complex-valued signal data from the MATLAB workspace into the model via a root-level inport.
- 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 and then combine the parts into a complex signal, using Real-Imag to Complex conversion block.

You can manipulate complex signals via blocks that accept them. Most Simulink blocks accept complex signals as input. If you are not sure whether a block accepts complex signals, refer to the documentation for the block in Chapter 8, “Block Reference.”

## Summary of Mouse and Keyboard Actions

These tables summarize the use of the mouse and keyboard to manipulate blocks, lines, and signal labels. LMB means press the left mouse button; CMB, the center mouse button; and RMB, the right mouse button.

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

**Table 3-2: Manipulating Blocks**

<b>Task</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Select one block	LMB	LMB
Select multiple blocks	<b>Shift</b> + LMB	<b>Shift</b> + LMB; or CMB alone
Copy block from another window	Drag block	Drag block
Move block	Drag block	Drag block
Duplicate block	<b>Ctrl</b> + LMB and drag; or RMB and drag	<b>Ctrl</b> + LMB and drag; or RMB and drag
Connect blocks	LMB	LMB
Disconnect block	<b>Shift</b> + drag block	<b>Shift</b> + drag block; or CMB and drag

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

**Table 3-3: Manipulating Lines**

<b>Task</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Select one line	LMB	LMB
Select multiple lines	<b>Shift</b> + LMB	<b>Shift</b> + LMB; or CMB alone
Draw branch line	<b>Ctrl</b> + drag line; or <b>RMB</b> and drag line	<b>Ctrl</b> + drag line; or RMB + drag line

**Table 3-3: Manipulating Lines (Continued)**

<b>Task</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Route lines around blocks	<b>Shift</b> + draw line segments	<b>Shift</b> + draw line segments; or CMB and draw segments
Move line segment	Drag segment	Drag segment
Move vertex	Drag vertex	Drag vertex
Create line segments	<b>Shift</b> + drag line	<b>Shift</b> + drag line; or CMB + drag line

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

**Table 3-4: Manipulating Signal Labels**

<b>Action</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Create signal label	Double-click on line, then type label	Double-click on line, then type label
Copy signal label	<b>Ctrl</b> + drag label	<b>Ctrl</b> + 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	<b>Shift</b> + click on label, then press <b>Delete</b>	<b>Shift</b> + click on label, then press <b>Delete</b>

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

**Table 3-5: Manipulating Annotations**

<b>Action</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Create annotation	Double-click in diagram, then type text	Double-click in diagram, then type text
Copy annotation	<b>Ctrl</b> + drag label	<b>Ctrl</b> + drag label

**Table 3-5: Manipulating Annotations (Continued)**

<b>Action</b>	<b>Microsoft Windows</b>	<b>UNIX</b>
Move annotation	Drag label	Drag label
Edit annotation	Click in text, then edit	Click in text, then edit
Delete annotation	<b>Shift</b> + select annotation, then press <b>Delete</b>	<b>Shift</b> + select annotation, then press <b>Delete</b>



## Creating Subsystems

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.

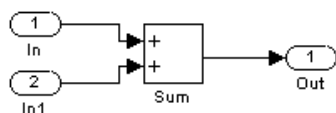
You can create a subsystem in two ways:

- 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.

### 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 Signals & Systems library into your model.
- 2 Open the Subsystem block by double-clicking on it.
- 3 In the empty Subsystem window, create the subsystem. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output. For example, the subsystem below includes a Sum block and Inport and Outport blocks to represent input to and output from the subsystem:

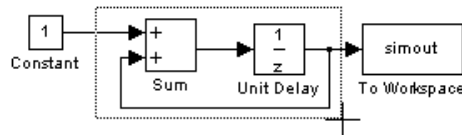


## Creating a Subsystem by Grouping Existing Blocks

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

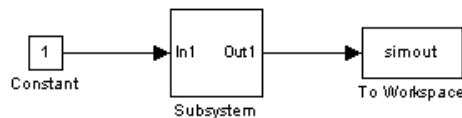
- 1 Enclose the blocks and connecting lines that you want to include in the subsystem within a bounding box. You cannot specify the blocks to be grouped by selecting them individually or by using the **Select All** command. For more information, see “Selecting Multiple Objects Using a Bounding Box” on page 3–7.

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

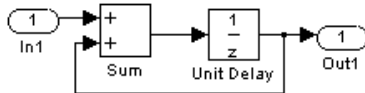


When you release the mouse button, the two blocks and all the connecting lines are selected.

- 2 Choose **Create Subsystem** from the **Edit** menu. Simulink replaces the selected blocks with a Subsystem block. This figure shows the model after choosing the **Create Subsystem** command (and resizing the Subsystem block so the port labels are readable).



If you open the Subsystem block, Simulink displays the underlying system, as shown below. Notice that Simulink adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem.



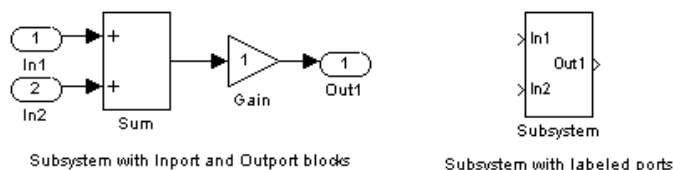
As with all blocks, you can change the name of the Subsystem block. Also, you can customize the icon and dialog box for the block using the masking feature, described in Chapter 6.

## 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 the port labels by selecting the Subsystem block, then choosing **Hide Port Labels** from the **Format** menu. You can also hide one or more port labels by selecting the appropriate Inport or Outport block in the subsystem and choosing **Hide Name** from the **Format** menu.

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.



## Using Callback Routines

You can define MATLAB expressions that execute when the block diagram or a block is acted upon in a particular way. These expressions, called *callback routines*, are associated with block or model parameters. For example, the callback associated with a block's `OpenFcn` parameter is executed when the model user double-clicks on that block's name or path changes.

To define callback routines and associate them with parameters, use the `set_param` command (see `set_param` on page 10-24).

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

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

You can examine the `clutch` system (`clutch.mdl`) for routines associated with many model callbacks.

These tables list the parameters for which you can define callback routines, and indicate when those callback routines are executed. Routines that are executed before or after actions take place occur immediately before or after the action.

**Table 3-6: Model Callback Parameters**

<b>Parameter</b>	<b>When Executed</b>
CloseFcn	Before the block diagram is closed.
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.
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 StopFcn is executed.

**Table 3-7: Block Callback Parameters**

<b>Parameter</b>	<b>When Executed</b>
CloseFcn	When the block is closed using the <code>close_system</code> command.
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.
DeleteFcn	Before a block is deleted. This callback is recursive for Subsystem blocks.
DestroyFcn	When block has been destroyed.
InitFcn	Before the block diagram is compiled and before block parameters are evaluated.
LoadFcn	After the block diagram is loaded. This callback is recursive for Subsystem blocks.
ModelCloseFcn	Before the block diagram is closed. This callback is recursive for Subsystem blocks.
MoveFcn	When block is moved or resized.
NameChangeFcn	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.

**Table 3-7: Block Callback Parameters (Continued)**

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 on 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> on page 10-19).
PreSaveFcn	Before the block diagram is saved. This callback is recursive for Subsystem blocks.
PostSaveFcn	After the block diagram is saved. This callback is recursive for Subsystem blocks.
StartFcn	After the block diagram is compiled and before the simulation starts.
StopFcn	At any termination of the simulation.
UndoDeleteFcn	When a block delete is undone.

## Tips for Building Models

Here are some model-building hints you might find useful:

- Memory issues

In general, the more memory, the better Simulink performs.

- 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 3–51. The Model Browser (see “The Model Browser” on page 3-66) provides useful information about complex models.

- Cleaning up models

Well organized and documented models are easier to read and understand. Signal labels and model annotations can help describe what is happening in a model. For more information, see “Signal Labels” on page 3–32 and “Annotations” on page 3–37.

- 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 Equations

One of the most confusing issues for new Simulink users is how to model equations. Here are some examples that may improve your understanding of how to model equations.

### Converting Celsius to Fahrenheit

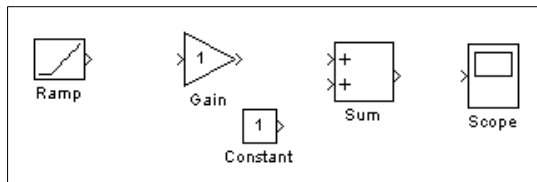
To model the equation that converts Celsius temperature to Fahrenheit:

$$T_F = 9/5(T_C) + 32$$

First, consider the blocks needed to build the model:

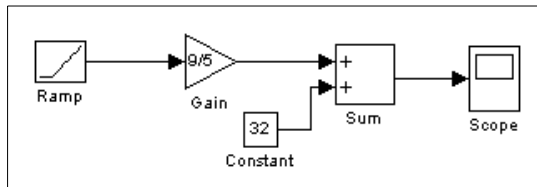
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block, to define a constant of 32, also from the Sources library
- A Gain block, to multiply the input signal by 9/5, from the Math library
- A Sum block, to add the two quantities, also from the Math library
- A Scope 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 on) each block and entering the appropriate value. Then, click on the **Close** 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 will run for 10 seconds.

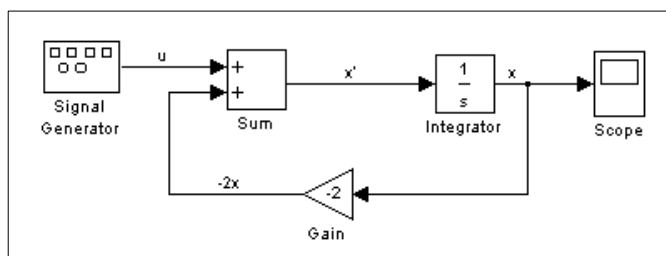
## Modeling a Simple Continuous System

To model the differential equation

$$x'(t) = -2x(t) + u(t)$$

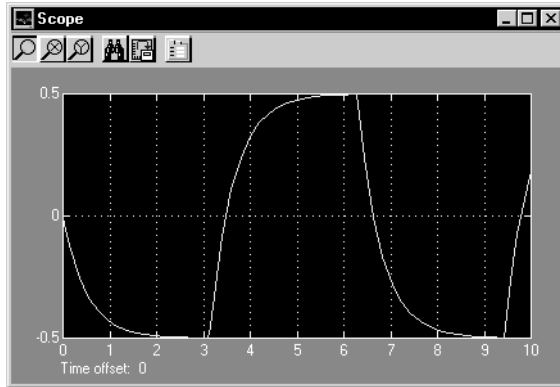
where  $u(t)$  is a square wave with an amplitude of 1 and a frequency of 1 rad/sec. The Integrator block integrates its input,  $x'$ , to produce  $x$ . Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator block and select the Square 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. Also, 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 3–28. 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 equation above.

$$sx = -2x + u$$

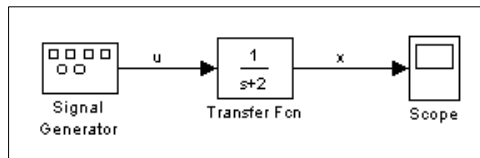
Solving for  $x$  gives

$$x = u/(s + 2)$$

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 model now becomes quite simple:



The results of this simulation are identical to those of the previous model.

## Saving a Model

You can save a model by choosing either the **Save** or **Save As** command from the **File** menu. Simulink saves the model by generating a specially formatted file called the *model file* (with the `.mdl` extension) that contains the block diagram and block properties. The format of the model file is described in Appendix B.

If you are saving a model for the first time, use the **Save** command to provide a name and location to the model file. Model file names must start with a letter and can contain no more than 31 letters, numbers, and underscores.

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.

Simulink 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 Simulink executes all block `PreSaveFcn` callback routines, then executes the block diagram's `PreSaveFcn` callback routine.
- 3 Simulink writes the model file to a new file using the same name and an extension of `mdl`.
- 4 Simulink executes all block `PostSaveFcn` callback routines, then executes the block diagram's `PostSaveFcn` callback routine.
- 5 Simulink deletes the temporary file.

If an error occurs during this process, Simulink renames the temporary file to the name of the original model file, writes the current version of the model to a file with an `.err` extension, and issues an error message. Simulink performs steps 2 through 4 even if an error occurs in an earlier step.

## Printing a Block Diagram

You can print a block diagram by selecting **Print** from the **File** menu (on a Microsoft Windows system) or by using the print command in the MATLAB command window (on all platforms).

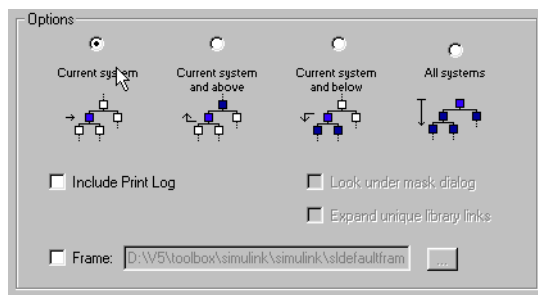
On a Microsoft Windows system, the **Print** menu item prints the block diagram in the current window.

### Print Dialog Box

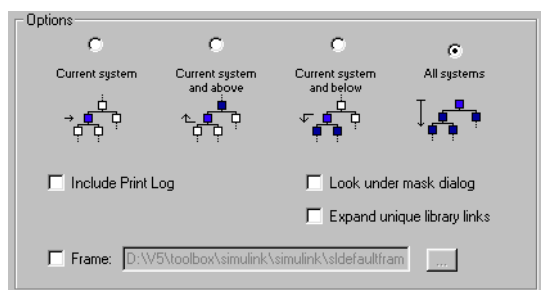
When you select the **Print** menu item, the **Print** dialog box appears. The **Print** dialog box enables you to selectively print systems within your model. Using the dialog box, you can:

- Print the current system only
- Print the current system and all systems above it in the model hierarchy
- Print 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
- Print all systems in the model, with the option of looking into the contents of masked and library blocks
- Print 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 Microsoft Windows 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 printing all systems, the top-level system is considered the current block so Simulink 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 “Libraries” on page 3-21.

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

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 MATLAB’s frame editor. See `frameedit` in the online MATLAB reference for information on using the frame editor to create title block frames.

## Print Command

The format of the print command is

```
print -ssys -device 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 *Using MATLAB Graphics*.

*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'])
```

## Specifying Paper Size and Orientation

Simulink lets you 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 Parameters” on page A-3), using the set\_param command. You can set the paper orientation alone, using MATLAB's orient command. On Windows, the **Print** dialog box lets 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, Simulink 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, Simulink centers the model diagram on the printed page, scaling the diagram, if necessary, to fit the page.

## The Model Browser

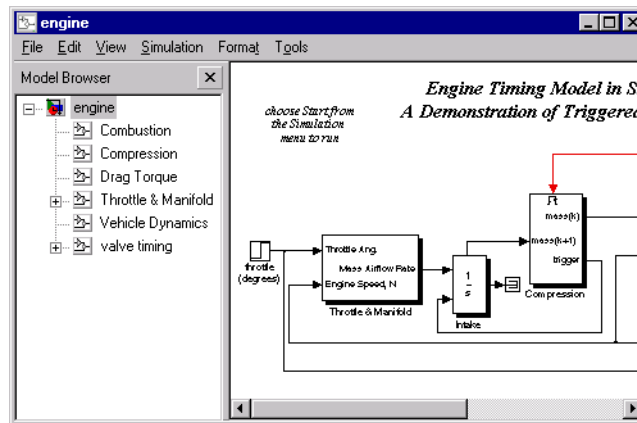
The Model Browser enables you to:

- Navigate a model hierarchically
- Open systems in a model directly
- Determine the blocks contained in a model

The browser operates differently on Microsoft Windows and UNIX platforms.

### Using the Model Browser on Windows

To display the Model Browser pane, select **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.

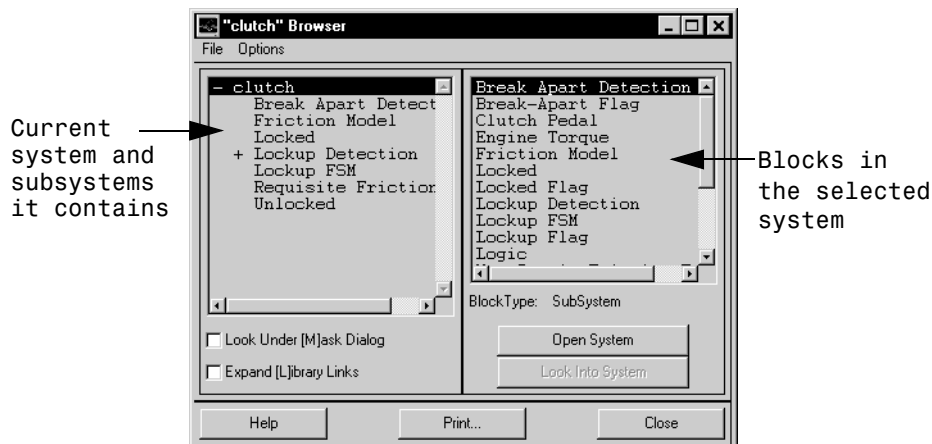


Each entry in the tree view corresponds to a subsystem in the model. You can expand/collapse the tree by clicking on the +/- boxes beside each subsystem, or by pressing the left/right arrow or +/- keys on your numeric keypad. You can move up/down the tree by pressing the up/down arrow on your keypad. Click on any subsystem to display its contents in the diagram view. To open a new window on a subsystem, double click the subsystem in the diagram view.



## Using the Model Browser on UNIX

To open the Model Browser, select **Show Browser** from the **File** menu. The Model Browser window appears, displaying information about the current model. This figure shows the Model Browser window displaying the contents of the clutch system.



### Contents of the Browser Window

The Model Browser window consists of:

- The systems list. The list on the left contains the current system and the subsystems it contains, with the current system selected.
- The blocks list. The list on the right contains the names of blocks in the selected system. Initially, this window displays blocks in the top-level system.
- The **File** menu, which contains the **Print**, **Close Model**, and **Close Browser** menu items.
- The **Options** menu, which contains these menu items: **Open System**, **Look Into System**, **Display Alphabetical/Hierarchical List**, **Expand All**, **Look Under Mask Dialog**, and **Expand Library Links**.
- **Options** check boxes and buttons: **Look Under [M]ask Dialog** and **Expand [L]ibrary Links** check boxes, and **Open System** and **Look Into System** buttons. By default, Simulink does not display contents of masked blocks and

blocks that are library links. These check boxes enable you to override the default.

- The block type of the selected block.
- Dialog box buttons: **Help**, **Print**, and **Close**.

### Interpreting List Contents

Simulink identifies masked blocks, reference blocks, blocks with defined OpenFcn parameters, and systems that contain subsystems using these symbols before a block or system name:

- A plus sign (+) before a system name in the systems list indicates that the system is expandable, which means that it has systems beneath it. Double-click on the system name to expand the list and display its contents in the blocks list. When a system is expanded, a minus sign (–) appears before its name.
- [M] indicates that the block is masked, having either a mask dialog box or a mask workspace. For more information about masking, see Chapter 6.
- [L] indicates that the block is a reference block. For more information, see “Libraries” on page 3-21.
- [O] indicates that an open function (OpenFcn) callback is defined for the block. For more information about block callbacks, see “Using Callback Routines” on page 3-53.
- [S] indicates that the system is a Stateflow<sup>®</sup> block.

### Opening a System

You can open any block or system whose name appears in the blocks list. To open a system:

- 1 In the systems list, select by single-clicking on the name of the parent system that contains the system you want to open. The parent system’s contents appear in the blocks list.
- 2 Depending on whether the system is masked, linked to a library block, or has an open function callback, you open it as follows:

- If the system has no symbol to its left, double-click on its name or select its name and click on the **Open System** button.
- If the system has an [M] or [O] before its name, select the system name and click on the **Look Into System** button.

### Looking into a Masked System or a Linked Block

By default, the Model Browser considers masked systems (identified by [M]) and linked blocks (identified by [L]) as blocks and not subsystems. If you click on **Open System** while a masked system or linked block is selected, the Model Browser displays the system or block's dialog box (**Open System** works the same way as double-clicking on the block in a block diagram). Similarly, if the block's `OpenFcn` callback parameter is defined, clicking on **Open System** while that block is selected executes the callback function.

You can direct the Model Browser to look beyond the dialog box or callback function by selecting the block in the blocks list, then clicking on **Look Into System**. The Model Browser displays the underlying system or block.

### Displaying List Contents Alphabetically

By default, the systems list indicates the hierarchy of the model. Systems that contain systems are preceded with a plus sign (+). When those systems are expanded, the Model Browser displays a minus sign (–) before their names. To display systems alphabetically, select the **Display Alphabetical List** menu item on the **Options** menu.

## Tracking Model Versions

A Simulink model can go through many versions during its development. Simulink helps you to track the various versions by generating and storing version control information, including the version number, the persons who created and last updated the model, and optionally a change history. The following Simulink features allow you to manage and use the version control information:

- The Simulink **Model Parameters** dialog box allows you to edit some of the version control information stored in the model and to select various version control options.
- The Simulink Model Info block allows you to display version control information, including that maintained by an external version control system, as an annotation block in a model diagram.
- Simulink version control parameters allow you to access version control information from the MATLAB command line or an M-file.

### Specifying the Current User

When a user creates or updates a model, Simulink logs the user's name in the model for version control purposes. Simulink assumes that the user's 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, Simulink does not update the user name in the model.

UNIX systems always define the USER environment variable and set its value to the name you use to log onto your system. Thus, if you are using a UNIX system, you do not have to do anything to enable Simulink to identify you as the current user. Windows systems, on the other hand, may define some or none of the "user name" environment variables that Simulink expects, depending on the version of Windows installed on your system and whether it operates stand-alone or connected to a network. You can use the MATLAB command `getenv` to determine which if any of the environment variables is defined. For example, enter

```
getenv('user')
```

at the MATLAB command line to determine if the USER environment variable exists on your Windows system. If not, you must set it yourself. On Windows 95 and 98, set the value by entering the following line

```
set user=yourname
```

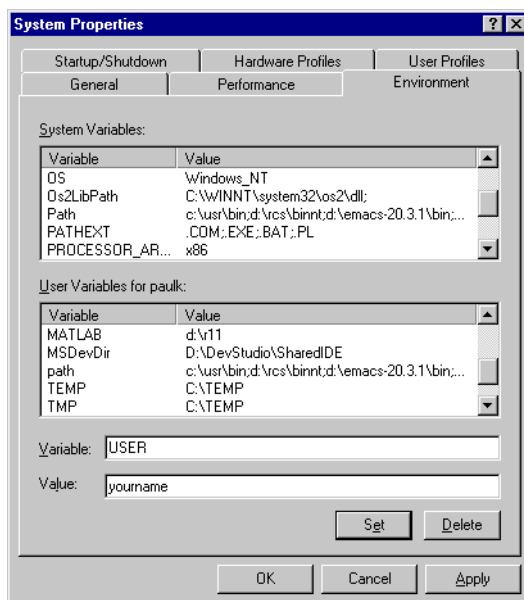
in your system's autoexec.bat file, where yourname is the name by which you want to be identified in a model file. Then, reboot your computer.

---

**Note** The autoexec.bat file typically is found in the c:\ directory on your system's hard disk.

---

On Windows NT, use the **Environment** panel of the Windows NT **System Properties** dialog box to set the USER environment variable (if it is not already defined).

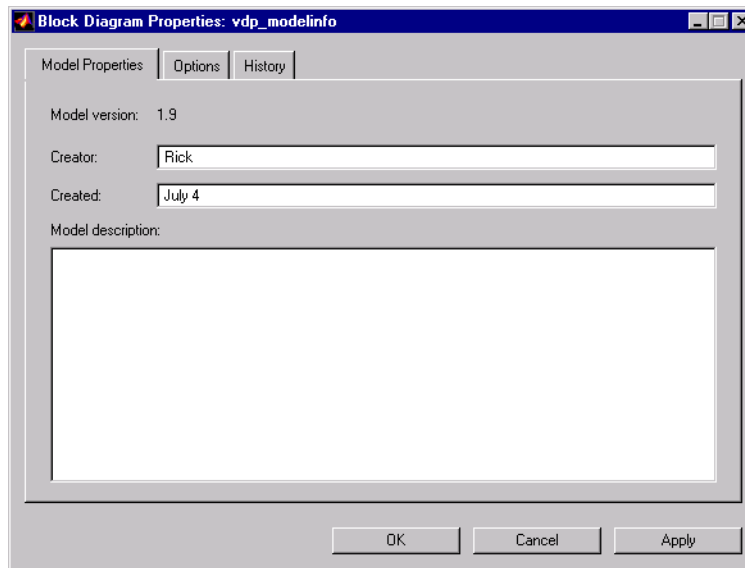


To display the **System Properties** dialog box, select **System** in your system's **Control Panel** folder, which resides in your system's **My Computer** folder, which resides on your Windows NT desktop. To set the USER variable, enter

USER in the **Variable** field, your login name in the **Value** field, and select the **Set** button. Then select **OK** to dismiss the dialog box.

### Model Properties Dialog

The **Model Properties** dialog box allows you to edit some version control parameters and set some related options. To display the dialog box, choose **Model Parameters** from the Simulink **File** menu.



#### Model Properties Pane

The **Model Properties** pane lets you edit the following version control parameters.

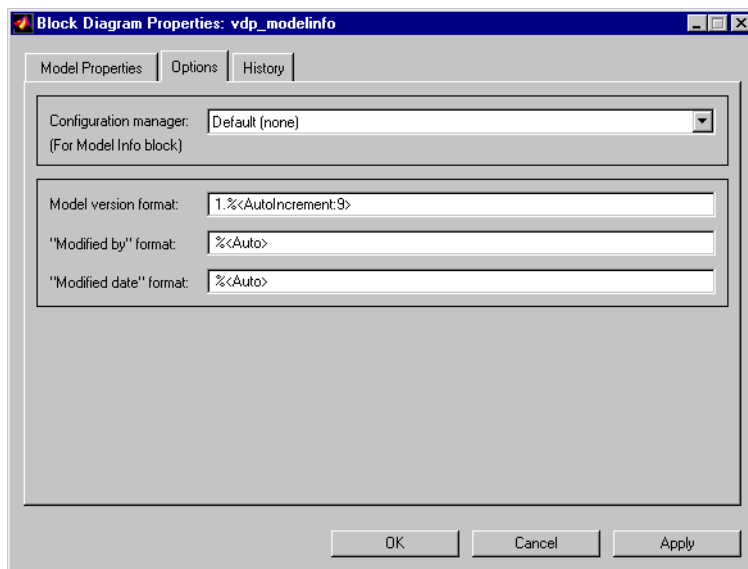
**Creator.** Name of the person who created this model. Simulink sets this property to the value of the USER environment variable when you create the model. Edit this field to change the value.

**Created.** Date and time this model was created.

**Model description.** Description of the model.

## Options Pane

The Options pane lets you choose a configuration manager and specify version control information formats.



**Configuration manager.** External configuration manager used to manage this model. Choosing this option allows you to include information from the configuration manager in a Model Info annotation block. See *Model Info* on page 8-131 for more information.

The file `cmopts.m` in the `MATLABROOT/toolbox/local` directory specifies the default configuration manager for models. The default “default” configuration manager is none. You can edit this file to specify another choice.

**Model version format.** Format used to display the model version number in the **Model Parameters** pane and in Model Info blocks. The value of this parameter can be any text string. The text string can include occurrences of the tag `%<AutoIncrement:#>` where `#` is an integer. Simulink replaces the tag with `#` when displaying the model’s version number. For example, it displays

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

as

```
1.2
```

Simulink increments # by 1 when saving the model. For example,

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

becomes

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

when you save the model.

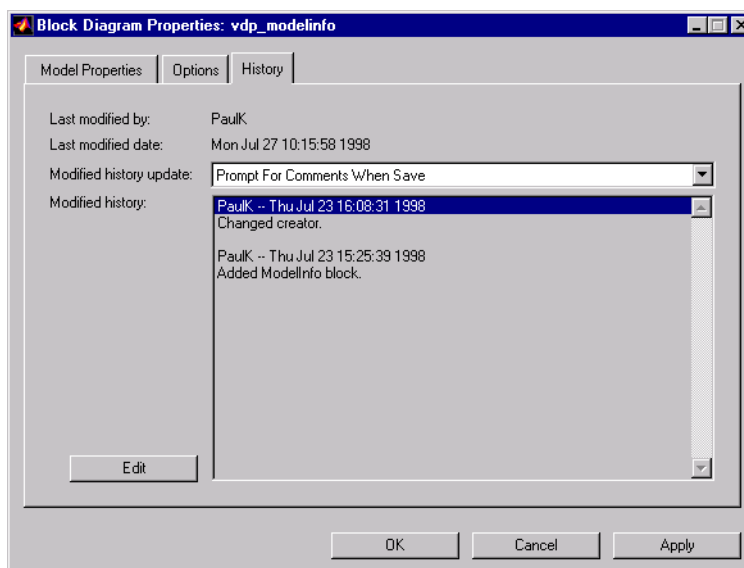
**Modified by format.** Format used to display the “Modified By” value in the **History** pane, in the history log, and in Model Info blocks. The value of this field can be any string. The string can include the tag `%<Auto>`. Simulink replaces occurrences of this tag with the current value of the `USER` environment variable.

**Modified date format.** Format used to display the “Last modified date” in the **History** pane, in the history log, and in Model Info blocks. The value of this field can be any string. The string can contain the tag `%<Auto>`. Simulink replaces occurrences of this tag with the current date and time.



## History Pane

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



**Last modified by.** Name of the person who last modified this model. Simulink sets the value of this parameter to the value of the USER environment variable when you save a model. You cannot edit this field.

**Last modified date.** Date that this model was last modified. Simulink sets the value of this parameter to the system date and time when you save a model. You cannot edit this field.

**Modified history update.** Specifies whether to prompt a user for a comment when this model is saved. If you choose “Prompt for Comments When Save,” Simulink prompts you for a comment to store in the model. You would typically use the comment to document any changes you made to the model in the current session. Simulink stores the previous value of this parameter in the model's change history. See “Creating a Model Change History” on page 3–76 for more information.

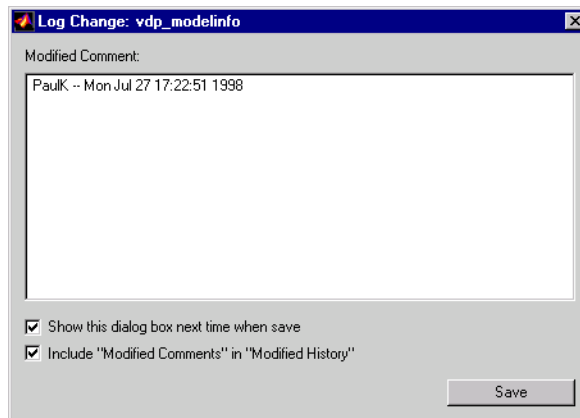
**Modified history.** History of modifications of this model. Simulink compiles the history from comments entered by users when they update the model. You can edit the history at any time by selecting the adjacent **Edit** button.

### Creating a Model Change History

Simulink allows you to create and store a record of changes to a model in the model itself. Simulink compiles the history automatically from comments that you or other users enter when they save changes to a model.

#### Logging Changes

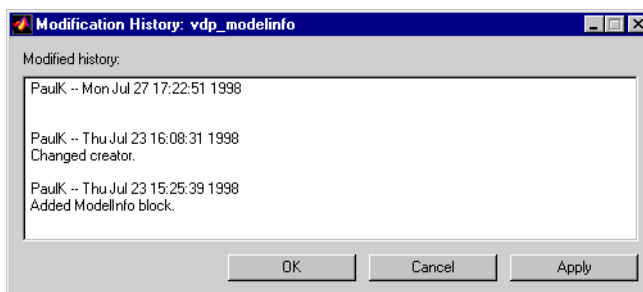
To start a change history, select the “Prompt for Comments When Save” option from the **History** pane on the Simulink **Model Properties** dialog box. The next time you save the model, Simulink displays a **Log Change** dialog box.



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

## Editing the Change History

To edit the change history for a model, click the **Edit** button on the Simulink **Model Properties** dialog box. Simulink displays the model's history in a **Modification History** dialog box.



Edit the history displayed in the dialog and select **Apply** or **OK** to save the changes.

## Version Control Properties

Simulink stores version control information in a model as model parameters. 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
ModifiedBy	Person who last modified this model
ModifiedByFormat	Format of the ModifiedBy parameter. Value can be a string. The string can include the tag <code>%&lt;Auto&gt;</code> . Simulink replaces the tag with the current value of the USER environment variable.

<b>Property</b>	<b>Description</b>
ModifiedDate	Date modified
ModifiedDateFormat	Format of the ModifiedDate parameter. Value can be any string. The string can include the tag %<Auto>. Simulink replaces the tag with the current date and time when saving the model.
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. Simulink replaces the tag with # when displaying the version number. It increments # when saving the model.
Description	Description of model
LastModificationDate	Date last modified.

## Ending a Simulink Session

Terminate a Simulink session by closing all Simulink windows.

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

- On a Microsoft Windows system: **Exit MATLAB**
- On a UNIX system: **Quit MATLAB**



# Running a Simulation

---

<b>Introduction</b> . . . . .	4-2
Using Menu Commands . . . . .	4-2
Running a Simulation from the Command Line . . . . .	4-3
<b>Running a Simulation Using Menu Commands</b> . . . . .	4-4
Setting Simulation Parameters and Choosing the Solver . . . . .	4-4
Applying the Simulation Parameters . . . . .	4-4
Starting the Simulation . . . . .	4-4
Simulation Diagnostics Dialog Box . . . . .	4-6
<b>The Simulation Parameters Dialog Box</b> . . . . .	4-8
The Solver Page . . . . .	4-8
The Workspace I/O Page . . . . .	4-17
The Diagnostics Page . . . . .	4-24
<b>Improving Simulation Performance and Accuracy</b> . . . . .	4-27
Speeding Up the Simulation . . . . .	4-27
Improving Simulation Accuracy . . . . .	4-28
<b>Running a Simulation from the Command Line</b> . . . . .	4-29
Using the sim Command . . . . .	4-29
Using the set_param Command . . . . .	4-29

# Introduction

You can run a simulation either by using Simulink menu commands or by entering commands in the MATLAB command window.

Many users use menu commands while they develop and refine their models, then enter commands in the MATLAB command window to run the simulation in “batch” mode.

## Using Menu Commands

Running a simulation using menu commands is easy and interactive. These commands let you select an ordinary differential equation (ODE) solver and define simulation parameters without having to remember command syntax. An important advantage is that you can perform certain operations interactively while a simulation is running:

- You can modify many simulation parameters, including the stop time, the solver, and the maximum step size.
- You can change the solver.
- You can simulate another system at the same time.
- You can click on a line to see the signal carried on that line on a floating (unconnected) Scope or Display block.
- You can modify the parameters of a block, as long as you do not cause a change in:
  - The number of states, inputs, or outputs
  - The sample time
  - The number of zero crossings
  - The vector length of any block parameters
  - The length of the internal block work vectors

You cannot make changes to the structure of the model, such as adding or deleting lines or blocks, during a simulation. If you need to make these kinds of changes, you need to stop the simulation, make the change, then start the simulation again to see the results of the change.



## **Running a Simulation from the Command Line**

Running a simulation from the command line has these advantages over running a simulation using menu commands:

- You can simulate M-file and MEX-file models, as well as Simulink block diagram models.
- You can run a simulation from an M-file, allowing simulation and block parameters to be changed iteratively.

For more information, see “Running a Simulation from the Command Line” on page 4-29.

# Running a Simulation Using Menu Commands

This section discusses how to use Simulink menu commands and the **Simulation Parameters** dialog box to run a simulation.

## Setting Simulation Parameters and Choosing the Solver

You set the simulation parameters and select the solver by choosing **Parameters** from the **Simulation** menu. Simulink displays the **Simulation Parameters** dialog box, which uses three “pages” to manage simulation parameters:

- The **Solver** page allows you to set the start and stop times, choose the solver and specify solver parameters, and choose some output options.
- The **Workspace I/O** page manages input from and output to the MATLAB workspace.
- The **Diagnostics** page allows you to select the level of warning messages displayed during a simulation.

Each page of the dialog box, including the parameters you set on the page, is discussed in detail in “The Simulation Parameters Dialog Box” on page 4-8.

You can specify parameters as valid MATLAB expressions, consisting of constants, workspace variable names, MATLAB functions, and mathematical operators.

## Applying the Simulation Parameters

After you have set the simulation parameters and selected the solver, you are ready to apply them to your model. Press the **Apply** button on the bottom of the dialog box to apply the parameters to the model. To apply the parameters and close the dialog box, press the **Close** button.

## Starting the Simulation

After you have applied the solver and simulation parameters to your model, you are ready to run the simulation. Select **Start** from the **Simulation** menu to run the simulation. You can also use the keyboard shortcut, **Ctrl-T**. When you select **Start**, the menu item changes to **Stop**.

Your computer beeps to signal the completion of the simulation.

---

**Note** A common mistake that new Simulink users make is to start a simulation while the Simulink block library is the active window. Make sure your model window is the active window before starting a simulation.

---

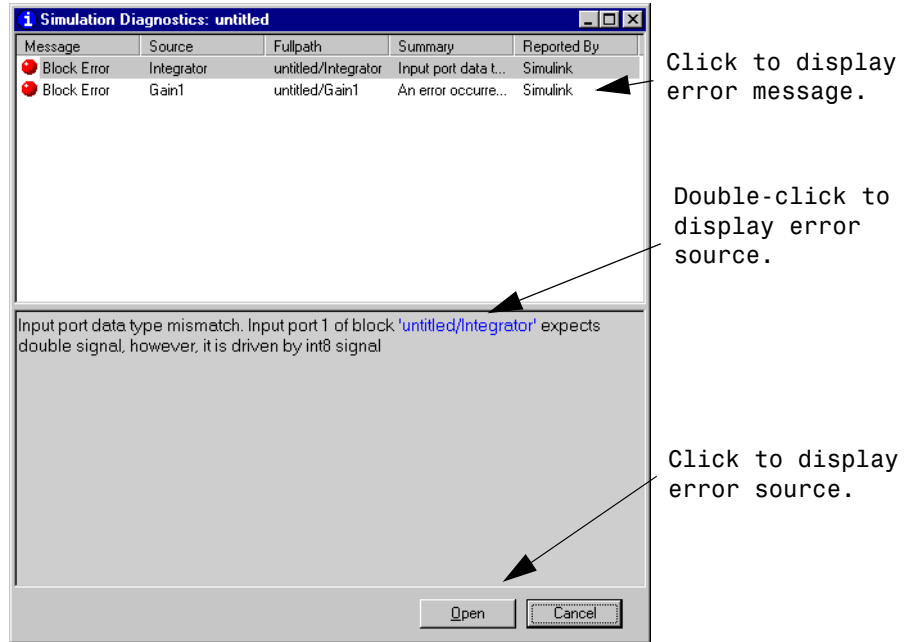
To stop a simulation, choose **Stop** from the **Simulation** menu. The keyboard shortcut for stopping a simulation is **Ctrl-T**, the same as for starting a simulation.

You can suspend a running simulation by choosing **Pause** from the **Simulation** menu. When you select **Pause**, the menu item changes to **Continue**. You proceed with a suspended simulation by choosing **Continue**.

If the model includes any blocks that write output to a file or to the workspace, or if you select output options on the **Simulation Parameters** dialog box, Simulink writes the data when the simulation is terminated or suspended.

## Simulation Diagnostics Dialog Box

If errors occur during a simulation, Simulink halts the simulation and displays the errors in the **Simulation Diagnostics** dialog box.



The dialog box has two panes. The upper pane consist of columns that display the following information for each error.

**Message.** Message type (for example, block error, warning, log)

**Source.** Name of the model element (for example, a block) that caused the error.

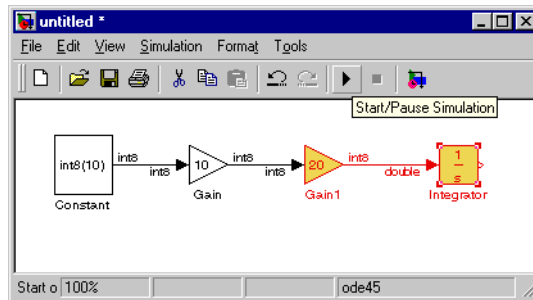
**Fullpath.** Path of the element that caused the error.

**Summary.** Error message abbreviated to fit in the column.

**Reported by.** Component that reported the error (for example, Simulink, Stateflow, Real-Time Workshop, etc).

The lower pane initially contains the full content of the first error message listed in the top pane. You can display the content of other messages by single-clicking on their entries in the upper pane.

In addition to displaying the **Simulation Diagnostics** dialog box, Simulink also opens (if necessary) the diagram that contains the error source and highlights the source.



You can similarly display other error sources by double-clicking on the corresponding error message in the top pane, by double-clicking on the name of the error source in the error message (highlighted in blue), or by selecting the **Open** button on the dialog box.

## The Simulation Parameters Dialog Box

This section discusses the simulation parameters, which you specify either on the **Simulation Parameters** dialog box or using the `sim` (see `sim` on page 4-30) and `simset` (see `simset` on page 4-32) commands. Parameters are described as they appear on the dialog box pages.

This table summarizes the actions performed by the dialog box buttons, which appear on the bottom of each dialog box page.

**Table 4-1: Simulation Parameters Dialog Box Buttons**

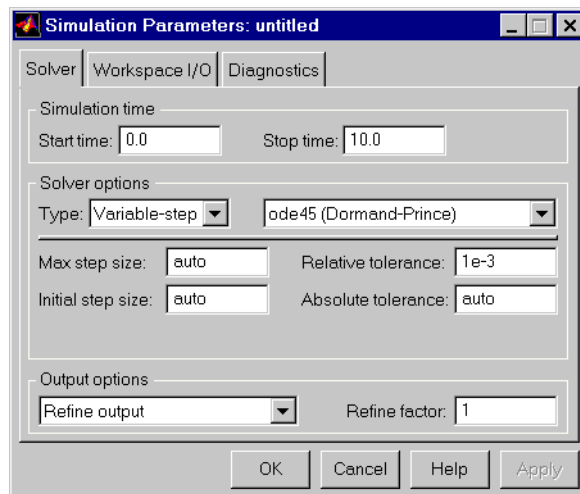
<b>Button</b>	<b>Action</b>
<b>Ok</b>	Applies the parameter values and closes the dialog box. During a simulation, the parameter values are applied immediately.
<b>Cancel</b>	Changes the parameter values back to the values they had when the dialog box was most recently opened and closes the dialog box.
<b>Help</b>	Displays help text for the dialog box page.
<b>Apply</b>	Applies the current parameter values and keeps the dialog box open. During a simulation, the parameter values are applied immediately.

## The Solver Page

The **Solver** page appears when you first choose **Parameters** from the **Simulation** menu or when you select the **Solver** tab.

The **Solver** page allows you to:

- Set the simulation start and stop times
- Choose the solver and specify its parameters
- Select output options



## Simulation Time

You can change the start time and stop time for the simulation by entering new values in the **Start time** and **Stop time** fields. The default start time is 0.0 seconds and the default stop time is 10.0 seconds.

Simulation time and actual clock time are not the same. For example, running a simulation for 10 seconds will usually not take 10 seconds. The amount of time it takes to run a simulation depends on many factors, including the model's complexity, the solver's step sizes, and the computer's clock speed.

## Solvers

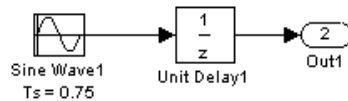
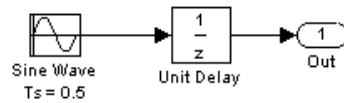
Simulation of Simulink models involves the numerical integration of sets of ordinary differential equations (ODEs). Simulink provides a number of solvers for the simulation of such equations. Because of the diversity of dynamic system behavior, some solvers may be more efficient than others at solving a particular problem. To obtain accurate and fast results, take care when choosing the solver and setting parameters.

You can choose between variable-step and fixed-step solvers. *Variable-step solvers* can modify their step sizes during the simulation. They provide error control and zero crossing detection. *Fixed-step solvers* take the same step size

during the simulation. They provide no error control and do not locate zero crossings. For a thorough discussion of solvers, see *Using MATLAB*.

**Default solvers.** If you do not choose a solver, Simulink chooses one based on whether your model has states:

- If the model has continuous states, ode45 is used. ode45 is an excellent general purpose solver. However, if you know that your system is stiff and if ode45 is not providing acceptable results, try ode15s. For a definition of stiff, see the note at the end of the section “Variable-step solvers” on page 4-11.
- If the model has no continuous states, Simulink uses the variable-step solver called discrete and displays a message indicating that it is not using ode45. Simulink also provides a fixed-step solver called discrete. This model shows the difference between the two discrete solvers.



With sample times of 0.5 and 0.75, the *fundamental sample time* for the model is 0.25 seconds. The difference between the variable-step and the fixed-step discrete solvers is the time vector that each generates.

The fixed-step discrete solver generates this time vector:

[0.0 0.25 0.5 0.75 1.0 1.25 ...]

The variable-step discrete solver generates this time vector:

[0.0 0.5 0.75 1.0 1.5 2.0 2.25 ...]

The step size of the fixed-step discrete solver is the fundamental sample time. The variable-step discrete solver takes the largest possible steps.



**Variable-step solvers.** You can choose these variable-step solvers: ode45, ode23, ode113, ode15s, ode23s, and discrete. The default is ode45 for systems with states, or discrete for systems with no states:

- 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.
- ode23 is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may 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 may be more efficient than ode45 at stringent tolerances. ode113 is a *multistep* solver; that is, it normally needs the solutions at several preceding time points to compute the current solution.
- ode15s is a variable order solver based on the numerical differentiation formulas (NDFs). These are related to but are more efficient than the backward differentiation formulas, BDFs (also known as Gear’s method). Like ode113, ode15s is a multistep method solver. If you suspect that a problem is stiff or if ode45 failed or was very inefficient, try ode15s.
- ode23s is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some 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 same iteration matrix is used in evaluating both stages. Like ode23s, this solver may be more efficient than ode15s at crude tolerances.
- discrete (variable-step) is the solver Simulink chooses when it detects that your model has no continuous states.

---

**Note** For a *stiff* problem, solutions can change on a time scale that is very short compared to the interval of integration, but 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. Jacobian matrices are generated numerically for `ode15s` and `ode23s`. For more information, see Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, 1994.

---

**Fixed-step solvers.** You can choose these fixed-step solvers: `ode5`, `ode4`, `ode3`, `ode2`, `ode1`, and `discrete`:

- `ode5` is the fixed-step version of `ode45`, the Dormand-Prince formula.
- `ode4` is RK4, the fourth-order Runge-Kutta formula.
- `ode3` is the fixed-step version of `ode23`, the Bogacki-Shampine formula.
- `ode2` is Heun's method, also known as the improved Euler formula.
- `ode1` is Euler's method.
- `discrete` (fixed-step) is a fixed-step solver that performs no integration. It is suitable for models having no states and for which zero crossing detection and error control are not important.

If you think your simulation may be providing unsatisfactory results, see “Improving Simulation Performance and Accuracy” on page 4-27.

### Solver Options

The default solver parameters provide accurate and efficient results for most problems. In some cases, however, tuning the parameters can improve performance. (For more information about tuning these parameters, see “Improving Simulation Performance and Accuracy” on page 4-27). You can tune the selected solver by changing parameter values on the **Solver** panel.

### Step Sizes

For variable-step solvers, you can set the maximum and suggested initial step size parameters. By default, these parameters are automatically determined, indicated by the value `auto`.

For fixed-step solvers, you can set the fixed step size. The default is also auto.

**Maximum step size.** The **Max step size** parameter controls the largest time step the solver can take. The default is determined from the start and stop times:

$$h_{max} = \frac{t_{stop} - t_{start}}{50}$$

Generally, the default maximum step size is sufficient. If you are concerned about the solver missing significant behavior, change the parameter to prevent the solver from taking too large a step. If the time span of the simulation is very long, the default step size may be too large for the solver to find the solution. Also, if your model contains periodic or nearly periodic behavior and you know the period, set the maximum step size to some fraction (such as 1/4) of that period.

In general, for more output points, change the refine factor, not the maximum step size. For more information, see “Refine output” on page 4-16.

**Initial step size.** By default, the solvers select an initial step size by examining the derivatives of the states at the start time. If the first step size is too large, the solver may step over important behavior. The initial step size parameter is a *suggested* first step size. The solver tries this step size but reduces it if error criteria are not satisfied.

## Error Tolerances

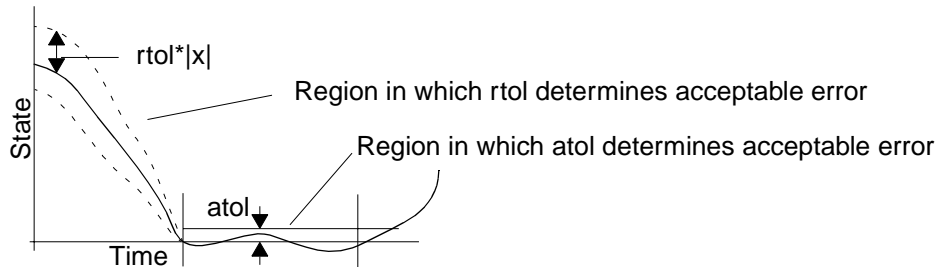
The solvers use standard local error control techniques to monitor the error at each time step. During each time step, the solvers compute the state values at the end of the step and also determine the *local error*, the estimated error of these state values. They then compare the local error to the *acceptable error*, which is a function of the relative tolerance (*rtol*) and absolute tolerance (*atol*). If the error is greater than the acceptable error for *any* state, the solver reduces the step size and tries again:

- *Relative tolerance* measures the error relative to the size of each state. The relative tolerance represents a percentage of the state’s value. The default, 1e-3, means that the computed state will be 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 error for the  $i$ th state,  $e_i$ , is required to satisfy.

$$e_i \leq \max(\text{rtol} \times |x_i|, \text{atol}_i)$$

The figure below shows a plot of a state and the regions in which the acceptable error is determined by the relative tolerance and the absolute tolerance:



If you specify auto (the default), Simulink sets the absolute tolerance for each state initially to  $1e-6$ . As the simulation progresses, Simulink resets the absolute tolerance for each state to the maximum value that the state has assumed thus far times the relative tolerance for that state. Thus, if a state goes from 0 to 1 and  $\text{reltol}$  is  $1e-3$ , then by the end of the simulation the  $\text{abstol}$  is set to  $1e-3$  also. If a state goes from 0 to 1000, then the  $\text{abstol}$  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. If the magnitudes of the states vary widely, it might be appropriate to specify different absolute tolerance values for different states. You can do this on the Integrator block's dialog box.

### The Maximum Order for ode15s

The `ode15s` solver is based on NDF formulas of order one through five. Although the higher order formulas are more accurate, they are less stable. If your model is stiff and requires more stability, reduce the maximum order to 2 (the highest order for which the NDF formula is A-stable). When you choose the `ode15s` solver, the dialog box displays this parameter.

As an alternative, you might try using the `ode23s` solver, which is a fixed-step, lower order (and A-stable) solver.

## Multitasking Options

If you select a fixed-step solver, the **Solver** page of the **Simulation Parameters** dialog box displays a **Mode** options list. The list allows you to select one of the following simulation modes.

**MultiTasking.** This mode issues an error if it detects an illegal sample rate transition between blocks, that is, a direct connection between blocks operating at different sample rates. In real-time multitasking systems, illegal sample rate transitions between tasks can result in a task's output not being available when needed by another task. By checking for such transitions, multitasking mode helps you to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks.

Use *rate transition* blocks to eliminate illegal rate transitions from your model. Simulink provides two such blocks: *Unit Delay* (see *Unit Delay* on page 8-214) and *Zero-Order Hold* (see *Zero-Order Hold* on page 8-221). To eliminate an illegal slow-to-fast transition, insert a *Unit Delay* block running at the slow rate between the slow output port and the fast input port. To eliminate an illegal fast-to-slow transition, insert a *Zero-Order Hold* block running at the slow rate between the fast output port and the slow input port. For more information, see Chapter 7, "Models with Multiple Sample Rates," in the *Real-Time Workshop Users Guide*.

**SingleTasking.** This mode does not check for sample rate transitions among blocks. This mode is useful when you are modeling a single-tasking system. In such systems, task synchronization is not an issue.

**Auto.** This option causes Simulink to use single-tasking mode if all blocks operate at the same rate and multitasking mode if the model contains blocks operating at different rates.

## Output Options

The **Output options** area of the dialog box enables you to control how much output the simulation generates. You can choose from three popup options:

- Refine output
- Produce additional output
- Produce specified output only

**Refine 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. Changing the refine factor does not change the steps used by the solver.

The refine factor applies to variable-step solvers and is most useful when using ode45. The ode45 solver is capable of taking large steps; when graphing simulation output, you may 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.

**Produce 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, Simulink displays an **Output Times** field on the **Solver** page. Enter a MATLAB expression in this field that evaluates to an additional time or a vector of additional times. The additional output is produced using a continuous extension formula at the additional times. Unlike the refine factor, this option changes the simulation step size so that time steps coincide with the times that you have specified for additional output.

**Produce specified output only.** The **Produce specified output only** choice provides simulation output *only* at the specified output times. This option changes the simulation step size so that time steps coincide with the times that you have specified for producing output. This choice is useful when comparing different simulations to ensure that the simulations produce output at the same times.

**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

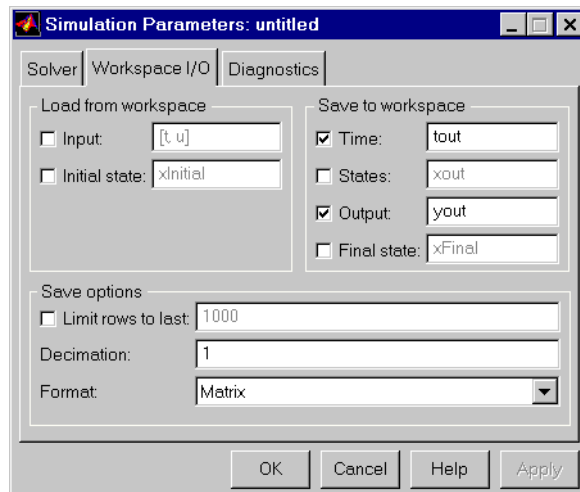
and perhaps at additional times, depending on the step-size chosen by the variable-step solver.

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

## The Workspace I/O Page

You can direct simulation output to workspace variables and get input and initial states from the workspace. On the **Simulation Parameters** dialog box, select the **Workspace I/O** tab. This page appears:



### Loading Input from the Base Workspace

Simulink can apply input from a model's base workspace to the model's top-level inports during a simulation run. To specify this option, check the **Input** box in the **Load from workspace** area of the **Workspace I/O** page. Then, enter an external input specification (see below) in the adjacent edit box and select **Apply**.

The external input can take any of the following forms.

**External Input Matrix.** The first column of an external input 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. Simulink linearly interpolates or extrapolates input values as necessary, if the **Interpolate data** option is selected for the corresponding inport (see “Interpolate data” on page 8-102).

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 inports. If you define  $t$  and  $u$  in the base workspace, you do not need to enter an external input specification for the model. This is because the default external input specification for a model is  $[t, u]$ .

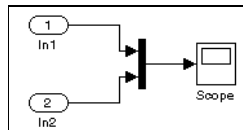
For example, suppose that a model has two inports, one of which accepts two signals and the other, one signal. Also, suppose that the base workspace defines  $u$  and  $t$  as follows:

```
t = (0:0.1:1)';  
u = [sin(t), cos(t), 4*cos(t)];
```

Then, to specify the external input for this model, simply check the model’s external input box.

**Structure with time.** Simulink can read data from the workspace in the form of a structure whose name is specified in the **Input** text field. 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 substructure has the field: **values**. The **values** field contains a column vector of inputs for the corresponding input port.

For example, consider the following model, which has two inputs.





Suppose that the base workspace defines a model input vector, `a`, as follows:

```
a.time = (0:0.1:1)';  
a.signals(1).values = sin(a.time);  
a.signals(2).values = cos(a.time);
```

Then, to specify `a` as the external input for this model, check the **Input** box and enter `a` in the adjacent text field.

---

**Note** Simulink can read back simulation data saved to the workspace in the **Structure with time** output format. See “Structure with time” on page 4-21 for more information.

---

**Structure.** The structure format is the same as the **Structure with time** format except that `time` field is empty. For example, in the preceding example, you could set the `time` field as follows.

```
a.time = []
```

In this case, Simulink reads the input for the first time step from the first element of an inport’s value array, the value for the second time step from the second element of the value array, etc.

---

**Note** Simulink can read back simulation data saved to the workspace in the **Structure** output format. See “Structure” on page 4-21 for more information.

---

**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 inport, and so on.

**External Input Time Expression.** 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 inports. 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'
```

Simulink evaluates the expression at each step of the simulation, applying the resulting values to the model's inports. Note that Simulink defines the variable  $t$  when it runs the simulation. Also, you can omit the time variable in expressions for functions of one variable. For example, Simulink interprets the expression  $\sin$  as  $\sin(t)$ .

### Saving Output to the Workspace

You can specify return variables by selecting the **Time**, **States**, and/or **Output** check boxes in the **Save to workspace** area of this dialog box page. Specifying return variables causes Simulink to write values for the time, state, and output trajectories (as many as are selected) into the workspace.

To assign values to different variables, specify those variable names in the field to the right of the check boxes. To write output to more than one variable, specify the variable names in a comma-separated list. Simulink saves the simulation times in a vector have the name specified in the **Save to Workspace** area.

---

**Note** Simulink saves the output to the 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 (see To Workspace on page 8-199).

---

The **Save options** area enables you to specify the format and restrict the amount of output saved.

Format options for model states and outputs are:

**Matrix.** Simulink saves the model states in a matrix that has the name specified in the **Save to Workspace** area (for example,  $xout$ ). Each column of the state matrix corresponds to a model state, each row to the states at a specific time. The model output matrix has the name specified in the **Save to Workspace**

area (for example, `yout`). Each column corresponds to a model output, each row to the outputs at a specific time.

**Structure with time.** Simulink saves the model's outputs in a structure having the name specified in the **Save to Workspace** area (for example, `yout`). The structure has two top-level fields: `time` and `signals`. The `time` field contains a vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to a model output. Each substructure has three fields: `values`, `label`, `blockName`. The `values` field contains a vector of outputs for the corresponding output. The `label` field specifies the label of the signal connected to the output. The `blockName` field specifies the name of the output. Simulink saves the model's states in a structure have the same organization as the model output structure.

**Structure.** This format is the same as the preceding except that Simulink 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 inport, and so on.

To set a limit on the number of rows of data saved, select the check box labeled **Limit rows to last** and specify the number of rows 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.

## Loading and Saving States

Initial conditions, which are applied to the system at the start of the simulation, are generally set in the blocks. You can override initial conditions set in the blocks by specifying them in the **States** area of this page.

You can also save the final states for a simulation and apply them to another simulation. This feature might be useful when you want to save a steady-state solution and restart the simulation at that known state. The states are saved in the format that you select in the Save options area of the Workspace I/O page. If you select Structure or Structure with Time, the saved format is as follows:

**Structure with time.** Simulink saves the model's states in a structure having the name specified in the **Final State** field of the **Save to Workspace** area (for example, `xFinal`). The structure has two top-level fields: `time` and `signals`. The `time` field contains a vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to a block that has states. Each substructure has three fields: `values`, `label`, `blockName`. The `values` field contains a vector of states for the corresponding block. The `label` field can be either `CState` (for continuous state) or `DState_n` where `n` can be 1, 2, 3 ... to the maximum number of sets of discrete states for the corresponding block. The `blockName` field specifies the name of the block represented by this structure element.

**Structure.** This format is the same as the preceding except that Simulink does not store simulation times in the `time` field of the saved structure

You load states by selecting the **Initial State** check box and specifying the name of a variable that contains the initial state values. This variable can be a matrix or a structure of the same form as is used to save final states. This allows Simulink to set the initial states for the current session to the final states saved in previous session, using the **Structure** or **Structure with time** format.

If the check box is not selected or the state vector is empty (`[]`), Simulink uses the initial conditions defined in the blocks.

You save the final states (the values of the states at the termination of the simulation) by selecting the **Final State** check box and entering a variable in the adjacent edit field.

**When the Model Has Multiple States.** If you want to specify the initial conditions for a model that has multiple states, you need to determine the order of the states. You can determine a model's initial conditions and the ordering of its states with this command

```
[sizes, x0, xstord] = sys([], [], [], 0)
```

where `sys` is the model name. The command returns:

- `sizes`, a vector that indicates certain model characteristics. Only the first two elements apply to initial conditions: `sizes(1)` is the number of continuous states, and `sizes(2)` is the number of discrete states. The `sizes`

vector is described in more detail in the companion volume to this guide, *Writing S-Functions*.

- `x0`, the block initial conditions.
- `xstord`, a string matrix that contains the full path name of all blocks in the model that have states. The order of the blocks in the `xstord` and `x0` vectors are the same.

For example, this statement obtains the values of the initial conditions and the ordering of the states for the `vdp` model (the example shows only the values for `sizes(1)`, the number of continuous states, and `sizes(2)`, the number of discrete states):

```
[sizes, x0, xstord] = vdp([], [], [], 0)
```

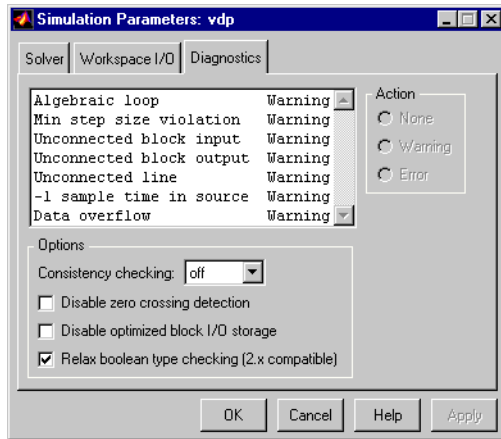
```
sizes =  
    2  
    0
```

```
x0 =  
    2  
    0
```

```
xstord =  
    'vdp/Integrator1'  
    'vdp/Integrator2'
```

## The Diagnostics Page

You can indicate the desired action for many types of events or conditions that can be encountered during a simulation by selecting the **Diagnostics** tab on the **Simulation Parameters** dialog box. This dialog box appears:



For each event type, you can choose whether you want no message, a warning message, or an error message. A warning message does not terminate a simulation, but an error message does.

## Consistency Checking

Consistency checking is a debugging tool that validates certain assumptions made by Simulink's ODE solvers. Its main use is to make sure that S-functions adhere to the same rules as Simulink built-in blocks. Because consistency checking results in a significant decrease in performance (up to 40%), it should generally be set to off. Use consistency checking to validate your S-functions and to help you determine the cause of unexpected simulation results.

To perform efficient integration, Simulink saves (caches) certain values from one time step for use in the next time step. For example, the derivatives at the end of a time step can generally be reused at the start of the next time step. The solvers take advantage of this to avoid redundant derivative calculations.

Another purpose of consistency checking is to ensure that blocks produce constant output when called with a given value of  $t$  (time). This is important for the stiff solvers (ode23s and ode15s) because, while calculating the

Jacobian, the block's output functions may be called many times at the same value of  $t$ .

When consistency checking is enabled, Simulink recomputes the appropriate values and compares them to the cached values. If the values are not the same, a consistency error occurs. Simulink compares computed values for these quantities:

- Outputs
- Zero crossings
- Derivatives
- States

### **Disabling Zero Crossing Detection**

You can disable zero crossing detection for a simulation. For a model that has zero crossings, disabling the detection of zero crossings may speed up the simulation but might have an adverse effect on the accuracy of simulation results.

This option disables zero crossing detection for those blocks that have intrinsic zero crossing detection. It does not disable zero crossing detection for the Hit Crossing block.

### **Disable optimized I/O storage**

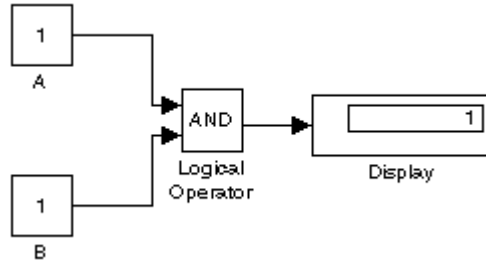
Checking this option causes Simulink to allocate a separate memory buffer for each block's I/O values instead of reusing memory buffers. This can substantially increase the amount of memory required to simulate large models. So you should select this option only when you need to debug a model. In particular, you should disable buffer reuse if you need to:

- Debug a C-MEX s-function
- Use a floating scope or display to inspect signals in a model that you are debugging

Simulink opens an error dialog if buffer reuse is enabled and you attempt to use a floating scope or display to display a signal whose buffer has been reused.

### Relax boolean type checking (2.x compatible)

Checking this option causes blocks that would otherwise require inputs of type boolean to accept inputs of type double. This ensures compatibility with models created by versions of Simulink earlier than Simulink 3. For example, consider the following model.



This model connects signals of type double to a Logical Operator block that ordinarily requires inputs of type boolean. Consequently, this model runs without error only if the **Relax boolean type checking** option is selected.



## Improving Simulation Performance and Accuracy

Simulation performance and accuracy can be affected by many things, including the model design and choice of simulation parameters.

The solvers handle most model simulations accurately and efficiently with their default parameter values. However, some models will yield better results if you adjust solver and simulation 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 Elementary Math 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 may take too many steps around the near-zero state values. See the discussion of error in “Error Tolerances” on page 4-13.
- The time scale may be too long. Reduce the time interval.
- The problem may be stiff but you're 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 9-7.
- 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.

### Improving Simulation Accuracy

To check your simulation accuracy, run the simulation over a reasonable time span. Then, reduce either the relative tolerance to  $1e-4$  (the default is  $1e-3$ ) or 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 may be unstable.
- If you are using `ode15s`, you may 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 will take 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 do 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.

## Running a Simulation from the Command Line

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 run a simulation from the command line using the `sim` command or the `set_param` command. Both are described below.

### Using the `sim` Command

The full syntax of the command that runs the simulation is:

```
[t,x,y] = sim(model, timespan, options, ut);
```

Only the `model` parameter is required. Parameters not supplied on the command are taken from the **Simulation Parameters** dialog box settings.

For detailed syntax for the `sim` command, see `sim` on page 4-30. The `options` parameter is a structure that supplies additional simulation parameters, including the solver name and error tolerances. You define parameters in the `options` structure using the `simset` command (see `simset` on page 4-32). The simulation parameters are discussed earlier in this chapter.

### Using the `set_param` Command

You can use the `set_param` command to start, stop, pause, or continue a simulation, or update a block diagram. Similarly, you can use the `get_param` command to check the status of a simulation. The format of the `set_param` command for this use is

```
set_param('sys', 'SimulationCommand', 'cmd')
```

where `'sys'` is the name of the system and `'cmd'` is `'start'`, `'stop'`, `'pause'`, `'continue'`, or `'update'`.

The format of the `get_param` command for this use is:

```
get_param('sys', 'SimulationStatus')
```

Simulink returns `'stopped'`, `'initializing'`, `'running'`, `'paused'`, `'terminating'`, and `'external'` (used with Real-Time Workshop).

# sim

---

**Purpose** Simulate a dynamic system.

**Syntax** `[t,x,y] = sim(model,timespan,options,ut);`  
`[t,x,y1, y2, ..., yn] = sim(model,timespan,options,ut);`

**Description** The `sim` command simulates a dynamic system represented by a Simulink model.

You can supply a null (`[]`) matrix for any right-side argument except the first (the model name). The `sim` command uses default values for arguments specified as null matrices. You can set optional simulation parameters, using the `sim` command's options argument. Parameters set in this way override parameters specified by the model.

If you want to simulate a continuous system, you must specify the solver parameter, using `simset` (see `simset` on page 4-32). The solver defaults to `VariableStepDiscrete` for purely discrete models.

**Arguments**

<code>t</code>	Returns the simulation's time vector.
<code>x</code>	Returns the simulation's state matrix consisting of continuous states followed by discrete states.
<code>y</code>	Returns the simulation's output matrix. Each column contains the output of a root-level Outputport block, in port number order. If any Outputport block has a vector input, its output takes the appropriate number of columns.
<code>y1, ..., yn</code>	Each $y_i$ returns the output of the corresponding root-level Outputport block for a model that has $n$ such blocks.
<code>model</code>	Name of a block diagram.
<code>timespan</code>	Simulation start and stop time. Specify as one of these: <code>tFinal</code> to specify the stop time. The start time is 0. <code>[tStart tFinal]</code> to specify the start and stop times. <code>[tStart OutputTimes tFinal]</code> to specify the start and stop times and time points to be returned in <code>t</code> . Generally, <code>t</code> will include more time points. <code>OutputTimes</code> is equivalent to choosing <b>Produce additional output</b> on the dialog box.

options	Optional simulation parameters specified as a structure created by the <code>simset</code> command (see <code>simset</code> on page 4-32).
ut	Optional external inputs to top-level Inport blocks. <code>ut</code> can be a MATLAB function (expressed as a string) that specifies the input $u = UT(t)$ at each simulation time step, a table of input values versus time for all input ports, or a comma-separated list of tables, <code>ut1, ut2, ...</code> , each of which corresponds to a specific port. Tabular input for all ports may be in the form of a MATLAB matrix or a structure. Tabular input for individual ports must be in the form of a structure. See “Loading Input from the Base Workspace” on page 4-17f or a description of the matrix and structure input formats.

## Examples

This command simulates the Van der Pol equations, using the `vdp` model that comes with Simulink. The command uses all default parameters:

```
[t,x,y] = sim('vdp')
```

This command simulates the Van der Pol equations, using the parameter values associated with the `vdp` model, but defines a value for the `Refine` parameter:

```
[t,x,y] = sim('vdp', [], simset('Refine',2));
```

This command simulates the Van der Pol equations for 1,000 seconds, saving the last 100 rows of the return variables. The simulation outputs values for `t` and `y` only, but saves the final state vector in a variable called `xFinal`:

```
[t,x,y] = sim('vdp', 1000, simset('MaxRows', 100,  
    'OutputVariables', 'ty', 'FinalStateName', 'xFinal'));
```

## See Also

`simset`, `simget`

# simset

---

**Purpose** Create or edit simulation parameters and solver properties for the `sim` command.

**Syntax**

```
options = simset(property, value, ...);
options = simset(old_opstruct, property, value, ...);
options = simset(old_opstruct, new_opstruct);
simset
```

**Description** The `simset` command creates a structure called `options`, in which the named simulation parameters and solver properties have the specified values. All unspecified parameters and properties take their default values. It is only necessary to enter enough leading characters to uniquely identify the parameter or property. Case is ignored for parameters and properties.

`options = simset(property, value, ...)` sets the values of the named properties and stores the structure in `options`.

`options = simset(old_opstruct, property, value, ...)` modifies the named properties in `old_opstruct`, an existing structure.

`options = simset(old_opstruct, new_opstruct)` combines two existing options structures, `old_opstruct` and `new_opstruct`, into `options`. Any properties defined in `new_opstruct` overwrite the same properties defined in `old_opstruct`.

`simset` with no input arguments displays all property names and their possible values.

You cannot obtain or set values of these properties and parameters using the `get_param` and `set_param` commands.

**Parameters** `AbsTol` positive scalar {1e-6}

*Absolute error tolerance.* This scalar applies to all elements of the state vector. `AbsTol` applies only to the variable-step solvers.

`Decimation` positive integer {1}

*Decimation for output variables.* Decimation factor applied to the return variables `t`, `x`, and `y`. A decimation factor of 1 returns every data logging time point, a decimation factor of 2 returns every other data logging time point, etc.

`DstWorkspace`            base | {current} | parent

*Where to assign variables.* This property specifies the workspace in which to assign any variables defined as return variables or as output variables on the To Workspace block.

`FinalStateName`        string {''}

*Name of final states variable.* This property specifies the name of a variable into which Simulink saves the model's states at the end of the simulation.

`FixedStep`             positive scalar

*Fixed step size.* This property applies only to the fixed-step solvers. If the model contains discrete components, the default is the fundamental sample time; otherwise, the default is one-fiftieth of the simulation interval.

`InitialState`          vector {[]}

*Initial continuous and discrete states.* The initial state vector consists of the continuous states (if any) followed by the discrete states (if any). `InitialState` supersedes the initial states specified in the model. The default, an empty matrix, causes the initial state values specified in the model to be used.

`InitialStep`           positive scalar {auto}

*Suggested initial step size.* This property applies only to the variable-step solvers. The solvers try a step size of `InitialStep` first. By default, the solvers determine an initial step size automatically.

`MaxOrder`              1 | 2 | 3 | 4 | {5}

*Maximum order of ode15s.* This property applies only to ode15s.

`MaxRows`               nonnegative integer {0}

*Limit number of output rows.* This property limits the number of rows returned in t, x, and y to the last `MaxRows` data logging time points. If specified as 0, the default, no limit is imposed.

`MaxStep`                positive scalar {auto}

*Upper bound on the step size.* This property applies only to the variable-step solvers and defaults to one-fiftieth of the simulation interval.

OutputPoints            {specified} | all

*Determine output points.* When set to `specified`, the solver produces outputs `t`, `x`, and `y` only at the times specified in `timespan`. When set to `all`, `t`, `x`, and `y` also include the time steps taken by the solver.

OutputVariables        {txy} | tx | ty | xy | t | x | y

*Set output variables.* If `'t'`, `'x'`, or `'y'` is missing from the property string, the solver produces an empty matrix in the corresponding output `t`, `x`, or `y`.

Refine                    positive integer {1}

*Output refine factor.* This property increases the number of output points by the specified factor, producing smoother output. `Refine` applies only to the variable-step solvers. It is ignored if output times are specified.

RelTol                   positive scalar {1e-3}

*Relative error tolerance.* This property applies to all elements of the state vector. The estimated error in each integration step satisfies

$$e(i) \leq \max(\text{RelTol} * \text{abs}(x(i)), \text{AbsTol}(i))$$

This property applies only to the variable-step solvers and defaults to `1e-3`, which corresponds to accuracy within 0.1%.

Solver                    VariableStepDiscrete |  
ode45 | ode23 | ode113 | ode15s | ode23s |  
FixedStepDiscrete |  
ode5 | ode4 | ode3 | ode2 | ode1

*Method to advance time.* This property specifies which solver is used to advance time.

SrcWorkspace            {base} | current | parent

*Where to evaluate expressions.* This property specifies the workspace in which to evaluate MATLAB expressions defined in the model.

Trace                    'minstep', 'siminfo', 'compile' {''}

*Tracing facilities.* This property enables simulation tracing facilities (specify one or more as a comma-separated list):

- The `'minstep'` trace flag specifies that simulation will stop when the solution changes so abruptly that the variable-step solvers cannot take a



step and satisfy the error tolerances. By default, Simulink issues a warning message and continues the simulation.

- The 'siminfo' trace flag provides a short summary of the simulation parameters in effect at the start of simulation.
- The 'compile' trace flag displays the compilation phases of a block diagram model.

ZeroCross                    {on} | off

*Enable/disable location of zero crossings.* This property applies only to the variable-step solvers. If set to off, variable-step solvers will not detect zero crossings for blocks having intrinsic zero crossing detection. The solvers adjust their step sizes only to satisfy error tolerance.

## Examples

This command creates an options structure called `myopts` that defines values for the `MaxRows` and `Refine` parameters, using default values for other parameters:

```
myopts = simset('MaxRows', 100, 'Refine', 2);
```

This command simulates the `vdp` model for 10 seconds and uses the parameters defined in `myopts`:

```
[t,x,y] = sim('vdp', 10, myopts);
```

## See Also

`sim`, `simget`

# simget

---

**Purpose** Get options structure properties and parameters.

**Syntax**

```
struct = simget(model)
value = simget(model, property)
```

**Description** The `simget` command gets simulation parameter and solver property values for the specified Simulink model. If a parameter or property is defined using a variable name, `simget` returns the variable's value, not its name. If the variable does not exist in the workspace, Simulink issues an error message.

`struct = simget(model)` returns the current options structure for the specified Simulink model. The options structure is defined using the `sim` and `simset` commands.

`value = simget(model, property)` extracts the value of the named simulation parameter or solver property from the model.

`value = simget(OptionStructure, property)` extracts the value of the named simulation parameter or solver property from `OptionStructure`, returning an empty matrix if the value is not specified in the structure. `property` can be a cell array containing the list of parameter and property names of interest. If a cell array is used, the output is also a cell array.

You need to enter only as many leading characters of a property name as are necessary to uniquely identify it. Case is ignored for property names.

**Examples** This command retrieves the options structure for the `vdp` model:

```
options = simget('vdp');
```

This command retrieves the value of the `Refine` property for the `vdp` model:

```
refine = simget('vdp', 'Refine');
```

**See Also** `sim`, `simset`

# Analyzing Simulation Results

---

<b>Viewing Output Trajectories</b> . . . . .	5-2
Using the Scope Block . . . . .	5-2
Using Return Variables . . . . .	5-2
Using the To Workspace Block . . . . .	5-3
 <b>Linearization</b> . . . . .	 5-4
 <b>Equilibrium Point Determination</b> . . . . .	 5-7
 <b>linfun</b> . . . . .	 5-9
 <b>trim</b> . . . . .	 5-13

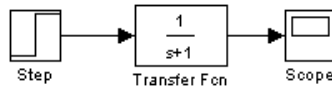
## Viewing Output Trajectories

Output trajectories from Simulink can be plotted using one of three methods:

- Feeding a signal into either a Scope or an XY Graph block
- Writing output to return variables and using MATLAB plotting commands
- Writing output to the workspace using To Workspace blocks and plotting the results using MATLAB plotting commands

### Using the Scope Block

You can use display output trajectories on a Scope block during a simulation. This simple model shows an example of the use of the Scope block.



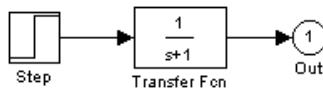
The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

The XY Graph block enables you to plot one signal against another.

These blocks are described in Chapter 8.

### Using Return Variables

By returning time and output histories, you can use MATLAB plotting commands to display and annotate the output trajectories.



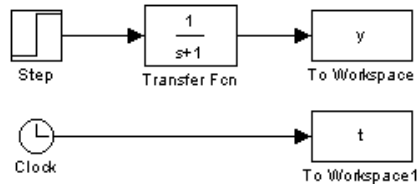
The block labeled Out is an Outport block from the Signals & Systems library. The output trajectory, `yout`, is returned by the integration solver. For more information, see Chapter 4.

You can also run this simulation from the **Simulation** menu by specifying variables for the time, output, and states on the **Workspace I/O** page of the **Simulation 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 MATLAB workspace. The model below illustrates this use.



The variables  $y$  and  $t$  appear in the workspace when the simulation is complete. The time vector is stored by feeding a Clock block into a To Workspace block. The time vector can also be acquired by entering a variable name for the time on the **Workspace I/O** page of the **Simulation Parameters** dialog box for menu-driven simulations, or by returning it using the `sim` command (see Chapter 4 for more information).

The To Workspace block can accept a vector input, with each input element's trajectory stored as a column vector in the resulting workspace variable.

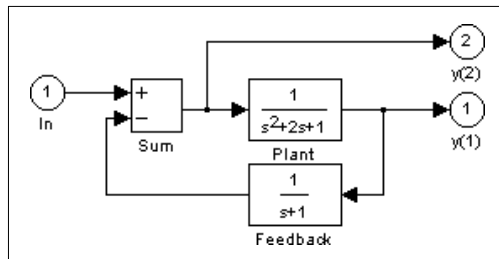
## Linearization

Simulink provides the `linmod` 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 Simulink 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 Signals & Systems 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 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)`

Other functions in the Control System Toolbox and Robust Control Toolbox can be used for linear control system design.

When the model is nonlinear, an operating point may be chosen at which to extract the linearized model. The nonlinear model is also sensitive to the perturbation sizes at which the model is extracted. These must be selected to balance the trade-off between truncation and roundoff error. Extra arguments to `linmod` specify the operating point and perturbation points:

```
[A,B,C,D] = linmod('sys', x, u, pert, xpert, upert)
```

For discrete systems or mixed continuous and discrete systems, use the function `dlinmod` for linearization. This 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. For more information, see `linfun` on page 5-9.

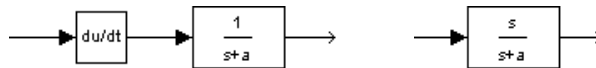
Using `linmod` to linearize a model that contains Derivative or Transport Delay blocks can be troublesome. Before linearizing, replace these blocks with specially designed blocks that avoid the problems. These blocks are in the Simulink Extras library in the Linearization sublibrary. You access the Extras 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.)

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

$$\frac{s}{s+a}$$

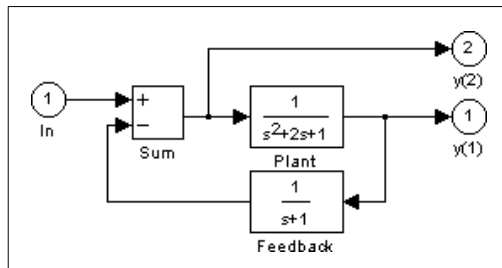
In this example, the blocks on the left of this figure can be replaced by the block on the right.





## Equilibrium Point Determination

The Simulink `trim` function determines steady-state equilibrium points. 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 may differ due to 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 may 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` on page 5-13.

**Purpose** Extract the linear state-space model of a system around an operating point.

**Syntax**

```
[A,B,C,D] = linfun('sys')
[A,B,C,D] = linfun('sys', x, u)
[A,B,C,D] = linfun('sys', x, u, pert)
[A,B,C,D] = linfun('sys', x, u, pert, xpert, upert)
```

**Arguments**

<i>linfun</i>	linmod, dlinmod, or linmod2.
sys	The name of the Simulink system from which the linear model is to be extracted.
x and u	The state and the input vectors. If specified, they set the operating point at which the linear model is to be extracted.
pert	Optional scalar perturbation factor used for both x and u. If not specified, a default value of 1e-5 is used.
xpert and upert	Optional vectors that explicitly set perturbation levels for individual states and inputs. If specified, the pert argument is ignored. The <i>i</i> th state x is perturbed to $x(i) + xpert(i)$ The <i>j</i> th input u is perturbed to $u(j) + upert(j)$

**Description** linmod obtains linear models from systems of ordinary differential equations described as Simulink models. linmod returns the linear model in state-space form,  $A, B, C, D$ , which describes the linearized input-output relationship:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

$[A,B,C,D] = \text{linmod}('sys')$  obtains the linearized model of sys around an operating point with the state variables x and the input u set to zero.

linmod perturbs the states around the operating point to determine the rate of change in the state derivatives and outputs (Jacobians). This result is used to calculate the state-space matrices. Each state  $x(i)$  is perturbed to

$$x(i) + \Delta(i)$$

where

$$\Delta(i) = \delta(1 + |x(i)|)$$

Likewise the  $j$ th input is perturbed to

$$u(j) + \Delta(j)$$

where

$$\Delta(j) = \delta(1 + |u(j)|)$$

## Discrete-Time System Linearization

The function `dlinmod` can linearize discrete, multirate, and hybrid continuous and discrete systems at any given sampling time. Use the same calling syntax for `dlinmod` as for `linmod`, but insert the sample time at which to perform the linearization as the second argument. For example

```
[Ad,Bd,Cd,Dd] = dlinmod('sys', Ts, x, u);
```

produces a discrete state-space model at the sampling time  $T_s$  and the operating point given by the state vector  $x$  and input vector  $u$ . To obtain a continuous model approximation of a discrete system, set  $T_s$  to 0.

For systems composed of linear, multirate, discrete, and continuous blocks, `dlinmod` produces linear models having identical frequency and time responses (for constant inputs) at the converted sampling time  $T_s$ , provided that:

- $T_s$  is an integer multiple of all the sampling times in the system.
- $T_s$  is not less than the slowest sample time in the system.
- The system is stable.

It is possible for valid linear models to be obtained when these conditions are not met.

Computing the eigenvalues of the linearized matrix  $A_d$  provides an indication of the stability of the system. The system is stable if  $T_s > 0$  and the eigenvalues are within the unit circle, as determined by this statement:

$$\text{all}(\text{abs}(\text{eig}(A_d))) < 1$$

Likewise, the system is stable if  $T_s = 0$  and the eigenvalues are in the left half plane, as determined by this statement:

```
all(real(eig(Ad))) < 0
```

When the system is unstable and the sample time is not an integer multiple of the other sampling times, `dlinmod` produces  $A_d$  and  $B_d$  matrices, which may be complex. The eigenvalues of the  $A_d$  matrix in this case still, however, provide a good indication of stability.

You can use `dlinmod` to convert the sample times of a system to other values or to convert a linear discrete system to a continuous system or vice versa.

The frequency response of a continuous or discrete system can be found by using the `bode` command.

### An Advanced Form of Linearization

The `linmod2` routine provides an advanced form of linearization. This routine takes longer to run than `linmod`, but may produce more accurate results.

The calling syntax for `linmod2` is similar to that used for `linmod`, but functions differently. For instance, `linmod2('sys', x, u)` produces a linear model as does `linmod`; however, the perturbation levels for each state-space matrix element are set individually to attempt to minimize roundoff and truncation errors.

`linmod2` tries to balance roundoff error (caused by small perturbation levels, which cause errors associated with finite precision mathematics) and truncation error (caused by large perturbation levels, which invalidate the piecewise linear approximation).

With the form `[A,B,C,D] = linmod2('sys', x, u, pert)`, the variable `pert` indicates the lowest level of perturbation that can be used; the default is  $1e-8$ . `linmod2` has the advantage that it can detect discontinuities and produce warning messages, such as the following:

```
Warning: discontinuity detected at A(2,3)
```

When such a warning occurs, try a different operating point at which to obtain the linear model.

With the form

```
[A,B,C,D] = linmod2('sys', x, u, pert, Apert, Bpert, Cpert, Dpert)
```

the variables `Apert`, `Bpert`, `Cpert`, and `Dpert` are matrices used to set the perturbation levels for each state and input combination; therefore, the  $ij$ th element of `Apert` is the perturbation level associated with obtaining the  $ij$ th element of the `A` matrix. Return default perturbation sizes with

```
[A,B,C,D,Apert,Bpert,Cpert,Dpert] = linmod2('sys', x, u);
```

## Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `pert` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

When the model being linearized is itself a linear model, the problem of truncation error no longer exists; therefore, you can set the perturbation levels to whatever value is desired. A relatively high value is generally preferable, since this tends to reduce roundoff error. The operating point used does not affect the linear model obtained.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a string variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose  $i$ th row is the block name associated with the  $i$ th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

Linearizing a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearization” on page 5–4.

**Purpose** Find a trim point of a dynamic system.

**Syntax**

```
[x,u,y,dx] = trim('sys')
[x,u,y,dx] = trim('sys',x0,u0,y0)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)
[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)
[x,u,y,dx,options] = trim('sys',...)
```

**Description** A trim point, also known as an equilibrium point, is a point in the parameter space of a dynamic system where the system is in a steady state. For example, a trim point of an aircraft is a setting of its controls that causes the aircraft to fly straight and level. Mathematically, a trim point is a point where the system's state derivatives equal zero. `trim` starts from an initial point and searches, using a sequential quadratic programming algorithm, until it finds the nearest trim point. You must supply the initial point implicitly or explicitly. If `trim` cannot find a trim point, it returns the point encountered in its search where the state derivatives are closest to zero in a min-max sense; that is, it returns the point that minimizes the maximum deviation from zero of the derivatives. `trim` can find trim points that meet specific input, output, or state conditions and points where a system is changing in a specified manner, that is, points where the system's state derivatives equal specific, nonzero values.

`[x,u,y] = trim('sys')` finds the equilibrium point nearest to the system's initial state `x0`. Specifically, `trim` finds the equilibrium point that minimizes the maximum absolute value of  $[x-x_0, u, y]$ . If `trim` cannot find an equilibrium point near the system's initial state, it returns the point where the system is nearest to equilibrium. Specifically, it returns the point that minimizes  $\text{abs}(dx-0)$ . You can obtain `x0` using this command:

```
[sizes,x0,xstr] = sys([],[],[],0)
```

`[x,u,y] = trim('sys',x0,u0,y0)` finds the trim point nearest to `x0`, `u0`, `y0`, that is, the point that minimizes the maximum value of  $\text{abs}([x-x_0; u-u_0; y-y_0])$ .

The command

```
trim('sys', x0, u0, y0, ix, iu, iy)
```

finds the trim point closest to  $x_0$ ,  $u_0$ ,  $y_0$  that satisfies a specified set of state, input, and/or output conditions. The integer vectors  $ix$ ,  $iu$ , and  $iy$  select the values in  $x_0$ ,  $u_0$ , and  $y_0$  that must be satisfied. If `trim` cannot find an equilibrium point that satisfies the specified set of conditions exactly, it returns the nearest point that satisfies the conditions, namely

$$\text{abs}([x(ix)-x_0(ix); u(iu)-u_0(iu); y(iy)-y_0(iy)])$$

Use the syntax

```
[x,u,y,dx] = trim('sys', x0, u0, y0, ix, iu, iy, dx0, idx)
```

to find specific non-equilibrium points, that is, points where the system's state derivatives have some specified, nonzero value. Here,  $dx_0$  specifies the state derivative values at the search's starting point and  $idx$  selects the values in  $dx_0$  that the search must satisfy exactly.

The optional `options` argument is an array of optimization parameters that `trim` passes to the optimization function that it uses to find trim points. The optimization function, in turn, uses this array to control the optimization process and to return information about the process. `trim` returns the options array at the end of the search process. By exposing the underlying optimization process in this way, `trim` allows you to monitor and fine-tune the search for trim points.

Five of the optimization array elements are particularly useful for finding trim points. The following table describes how each element affects the search for a trim point.

No.	Default	Description
1	0	Specifies display options. 0 specifies no display; 1 specifies tabular output; -1 suppresses warning messages.
2	0.0001	Precision the computed trim point must attain to terminate the search.
3	0.0001	Precision the trim search goal function must attain to terminate the search.



No.	Default	Description
4	0.0001	Precision the state derivatives must attain to terminate the search.
10	N/A	Returns the number of iterations used to find a trim point.

See the *Optimization Toolbox User's Guide* for a detailed description of the options array.

## Examples

Consider a linear state-space model

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

The  $A$ ,  $B$ ,  $C$ , and  $D$  matrices are as follows in a system called `sys`:

$$A = [-0.09 \quad -0.01; \quad 1 \quad 0];$$

$$B = [ \quad 0 \quad -7; \quad 0 \quad -2];$$

$$C = [ \quad 0 \quad 2; \quad 1 \quad -5];$$

$$D = [-3 \quad 0; \quad 1 \quad 0];$$

### Example 1

To find an equilibrium point, use

```
[x,u,y,dx,options] = trim('sys')
```

```
x =
```

```
0
```

```
0
```

```
u =
```

```
0
```

```
y =
```

```
0
```

```
0
```

```
dx =
```

```
0
```

```
0
```

The number of iterations taken is

```
options(10)
ans =
     7
```

## Example 2

To find an equilibrium point near  $x = [1;1]$ ,  $u = [1;1]$  enter

```
x0 = [1;1];
u0 = [1;1];
[x,u,y,dx,options] = trim('sys', x0, u0);

x =
    1.0e-11 *
   -0.1167
   -0.1167
u =
    0.3333
    0.0000
y =
   -1.0000
    0.3333
dx =
    1.0e-11 *
    0.4214
    0.0003
```

The number of iterations taken is

```
options(10)
ans =
    25
```

**Example 3**

To find an equilibrium point with the outputs fixed to 1, use

```

y = [1;1];
iy = [1;2];
[x,u,y,dx] = trim('sys', [], [], y, [], [], iy)

x =
    0.0009
   -0.3075
u =
   -0.5383
    0.0004
y =
    1.0000
    1.0000
dx =
    1.0e-16 *
   -0.0173
    0.2396

```

**Example 4**

To find an equilibrium point with the outputs fixed to 1 and the derivatives set to 0 and 1, use

```

y = [1;1];
iy = [1;2];
dx = [0;1];
idx = [1;2];
[x,u,y,dx,options] = trim('sys',[],[],y,[],[],iy,dx,idx)

x =
    0.9752
   -0.0827
u =
   -0.3884
   -0.0124
y =
    1.0000
    1.0000
dx =
    0.0000
    1.0000

```

# trim

---

The number of iterations taken is

```
options(10)
ans =
    13
```

## Limitations

The trim point found by `trim` starting from any given initial point is only a local value. Other, more suitable trim points may exist. Thus, if you want to find the most suitable trim point for a particular application, it is important to try a number of initial guesses for  $x$ ,  $u$ , and  $y$ .

## Algorithm

`trim` uses a sequential quadratic programming algorithm to find trim points. See the *Optimization Toolbox User's Guide* for a description of this algorithm.

# Using Masks to Customize Blocks

---

<b>Introduction</b> . . . . .	6-2
<b>A Sample Masked Subsystem</b> . . . . .	6-3
Creating Mask Dialog Box Prompts . . . . .	6-4
Creating the Block Description and Help Text . . . . .	6-6
Creating the Block Icon . . . . .	6-6
Summary . . . . .	6-8
<b>The Mask Editor: An Overview</b> . . . . .	6-9
<b>The Initialization Page</b> . . . . .	6-10
Prompts and Associated Variables . . . . .	6-10
Control Types . . . . .	6-12
Default Values for Masked Block Parameters . . . . .	6-14
Tunable Parameters . . . . .	6-14
Initialization Commands . . . . .	6-15
<b>The Icon Page</b> . . . . .	6-18
Displaying Text on the Block Icon . . . . .	6-18
Displaying Graphics on the Block Icon . . . . .	6-20
Displaying Images on Masks . . . . .	6-21
Displaying a Transfer Function on the Block Icon. . . . .	6-22
Controlling Icon Properties . . . . .	6-23
<b>The Documentation Page</b> . . . . .	6-26
The Mask Type Field . . . . .	6-26
The Block Description Field . . . . .	6-26
The Mask Help Text Field . . . . .	6-27
<b>Creating Dynamic Masked Dialogs</b> . . . . .	6-28
Setting Masked Dialog Parameters . . . . .	6-28
Predefined Masked Dialog Parameters . . . . .	6-29

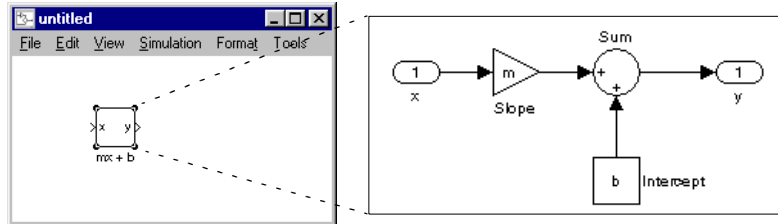
# Introduction

Masking is a powerful Simulink feature that enables you to customize the dialog box and icon for a subsystem. With masking, you can:

- Simplify the use of your model by replacing many dialog boxes in a subsystem with a single one. Instead of requiring the user of the model to open each block and enter parameter values, those parameter values can be entered on the mask dialog box and passed to the blocks in the masked subsystem.
- Provide a more descriptive and helpful user interface by defining a dialog box with your own block description, parameter field labels, and help text.
- Define commands that compute variables whose values depend on block parameters.
- Create a block icon that depicts the subsystem's purpose.
- Prevent unintended modification of subsystems by hiding their contents behind a customized interface.
- Create dynamic dialogs.

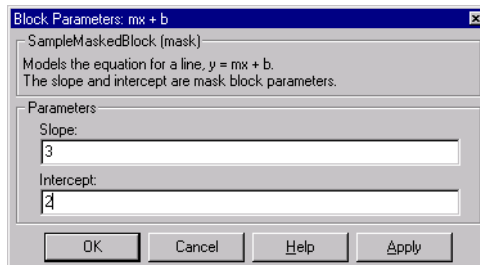
## A Sample Masked Subsystem

This simple subsystem models the equation for a line,  $y = mx + b$ .



Ordinarily, when you double-click on a Subsystem block, the Subsystem block opens, displaying its blocks in a separate window. The  $mx + b$  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.

This example creates a custom dialog box and icon for the subsystem. One dialog box contains prompts for both the slope and the intercept. After you create the mask, double-click on the Subsystem block to open the mask dialog box. The mask dialog box and icon look like this.



The mask dialog box

The block icon



A user enters values for **Slope** and **Intercept** into the mask dialog box. Simulink makes these values available to all the blocks in the underlying subsystem. Masking this subsystem creates a self-contained functional unit with its own application-specific parameters, Slope and Intercept. The mask maps these *mask parameters* to the generic parameters of the underlying blocks. The complexity of the subsystem is encapsulated by a new interface that has the look and feel of a built-in Simulink block.

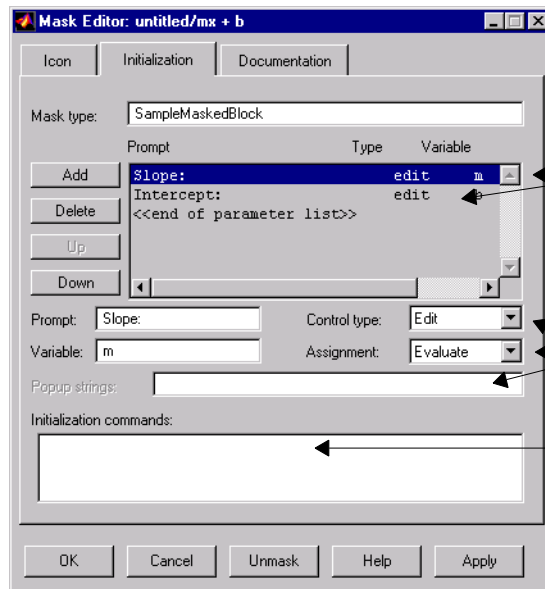
To create a mask for this subsystem, you need to:

- Specify the prompts for the mask dialog box parameters. In this example, the mask dialog box has prompts for the slope and intercept.
- Specify the variable name used to store the value of each parameter.
- Enter the documentation of the block, consisting of the block description and the block help text.
- Specify the drawing command that creates the block icon.
- Specify the commands that provide the variables needed by the drawing command (there are none in this example).

## Creating Mask Dialog Box Prompts

To create the mask for this subsystem, select the Subsystem block and choose **Mask Subsystem** from the **Edit** menu.

The mask dialog box shown at the beginning of this section is created largely on the **Initialization** page of the Mask Editor. For this sample model, the page looks like this.



Parameter fields: prompts, types, and variables that hold the values entered by the user.

Where you enter and edit the parameter field characteristics.

The commands that define variables used by the icon drawing command or by blocks in the masked subsystem.

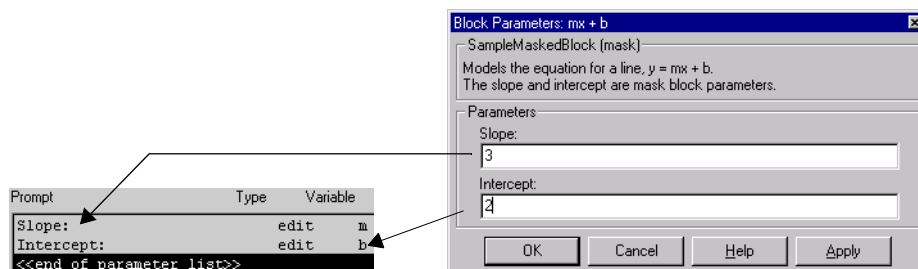


The Mask Editor enables you to specify these attributes of a mask parameter:

- The prompt – the text label that describes the parameter.
- The control type – the style of user interface control that determines how parameter values are entered or selected.
- The variable – the name of the variable that will store the parameter value.

Generally, it is convenient to refer to masked parameters by their prompts. In this example, the parameter associated with slope is referred to as the Slope parameter, and the parameter associated with intercept is referred to as the Intercept parameter.

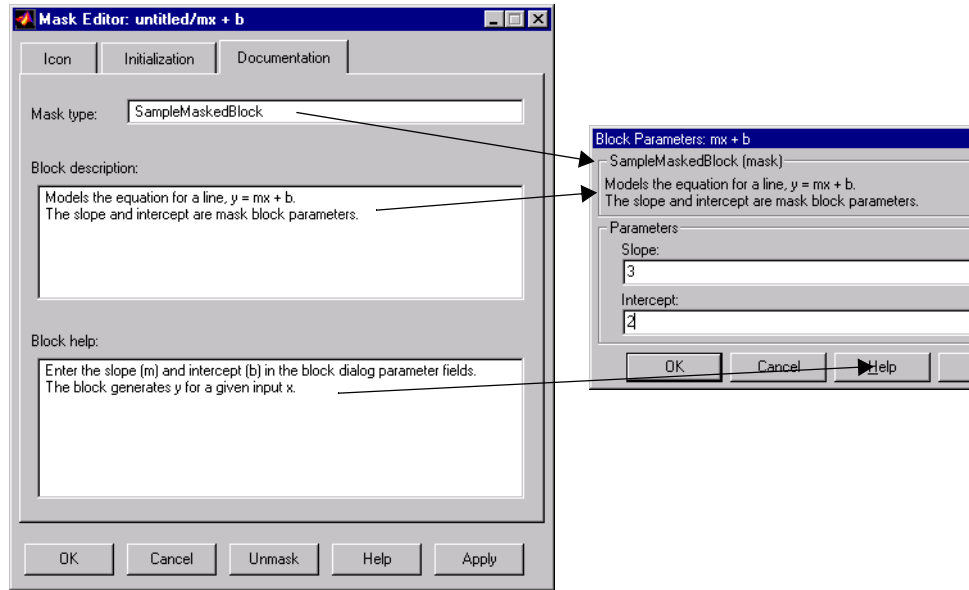
The slope and intercept are defined as edit controls. This means that the user types values into edit fields in the mask dialog box. These values are stored in variables in the *mask workspace* (see “The Mask Workspace” on page 6-15). Masked blocks can access variables only in the mask workspace. In this example, the value entered for the slope is assigned to the variable *m*. The Slope block in the masked subsystem gets the value for the slope parameter from the mask workspace. This figure shows how the slope parameter definitions in the Mask Editor map to the actual mask dialog box parameters.



After you have created the mask parameters for slope and intercept, press the **OK** button. Then, double-click on the Subsystem block to open the newly constructed dialog box. Enter 3 for the **Slope** and 2 for the **Intercept** parameters.

## Creating the Block Description and Help Text

The mask type, block description, and help text are defined on the **Documentation** page. For this sample masked block, the page looks like this.

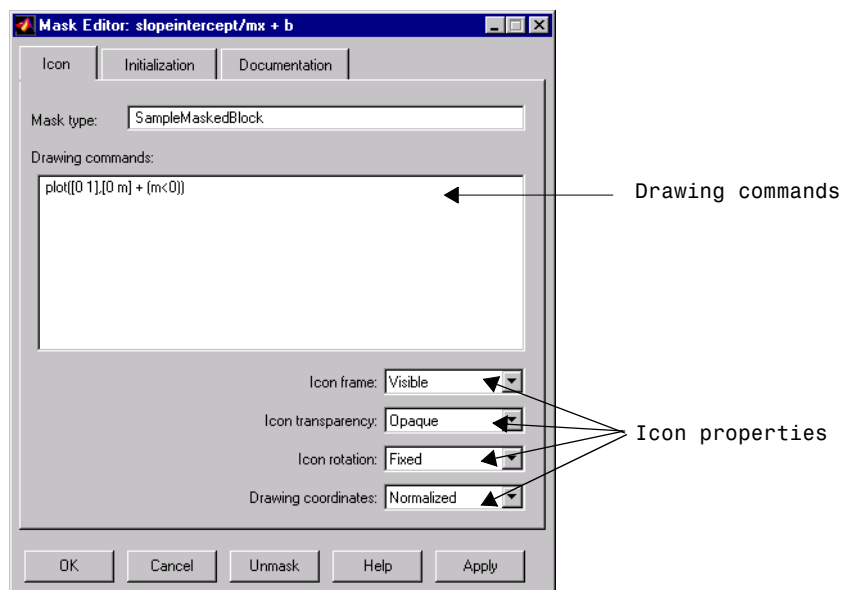


## Creating the Block Icon

So far, we have created a customized dialog box for the  $mx + b$  subsystem. However, the Subsystem block still displays the generic Simulink subsystem icon. An appropriate icon for this masked block is a plot that indicates the slope of the line. For a slope of 3, that icon looks like this.



The block icon is defined on the **Icon** page. For this block, the **Icon** page looks like this.



The drawing command plots a line from  $(0, 0)$  to  $(0, m)$ . If the slope is negative, Simulink shifts the line up by 1 to keep it within the visible drawing area of the block.

The drawing commands have access to all of the variables in the mask workspace. As you enter different values of slope, the icon updates the slope of the plotted line.

Select **Normalized** as the **Drawing coordinates** parameter, located at the bottom of the list of icon properties, to specify that the icon be drawn in a frame whose bottom-left corner is  $(0, 0)$  and whose top-right corner is  $(1, 1)$ . This parameter is described later in this chapter.

### **Summary**

This discussion of the steps involved in creating a sample mask introduced you to these tasks:

- Defining dialog box prompts and their characteristics
- Defining the masked block description and help text
- Defining the command that creates the masked block icon

The remainder of this chapter discusses the Mask Editor in more detail.

## The Mask Editor: An Overview

To mask a subsystem (you can only mask Subsystem blocks), select the Subsystem block, then choose **Mask Subsystem** from the **Edit** menu. The Mask Editor appears. The Mask Editor consists of three pages, each handling a different aspect of the mask:

- The **Initialization** page enables you to define and describe mask dialog box parameter prompts, name the variables associated with the parameters, and specify initialization commands.
- The **Icon** page enables you to define the block icon.
- The **Documentation** page enables you to define the mask type and specify the block description and the block help.

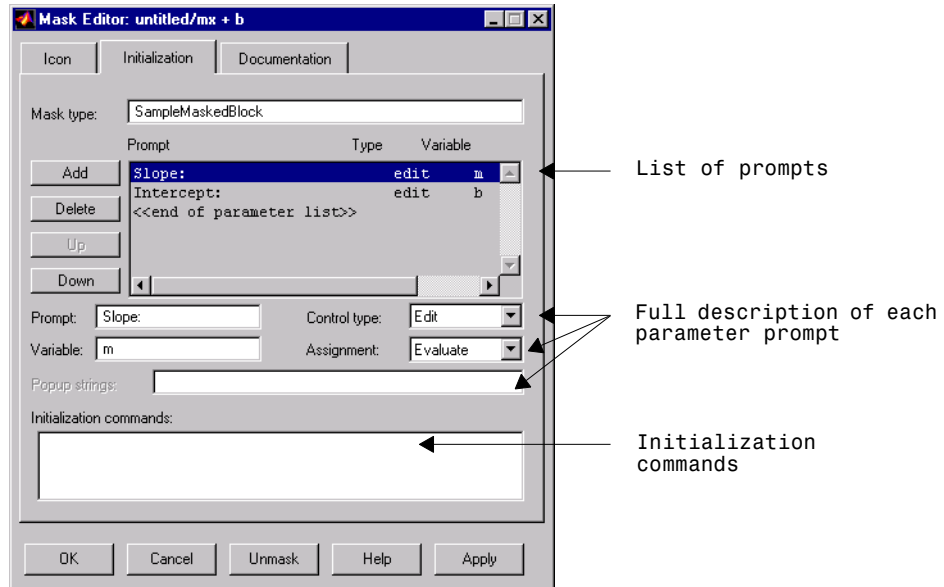
Five buttons appear along the bottom of the Mask Editor:

- The **Ok** button applies the mask settings on all pages and closes the Mask Editor.
- The **Cancel** button closes the Mask Editor without applying any changes made since you last pressed the **Apply** button.
- The **Unmask** button deactivates the mask and closes the Mask Editor. The mask information is retained so that the mask can be reactivated. To reactivate the mask, select the block and choose **Create Mask**. The Mask Editor opens, displaying the previous settings. The inactive mask information is discarded when the model is closed and cannot be recovered.
- The **Help** button displays the contents of this chapter.
- The **Apply** button creates or changes the mask using the information that appears on all masking pages. The Mask Editor remains open.

To see the system under the mask without unmasking it, select the Subsystem block, then choose **Look Under Mask** from the **Edit** menu. This command opens the subsystem. The block's mask is not affected.

## The Initialization Page

The mask interface enables a user of a masked system to enter parameter values for blocks within the masked system. You create the mask interface by defining prompts for parameter values on the **Initialization** page. The **Initialization** page for the  $mx+b$  sample masked system looks like this.



### Prompts and Associated Variables

A *prompt* provides information that helps the user enter or select a value for a block parameter. Prompts appear on the mask dialog box in the order they appear in the **Prompt** list.

When you define a prompt, you also specify the variable that is to store the parameter value, choose the style of control for the prompt, and indicate how the value is to be stored in the variable.

If the **Assignment** type is **Evaluate**, the value entered by the user is evaluated by MATLAB before it is assigned to the variable. If the type is **Literal**, the value entered by the user is not evaluated, but is assigned to the variable as a string.

For example, if the user enters the string `gain` in an edit field and the **Assignment** type is **Evaluate**, the string `gain` is evaluated by MATLAB and the result is assigned to the variable. If the type is **Literal**, the string is not evaluated by MATLAB so the variable contains the string `'gain'`.

If you need both the string entered as well as the evaluated value, choose **Literal**. Then use the MATLAB `eval` command in the initialization commands. For example, if `LitVal` is the string `'gain'`, then to obtain the evaluated value, use the command

```
value = eval(LitVal)
```

In general, most parameters use an **Assignment** type of **Evaluate**.

### Creating the First Prompt

To create the first prompt in the list, enter the prompt in the **Prompt** field, the variable that is to contain the parameter value in the **Variable** field, and choose a control style and an assignment type.

### Inserting a Prompt

To insert a prompt in the list:

- 1 Select the prompt that appears immediately *below* where you want to insert the new prompt and click on the **Add** button to the left of the prompt list.
- 2 Enter the text for the prompt in the **Prompt** field. Enter the variable that is to hold the parameter value in the **Variable** field.

### Editing a Prompt

To edit an existing prompt:

- 1 Select the prompt in the list. The prompt, variable name, control style, and assignment type appear in the fields below the list.
- 2 Edit the appropriate value. When you click the mouse outside the field or press the **Enter** or **Return** key, Simulink updates the prompt.

### Deleting a Prompt

To delete a prompt from the list:

- 1 Select the prompt you want to delete.
- 2 Click on the **Delete** button to the left of the prompt list.

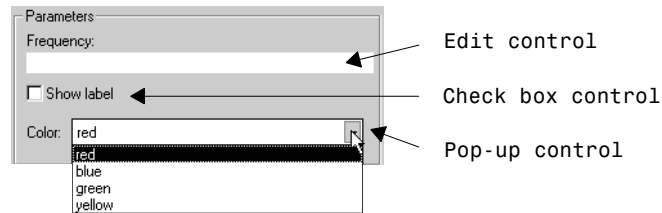
### Moving a Prompt

To move a prompt in the list:

- 1 Select the prompt you want to move.
- 2 To move the prompt up one position in the prompt list, click on the **Up** button to the left of the prompt list. To move the prompt down one position, click on the **Down** button.

## Control Types

Simulink enables you to choose how parameter values are entered or selected. You can create three styles of controls: edit fields, check boxes, and pop-up controls. For example, this figure shows the parameter area of a mask dialog which uses all three styles of controls (with the pop-up control open):



### Defining an Edit Control

An *edit field* enables the user to enter a parameter value by typing it into a field. This figure shows how the prompt for the sample edit control was defined





The value of the variable associated with the parameter (freq) is determined by the **Assignment** type defined for the prompt

Assignment	Value
Evaluate	The result of evaluating the expression entered in the field.
Literal	The actual string entered in the field.

### Defining a Check Box Control

A *check box* enables the user to choose between two alternatives by selecting or deselecting a check box. This figure shows how the sample check box control is defined.

The value of the variable associated with the parameter (label) depends on whether the check box is selected and the **Assignment** type defined for the prompt.

Check box	Evaluated Value	Literal Value
Checked	1	'on'
Not checked	0	'off'

### Defining a Pop-Up Control

A *popup* enables the user to choose a parameter value from a list of possible values. You specify the list in the **Popup strings** field, separating items with a vertical line (|). This figure shows how the sample pop-up control is defined.

The value of the variable associated with the parameter (`color`) depends on the item selected from the pop-up list and the **Assignment** type defined for the prompt.

Assignment	Value
Evaluate	The index of the value selected from the list, starting with 1. For example, if the third item is selected, the parameter value is 3.
Literal	A string that is the value selected. If the third item is selected, the parameter value is 'green'.

### Default Values for Masked Block Parameters

To change default parameter values in a masked library block, follow these steps:

- 1 Unlock the library.
- 2 Open the block to access its dialog box, fill in the desired default values, and close the dialog box.
- 3 Save the library.

When the block is copied into a model and opened, the default values appear on the block's dialog box.

For more information about libraries, see Chapter 3.

### Tunable Parameters

A tunable parameter is a mask parameter that a user can modify at runtime. When you create a mask, all its parameters are tunable. You can subsequently disable or re-enable tuning of any of a mask's parameters via the `MaskTunableValues` parameter. The value of this parameter is a cell array of strings, each of which corresponds to one of a masked block's parameters. The first cell corresponds to the first parameter, the second cell to the second parameter, and so on. If a parameter is tunable, the value of the corresponding cell is `on`; otherwise, the value is `off`. To enable or disable tuning of a parameter, first get the cell array, using `get_param`. Then, set the

corresponding cell to `on` or `off` and reset the `MaskTunableValues` parameter using `set_param`. For example, the following commands disable tuning of the first parameter of the currently selected masked block:

```
ca = get_param(gcb, 'MaskTunableValues');  
ca(1) = 'off'  
set_param(gcb, 'MaskTunableValues', ca)
```

After changing a block's tunable parameters, make the changes permanent by saving the block.

## Initialization Commands

Initialization commands define variables that reside in the mask workspace. These variables can be used by all initialization commands defined for the mask, by blocks in the masked subsystem, and by commands that draw the block icon (drawing commands).

Simulink executes the initialization commands when:

- The model is loaded.
- The simulation is started or the block diagram is updated.
- The masked block is rotated.
- The block's icon needs to be redrawn and the plot commands depend on variables defined in the initialization commands.

Initialization commands are valid MATLAB expressions, consisting of MATLAB functions, 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.

## The Mask Workspace

Simulink creates a local workspace, called a *mask workspace*, when either of the following occurs:

- The mask contains initialization commands.
- The mask defines prompts and associates variables with those prompts.

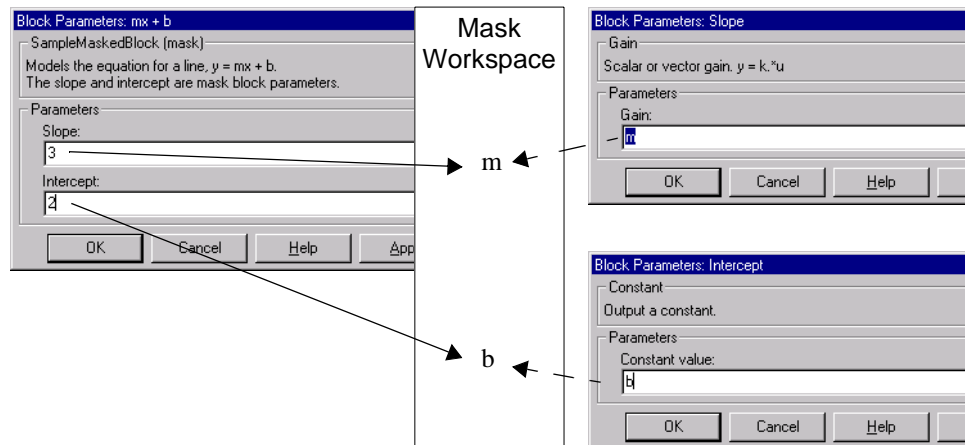
Masked blocks cannot access the base workspace or other mask workspaces.

The contents of a mask workspace include the variables associated with the mask's parameters and variables defined by initialization commands. The variables in the mask workspace can be accessed by the masked block. If the block is a subsystem, they can also be accessed by all blocks in the subsystem.

Mask workspaces are analogous to the local workspaces used by M-file functions. You can think of the expressions entered into the dialog boxes of the underlying blocks and the initialization commands entered on the Mask Editor as lines of an M-file function. Using this analogy, the local workspace for this "function" is the mask workspace.

In the  $mx + b$  example, described earlier in this chapter, the Mask Editor explicitly creates  $m$  and  $b$  in the mask workspace by associating a variable with a mask parameter. However, variables in the mask workspace are not explicitly assigned to blocks underneath the mask. Instead, blocks beneath the mask have access to all variables in the mask workspace. It may be instructive to think of the underlying blocks as "looking into" the mask workspace.

The figure below shows the mapping of values entered in the mask dialog box to variables in the mask workspace (indicated by the solid line) and the access of those variables by the underlying blocks (indicated by the dashed line).



## Debugging Initialization Commands

You can debug initialization commands in these ways:

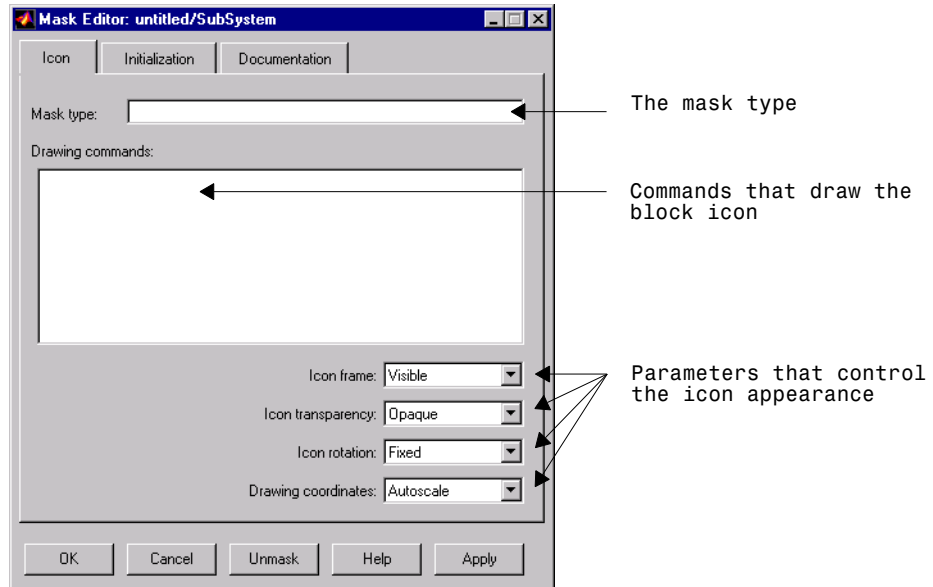
- Specify an initialization command without a terminating semicolon to echo its results to the command window.
- Place a keyboard command in the initialization commands to stop execution and give control to the keyboard. For more information, see the help text for the keyboard command.
- Enter either of these commands in the MATLAB command window.

```
dbstop if error  
dbstop if warning
```

If an error occurs in the initialization commands, execution stops and you can examine the mask workspace. For more information, see the help text for the `dbstop` command.

## The Icon Page

The **Icon** page enables you to customize the masked block's icon. You create a custom icon by specifying commands in the **Drawing commands** field. You can create icons that show descriptive text, state equations, images, and graphics. This figure shows the **Icon** page.



Drawing commands have access to all variables in the mask workspace.

Drawing commands can display text, one or more plots, or show a transfer function. If you enter more than one command, the results of the commands are drawn on the icon in the order the commands appear.

### Displaying Text on the Block Icon

To display text on the icon, enter one of these drawing commands.

```
disp('text') or disp(variablename)
```

```
text(x, y, 'text')
```

```
text(x, y, stringvariablename)
```

```

text(x, y, text, 'horizontalAlignment', halign,
'verticalAlignment', valign)

fprintf('text') or fprintf('format', variablename)

port_label(port_type, port_number, label)

```

The `disp` command displays text or the contents of `variablename` centered on the icon.

The `text` command places a character string (`text` or the contents of `stringvariablename`) at a location specified by the point (`x,y`). The units depend on the **Drawing coordinates** parameter. For more information, see “Controlling Icon Properties” on page 6-23.

You can optionally specify the horizontal and/or vertical alignment of the text relative to the point (`x, y`) in the `text` command. For example, the command

```
text(0.5, 0.5, 'foobar', 'horizontalAlignment', 'center')
```

centers `foobar` in the icon.

The `text` command offers the following horizontal alignment options.

<b>Option</b>	<b>Aligns</b>
<code>left</code>	The left end of the text at the specified point.
<code>right</code>	The right end of the text at the specified point.
<code>center</code>	The center of the text at the specified point.

The `text` command offers the following vertical alignment options.

<b>Option</b>	<b>Aligns</b>
<code>base</code>	The baseline of the text at the specified point.
<code>bottom</code>	The bottom line of the text at the specified point.
<code>middle</code>	The midline of the text at the specified point.

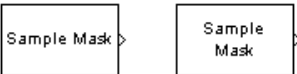
Option	Aligns
cap	The capitals line of the text at the specified point.
top	The top of the text at the specified point.

The `fprintf` command displays formatted text centered on the icon and can display text along with the contents of `variablename`.

**Note** While these commands are identical in name to their corresponding MATLAB functions, they provide only the functionality described above.

To display more than one line of text, use `\n` to indicate a line break. For example, the figure below shows two samples of the `disp` command

```
disp('Sample Mask')    disp('Sample\nMask')
```



The `port_label` command lets you specify the labels of ports displayed on the icon. The command's syntax is

```
port_label(port_type, port_number, label)
```

where `port_type` is either 'input' or 'output', `port_number` is an integer, and `label` is a string specifying the port's label. For example, the command

```
port_label('input', 1, 'a')
```

defines `a` as the label of input port 1.

## Displaying Graphics on the Block Icon

You can display plots on your masked block icon by entering one or more `plot` commands. You can use these forms of the `plot` command.

```
plot(Y);
plot(X1,Y1,X2,Y2,...);
```

`plot(Y)` plots, for a vector `Y`, each element against its index. If `Y` is a matrix, it plots each column of the matrix as though it were a vector.



`plot(X1,Y1,X2,Y2,...)` plots the vectors Y1 against X1, Y2 against X2, and so on. Vector pairs must be the same length and the list must consist of an even number of vectors.

For example, this command generates the plot that appears on the icon for the Ramp block, in the Sources library. The icon appears below the command.

```
plot([0 1 5], [0 0 4])
```



Plot commands can include NaN and inf values. When NaNs or infs are encountered, Simulink stops drawing, then begins redrawing at the next numbers that are not NaN or inf.

The appearance of the plot on the icon depends on the value of the **Drawing coordinates** parameter. For more information, see “Controlling Icon Properties” on page 6-23.

Simulink displays three question marks (? ? ?) in the block icon and issues warnings in these situations:

- When the values for the parameters used in the drawing commands are not yet defined (for example, when the mask is first created and values have not yet been entered into the mask dialog box).
- When a masked block parameter or drawing command is entered incorrectly.

## Displaying Images on Masks

The masked dialog functions, `image` and `patch`, enable you to display bitmapped images and draw patches on masked block icons.

`image(a)` displays the image `a` where `a` is an M by N by 3 array of RGB values. You can use the MATLAB commands, `imread` and `ind2rgb`, to read and convert bitmap files to the necessary matrix format. For example,

```
image(imread('icon.tif'))
```

reads the icon image from a TIFF file named `icon.tif` in the MATLAB path.

`image(a, [x, y, w, h])` creates the image at the specified position relative to the lower left corner of the mask.

`image(a, [x, y, w, h], rotation)` allows you to specify whether the image rotates ('on') or remains stationary ('off') as the icon rotates. The default is 'off'.

`patch(x, y)` creates a solid patch having the shape specified by the coordinate vectors `x` and `y`. The patch's color is the current foreground color.

`patch(x, y, [r g b])` creates a solid patch of the color specified by the vector `[r g b]`, where `r` is the red component, `g` the green, and `b` the blue. For example,

```
patch([0 .5 1], [0 1 0], [1 0 0])
```

creates a red triangle on the mask's icon.

## Displaying a Transfer Function on the Block Icon

To display a transfer function equation in the block icon, enter the following command in the **Drawing commands** field.

```
dpoly(num, den)
dpoly(num, den, 'character')
```

`num` and `den` are vectors of transfer function numerator and denominator coefficients, typically defined using initialization commands. The equation is expressed in terms of the specified character. The default is `s`. When the icon is drawn, the initialization commands are executed and the resulting equation is drawn on the icon.

- To display a continuous transfer function in descending powers of `s`, enter `dpoly(num, den)`

For example, for `num = [0 0 1]`; and `den = [1 2 1]`; the icon looks like this.

$$\frac{1}{s^2+2s+1}$$

- To display a discrete transfer function in descending powers of `z`, enter `dpoly(num, den, 'z')`

For example, for `num = [0 0 1]`; and `den = [1 2 1]`; the icon looks like this.

$$\frac{1}{z^2+2z+1}$$

- To display a discrete transfer function in ascending powers of  $1/z$ , enter `dpoly(num, den, 'z-')`

For example, for `num` and `den` as defined above, the icon looks like this.

$$\frac{z^2}{1+2z^{-1}+z^2}$$

- To display a zero-pole gain transfer function, enter `droots(z, p, k)`

For example, the above command creates this icon for these values.

`z = []; p = [-1 -1]; k = 1;`

$$\frac{1}{(s+1)(s+1)}$$

You can add a fourth argument ('z' or 'z-') to express the equation in terms of  $z$  or  $1/z$ .

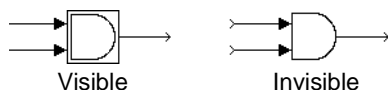
If the parameters are not defined or have no values when you create the icon, Simulink displays three question marks (? ? ?) in the icon. When the parameter values are entered in the mask dialog box, Simulink evaluates the transfer function and displays the resulting equation in the icon.

## Controlling Icon Properties

You can control a masked block's icon properties by selecting among the choices below the **Drawing commands** field.

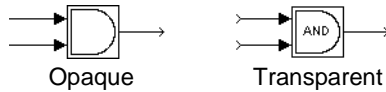
### Icon frame

The icon frame is the rectangle that encloses the block. You can choose to show or hide the frame by setting the **Icon frame** parameter to **Visible** or **Invisible**. The default is to make the icon frame visible. For example, this figure shows visible and invisible icon frames for an AND gate block.



## Icon transparency

The icon can be set to **Opaque** or **Transparent**, either hiding or showing what is underneath the icon. **Opaque**, the default, covers information Simulink draws, such as port labels. This figure shows opaque and transparent icons for an AND gate block. Notice the text on the transparent icon.



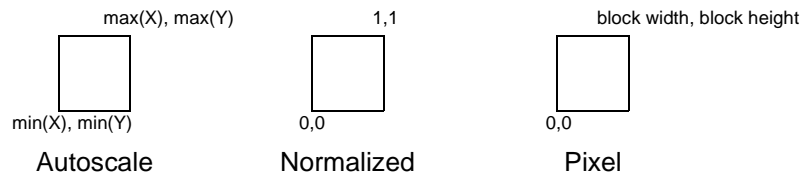
## Icon rotation

When the block is rotated or flipped, you can choose whether to rotate or flip the icon, or to have it remain fixed in its original orientation. The default is not to rotate the icon. The icon rotation is consistent with block port rotation. This figure shows the results of choosing **Fixed** and **Rotates** icon rotation when the AND gate block is rotated.



## Drawing coordinates

This parameter controls the coordinate system used by the drawing commands. This parameter applies only to plot and text drawing commands. You can select from among these choices: **Autoscale**, **Normalized**, and **Pixel**.



- **Autoscale** automatically scales the icon within the block frame. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors.

$X = [0 \ 2 \ 3 \ 4 \ 9]; Y = [4 \ 6 \ 3 \ 5 \ 8];$



The lower-left corner of the block frame is (0,3) and the upper-right corner is (9,8). The range of the  $x$ -axis is 9 (from 0 to 9), while the range of the  $y$ -axis is 5 (from 3 to 8).

- **Normalized** draws the icon within a block frame whose bottom-left corner is (0,0) and whose top right corner is (1,1). Only  $X$  and  $Y$  values between 0 and 1 appear. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors.

$X = [.0 \ .2 \ .3 \ .4 \ .9]; Y = [.4 \ .6 \ .3 \ .5 \ .8];$



- **Pixel** draws the icon with  $X$  and  $Y$  values expressed in pixels. The icon is not automatically resized when the block is resized. To force the icon to resize with the block, define the drawing commands in terms of the block size.

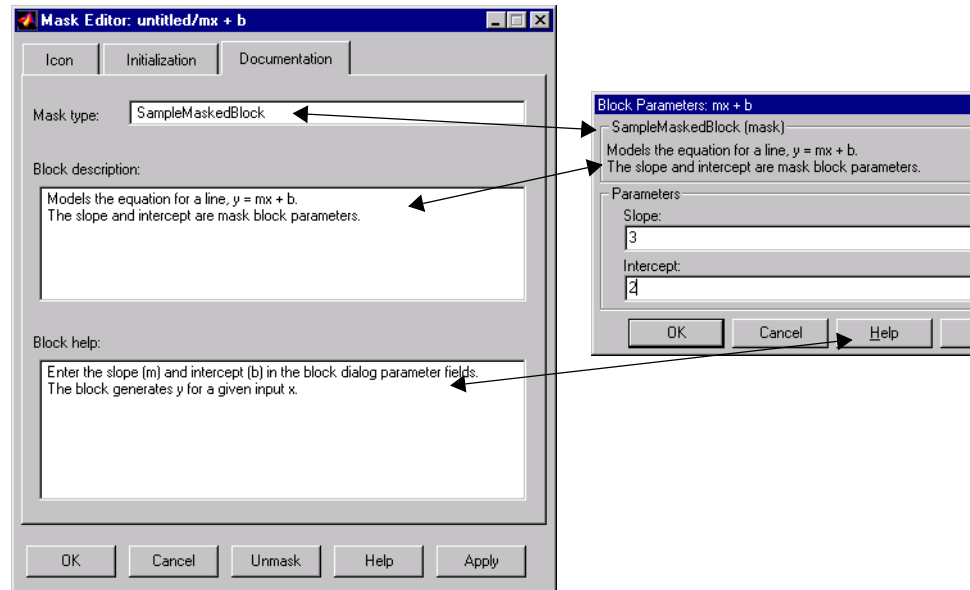
This example demonstrates how to create an improved icon for the  $m \times b$  sample masked subsystem discussed earlier in this chapter. These initialization commands define the data that enables the drawing command to produce an accurate icon regardless of the shape of the block.

```
pos = get_param(gcb, '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)`.

## The Documentation Page

The **Documentation** page enables you to define or modify the type, description, and help text for a masked block. This figure shows how fields on the **Documentation** page correspond to the  $mx+b$  sample mask block's dialog box.



### The Mask Type Field

The mask type is a block classification used only for purposes of documentation. It appears in the block's dialog box and on all Mask Editor pages for the block. You can choose any name you want for the mask type. When Simulink creates the block's dialog box, it adds "(mask)" after the mask type to differentiate masked blocks from built-in blocks.

### The Block Description Field

The block description is informative text that appears in the block's dialog box in the frame under the mask type. If you are designing a system for others to use, this is a good place to describe the block's purpose or function.

Simulink automatically wraps long lines of text. You can force line breaks by using the **Enter** or **Return** key.

## The Mask Help Text Field

You can provide help text that gets displayed when the **Help** button is pressed on the masked block's dialog box. If you create models for others to use, this is a good place to explain how the block works and how to enter its parameters.

You can include user-written documentation for a masked block's help. You can specify any of the following for the masked block help text:

- URL specification (a string starting with `http:`, `www`, `file:`, `ftp:`, or `mailto:`)
- web command (launches a browser)
- `eval` command (evaluates a MATLAB string)
- Static text displayed in the Web browser

Simulink examines the first line of the masked block help text. If it detects a URL specification, web command, or `eval` command, it accesses the block help as directed; otherwise, the full contents of the masked block help text are displayed in the browser.

These examples illustrate several acceptable commands.

```
web([docroot '/My Blockset Doc/' get_param(gcf, 'MaskType')
    '.html'])
eval('!Word My_Spec.doc')
http://www.mathworks.com
file:///c:/mydir/helpdoc.html
www.mathworks.com
```

Simulink automatically wraps long lines of text.

## Creating Dynamic Dialogs for Masked Blocks

Simulink allows you to create dialogs for masked blocks whose appearance changes in response to user input. Features of masked dialog features 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. Simulink grays a disabled control to indicate visually that it is disabled.
- **Parameter values**  
Changing a parameter can cause related parameters to be set to appropriate values.

Creating a dynamic masked dialog entails using the mask editor in combination with the Simulink `set_param` command. Specifically, you first use the mask editor to define all the dialog's parameters both static and dynamic. Next you use the Simulink `set_param` command at the MATLAB command line to specify callback functions that define the dialog's response to user input. Finally you save the model or library containing the masked subsystem to complete the creation of the dynamic masked dialog.

### Setting Masked Block Dialog Parameters

Simulink 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 advantage? 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 at the MATLAB command line to specify 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.

## Predefined Masked Dialog Parameters

Simulink associates the following predefined parameters 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.

The easiest way to set callbacks for a mask dialog 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(gcf, '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.

### 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(gcf, 'MaskEnables', {'on', 'on', 'off'});
```

would disable the third control of the currently open masked block's dialog. Simulink colors disabled controls gray to indicate visually that they are disabled.

### **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(gcf, 'MaskVisibilities', {'on', 'off', 'on'});
```

would hide the control for the currently selected block's second user-defined mask parameter. Simulink expands or shrinks a dialog to show or hide a control, respectively.

# Conditionally Executed Subsystems

---

<b>Introduction</b> . . . . .	7-2
<b>Enabled Subsystems</b> . . . . .	7-3
Creating an Enabled Subsystem . . . . .	7-3
Blocks an Enabled Subsystem Can Contain . . . . .	7-5
<b>Triggered Subsystems</b> . . . . .	7-8
Creating a Triggered Subsystem . . . . .	7-9
Function-Call Subsystems . . . . .	7-10
Blocks That a Triggered Subsystem Can Contain . . . . .	7-10
<b>Triggered and Enabled Subsystems</b> . . . . .	7-11
Creating a Triggered and Enabled Subsystem . . . . .	7-11
A Sample Triggered and Enabled Subsystem . . . . .	7-12
Creating Alternately Executing Subsystems . . . . .	7-12

## Introduction

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*.

Conditionally executed subsystems can be very useful when building complex models that contain components whose execution depends on other components.

Simulink supports three types of conditionally executed subsystems:

- An *enabled subsystem* executes while the control signal is positive. It starts execution at the time step where the control signal crosses zero (from the negative to the positive direction) and continues execution while the control signal remains positive. Enabled subsystems are described in more detail on “Enabled Subsystems” on page 7-3.
- 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 on “Triggered Subsystems” on page 7-8.
- 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 7-11 for more information.

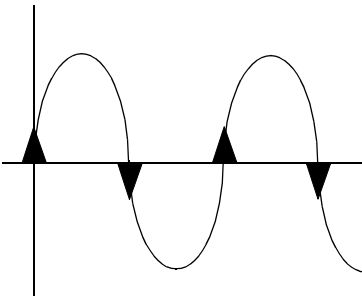
# Enabled Subsystems

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.



Simulink 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 Signals & Systems library into a subsystem. Simulink adds an enable symbol and an enable control input port to the Subsystem block icon:

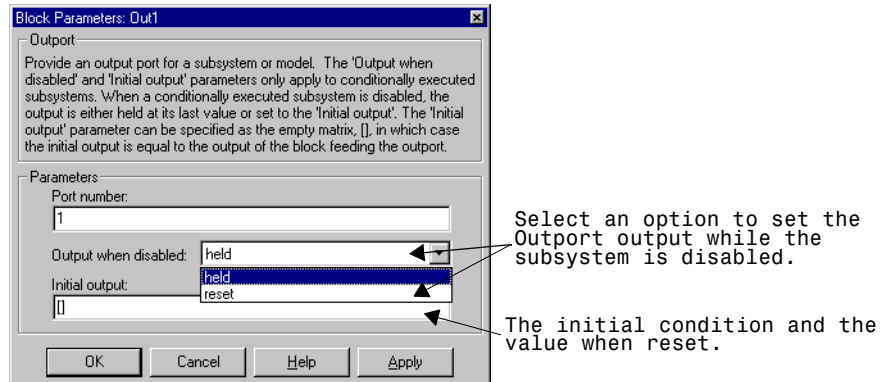


## Setting Output Values While the Subsystem Is Disabled

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

- Choose **held** to cause the output to maintain its most recent value.
- Choose **reset** to cause the output to revert to its initial condition. Set the **Initial output** to the initial value of the output.

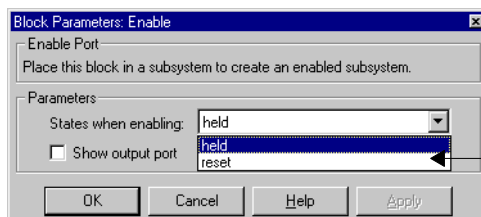


## Setting States When the Subsystem Becomes Re-enabled

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, as shown in the dialog box below:

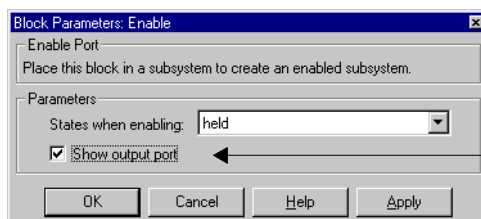
- Choose **held** to cause the states to maintain their most recent values.
- Choose **reset** to cause the states to revert to their initial conditions.



Select an option to set the states when the subsystem is re-enabled.

## 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:



Select this check box to show the output port.

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

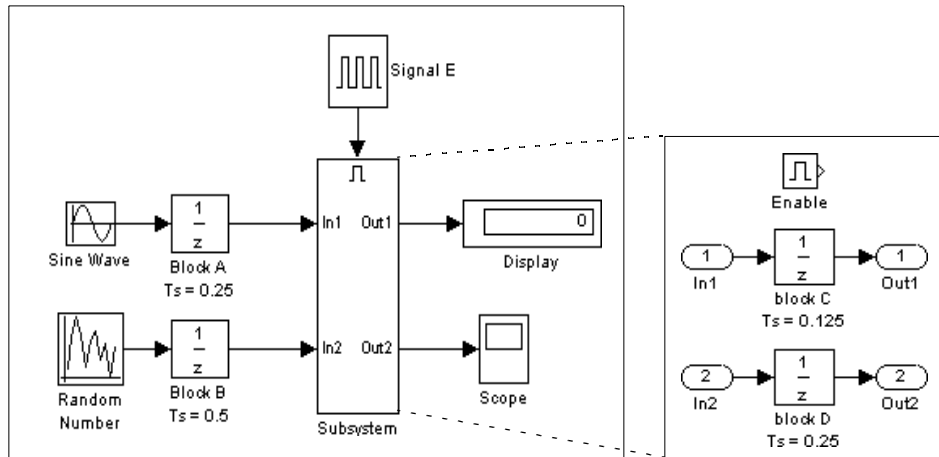
An enabled subsystem can contain any block, whether continuous or discrete. Discrete blocks in an enabled subsystem execute only when the subsystem executes, and only when their sample times are synchronized with the simulation sample time. Enabled subsystems and the model use a common clock.

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

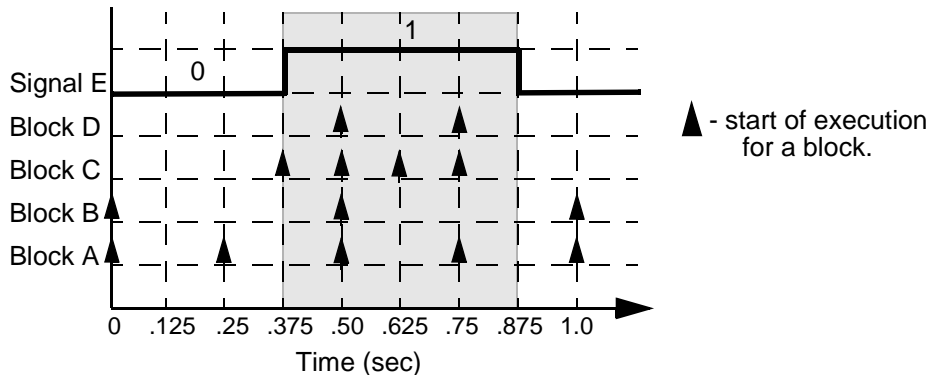
- Block A, which has a sample time of 0.25 seconds.
- Block B, which has a sample time of 0.5 seconds.

- Block C, within the Enabled subsystem, which has a sample time of 0.125 seconds.
- Block D, also within the Enabled subsystem, which has a sample time of 0.25 seconds.

The enable control signal is generated by a Pulse Generator block, labeled Signal E, which changes from 0 to 1 at 0.375 seconds and returns to 0 at 0.875 seconds.



The chart below indicates when the discrete blocks execute:



Blocks A and B execute independent of the enable signal because they are not part of the enabled subsystem. When the enable signal becomes positive, blocks



C and D execute at their assigned sample rates until the enable signal becomes zero again. Note that block C does not execute at 0.875 seconds when the enable signal changes to zero.

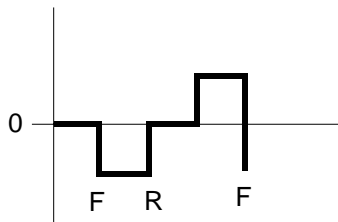
## Triggered Subsystems

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

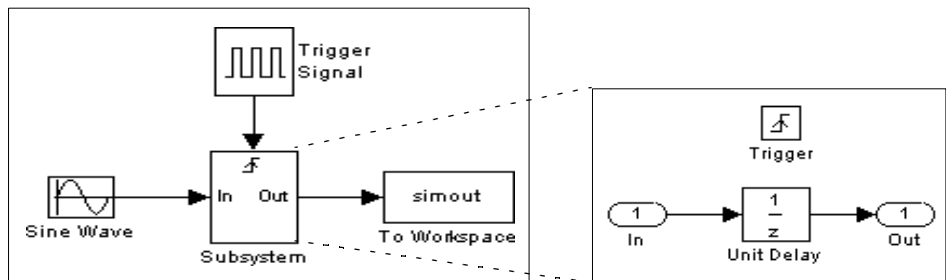
A triggered subsystem has a single control input, called the *trigger input*, which determines whether the subsystem executes. You can choose from three types of trigger events to force a triggered subsystem to begin execution:

- **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.

For example, this figure shows when rising (R) and falling (F) triggers occur for the given control signal.



A simple example of a trigger subsystem is illustrated below:



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

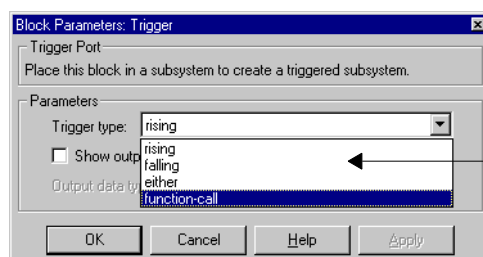
## Creating a Triggered Subsystem

You create a triggered subsystem by copying the Trigger block from the Signals & Systems library into a subsystem. Simulink adds a trigger symbol and a trigger control input port to the Subsystem block icon:



To select the trigger type, open the Trigger block dialog box and select one of the choices for the **Trigger type** parameter, as shown in the dialog box below:

- **rising** forces a trigger whenever the trigger signal crosses zero in a positive direction.
- **falling** forces a trigger whenever the trigger signal crosses zero in a negative direction.
- **either** forces a trigger whenever the trigger signal crosses zero in either direction.



Select the trigger type from these choices.

Simulink uses different symbols on the Trigger and Subsystem blocks to indicate rising and falling triggers (or either). This figure shows the trigger symbols on Subsystem blocks:

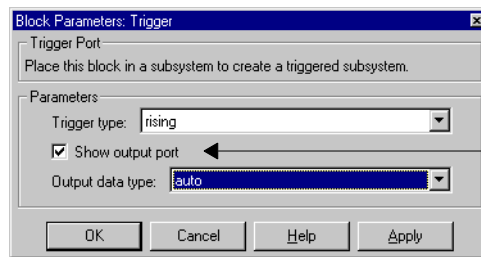


### Outputs and States Between Trigger Events

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.



Select this check box to show the output port.

The **Output data type** field allows you to 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 set to the data type (either `int8` or `double`) of the port to which the signal is connected.

### Function-Call Subsystems

You can create a triggered subsystem whose execution is determined by logic internal to an S-function instead of by the value of a signal. These subsystems are called *function-call subsystems*. For more information about function-call subsystems, see the companion guide *Writing S-Functions*.

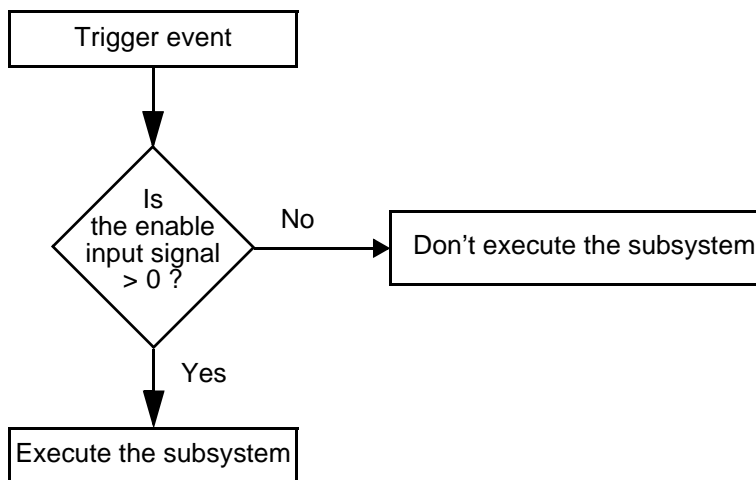
### Blocks That a Triggered Subsystem Can Contain

Triggered systems execute only at specific times during a simulation. As a result, the only blocks that are suitable for use in a triggered subsystem are:

- Blocks with inherited sample time, such as the Logical Operator block or the Gain block.
- Discrete blocks having their sample time set to `-1`, which indicates that the sample time is inherited from the driving block.

## Triggered and Enabled Subsystems

A third kind of conditionally executed 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, Simulink checks the enable input port to evaluate the enable control signal. If its value is greater than zero, Simulink executes the subsystem. 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 Signals & Systems library into an existing subsystem. Simulink adds enable and trigger symbols and enable and trigger and enable control inputs to the Subsystem block icon.

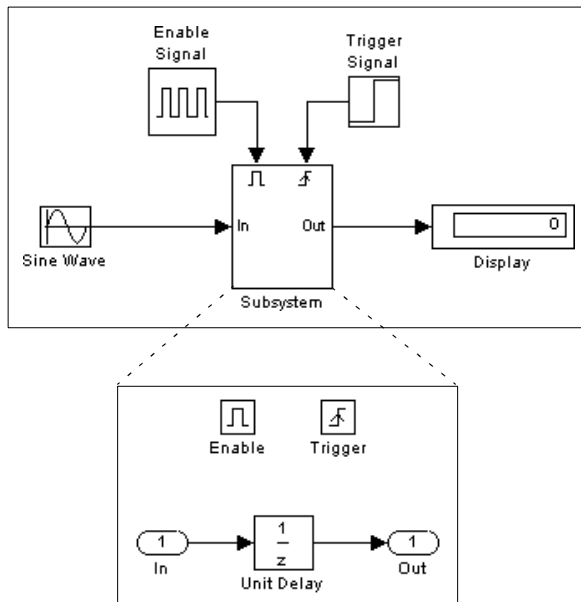


You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see “Setting Output Values While the Subsystem Is Disabled” on page 7–4. Also, you can specify what the values of the states are when the subsystem is re-enabled. See “Setting States When the Subsystem Becomes Re-enabled” on page 7–4.

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

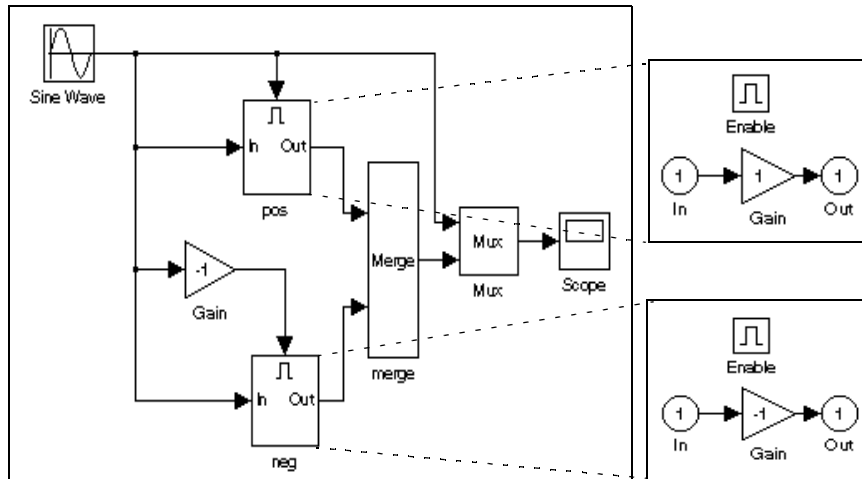
A simple example of a triggered and enabled subsystem is illustrated in the model below.



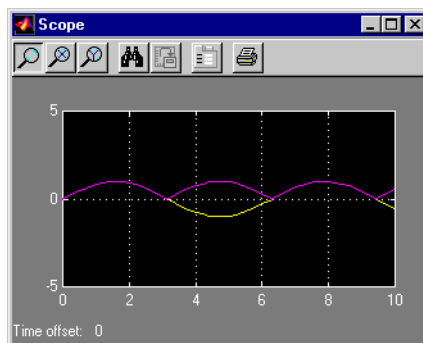
### Creating Alternately Executing Subsystems

You can use conditionally executed subsystems in combination with Merge blocks (see Merge on page 8-126) to create sets of subsystems that execute alternately, depending on the current state of the model. For example, the following figure shows a model that uses two enabled blocks and a Merge block

to model an inverter, that is, a device that converts AC current to pulsating DC current.



In this example, 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 to create the following display.







# Block Reference

---

<b>What Each Block Reference Page Contains . . . . .</b>	<b>8-2</b>
<b>Simulink Block Libraries . . . . .</b>	<b>8-3</b>

## What Each Block Reference Page Contains

Blocks appear in alphabetical order and contain this information:

- The block name, icon, and block library that contains the block
- The purpose of the block
- A description of the block's use
- The block dialog box and parameters
- The block characteristics, including some or all of these, as they apply to the block:
  - Direct Feedthrough – whether the block or any of its ports has direct feedthrough. For more information, see “Algebraic Loops” on page 9-7.
  - Sample Time – how the block's sample time is determined, whether by the block itself (as is the case with discrete and continuous blocks) or inherited from the block that drives it or is driven by it. For more information, see “Sample Time” on page 9-13.
  - Scalar Expansion – whether or not scalar values are expanded to vectors. Some blocks expand scalar inputs and/or parameters as appropriate. For more information, see “Scalar Expansion of Inputs and Parameters” on page 3-18.
  - States – the number of discrete and continuous states.
  - Vectorized – whether the block accepts and/or generates vector signals. For more information, see “Vector Input and Output” on page 3-18.
  - Zero Crossings – whether the block detects state events. For more information, see “Zero Crossings” on page 9-3.

## Simulink Block Libraries

Simulink organizes its blocks into block libraries according to their behavior. The **simulink** window displays the block library icons and names:

- The *Sources* library contains blocks that generate signals.
- The *Sinks* library contains blocks that display or write block output.
- The *Discrete* library contains blocks that describe discrete-time components.
- The *Continuous* library contains blocks that describe linear functions.
- The *Nonlinear* library contains blocks that describe nonlinear functions.
- The *Math* library contains blocks that describe general mathematics functions.
- The *Functions & Tables* library contains blocks that describe general functions and table look-up operations.
- The *Signal & Systems* library contains blocks that allow multiplexing and demultiplexing, implement external input/output, pass data to other parts of the model, create subsystems, and perform other functions.
- The *Blocksets and Toolboxes* library contains the Extras block library of specialized blocks.
- The *Demos* library contains useful MATLAB and Simulink demos.

**Table 8-1: Sources Library Blocks**

Block Name	Purpose
Band-Limited White Noise	Introduce white noise into a continuous system.
Chirp Signal	Generate a sine wave with increasing frequency.
Clock	Display and provide the simulation time.
Constant	Generate a constant value.
Digital Clock	Generate simulation time at the specified sampling interval.

**Table 8-1: Sources Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
Digital Pulse Generator	Generate pulses at regular intervals.
From File	Read data from a file.
From Workspace	Read data from a matrix defined in the workspace.
Pulse Generator	Generate pulses at regular intervals.
Ramp	Generate a constantly increasing or decreasing signal.
Random Number	Generate normally distributed random numbers.
Repeating Sequence	Generate a repeatable arbitrary signal.
Signal Generator	Generate various waveforms.
Sine Wave	Generate a sine wave.
Step	Generate a step function.
Uniform Random Number	Generate uniformly distributed random numbers.

**Table 8-2: Sinks Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Display	Show the value of the input.
Scope	Display signals generated during a simulation.
Stop Simulation	Stop the simulation when the input is nonzero.
To File	Write data to a file.

**Table 8-2: Sinks Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
To Workspace	Write data to a matrix in the workspace.
XY Graph	Display an X-Y plot of signals using a MATLAB figure window.

**Table 8-3: Discrete Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Discrete Filter	Implement IIR and FIR filters.
Discrete State-Space	Implement a discrete state-space system.
Discrete-Time Integrator	Perform discrete-time integration of a signal.
Discrete Transfer Fcn	Implement a discrete transfer function.
Discrete Zero-Pole	Implement a discrete transfer function specified in terms of poles and zeros.
First-Order Hold	Implement a first-order sample-and-hold.
Unit Delay	Delay a signal one sample period.
Zero-Order Hold	Implement zero-order hold of one sample period.

**Table 8-4: Continuous Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Derivative	Output the time derivative of the input.
Integrator	Integrate a signal.

**Table 8-4: Continuous Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
Memory	Output the block input from the previous time step.
State-Space	Implement a linear state-space system.
Transfer Fcn	Implement a linear transfer function.
Transport Delay	Delay the input by a given amount of time.
Variable Transport Delay	Delay the input by a variable amount of time.
Zero-Pole	Implement a transfer function specified in terms of poles and zeros.

**Table 8-5: Math Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Abs	Output the absolute value of the input.
Algebraic Constraint	Constrain the input signal to zero.
Combinatorial Logic	Implement a truth table.
Complex to Magnitude-Angle	Output the phase and magnitude of a complex input signal.
Complex to Real-Imag	Output the real and imaginary parts of a complex input signal.
Derivative	Output the time derivative of the input.
Dot Product	Generate the dot product.
Gain	Multiply block input.
Logical Operator	Perform the specified logical operation on the input.

**Table 8-5: Math Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
Magnitude-Angle to Complex	Output a complex signal from magnitude and phase inputs.
Math Function	Perform a mathematical function.
Matrix Gain	Multiply the input by a matrix.
MinMax	Output the minimum or maximum input value.
Product	Generate the product or quotient of block inputs.
Real-Imag to Complex	Output a complex signal from real and imaginary inputs.
Relational Operator	Perform the specified relational operation on the input.
Rounding Function	Perform a rounding function.
Sign	Indicate the sign of the input.
Slider Gain	Vary a scalar gain using a slider.
Sum	Generate the sum of inputs.
Trigonometric Function	Perform a trigonometric function.

**Table 8-6: Functions & Tables Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Fcn	Apply a specified expression to the input.
Look-Up Table	Perform piecewise linear mapping of the input.

**Table 8-6: Functions & Tables Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
Look-Up Table (2-D)	Perform piecewise linear mapping of two inputs.
MATLAB Fcn	Apply a MATLAB function or expression to the input.
S-Function	Access an S-function.

**Table 8-7: Nonlinear Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Backlash	Model the behavior of a system with play.
Coulomb & Viscous Friction	Model discontinuity at zero, with linear gain elsewhere.
Dead Zone	Provide a region of zero output.
Manual Switch	Switch between two inputs.
Multipoint Switch	Choose between block inputs.
Quantizer	Discretize input at a specified interval.
Rate Limiter	Limit the rate of change of a signal.
Relay	Switch output between two constants.
Saturation	Limit the range of a signal.
Switch	Switch between two inputs.



**Table 8-8: Signals & Systems Library Blocks**

<b>Block Name</b>	<b>Purpose</b>
Bus Selector	Output selected input signals.
Configurable Subsystem	Represent any block selected from a specified library.
Data Store Memory	Define a shared data store.
Data Store Read	Read data from a shared data store.
Data Store Write	Write data to a shared data store.
Data Type Conversion	Convert a signal to another data type.
Demux	Separate a vector signal into output signals.
Enable	Add an enabling port to a subsystem.
From	Accept input from a Goto block.
Goto	Pass block input to From blocks.
Goto Tag Visibility	Define the scope of a Goto block tag.
Ground	Ground an unconnected input port.
Hit Crossing	Detect crossing point.
IC	Set the initial value of a signal.
Inport	Create an input port for a subsystem or an external input.
Merge	Combine several input lines into a scalar line.
Model Info	Display revision control information in a model.

**Table 8-8: Signals & Systems Library Blocks (Continued)**

<b>Block Name</b>	<b>Purpose</b>
Mux	Combine several input lines into a vector line.
Outport	Create an output port for a subsystem or an external output.
Probe	Output an input signal's width, sample time, and/or signal type.
Subsystem	Represent a system within another system.
Terminator	Terminate an unconnected output port.
Trigger	Add a trigger port to a subsystem.
Width	Output the width of the input vector.

**Purpose** Output the absolute value of the input.

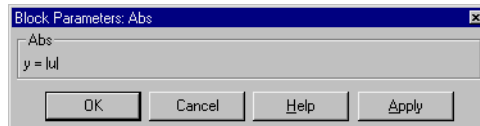
**Library** Math

**Description** The Abs block generates as output the absolute value of the input.



**Data Type Support** An Abs block accepts a real- or complex-valued input of type double and generates a real output of type double.

**Dialog Box**



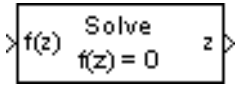
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	N/A
	Vectorized	Yes
	Zero Crossing	Yes, to detect zero

# Algebraic Constraint

**Purpose** Constrain the input signal to zero.

**Library** Math

**Description** The Algebraic Constraint block constrains the input signal  $f(z)$  to zero and outputs an algebraic state  $z$ . The block outputs the value necessary to produce a zero at the input. The output must affect the input through some feedback path. This enables you to specify algebraic equations for index 1 differential/algebraic systems (DAEs).

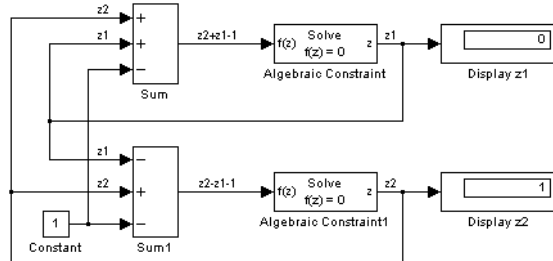


By default, the **Initial guess** parameter is zero. You can improve the efficiency of the algebraic loop solver by providing an **Initial guess** of the algebraic state  $z$  that is close to the solution value.

For example, the model below solves these equations:

$$\begin{aligned}z_2 + z_1 &= 1 \\z_2 - z_1 &= 1\end{aligned}$$

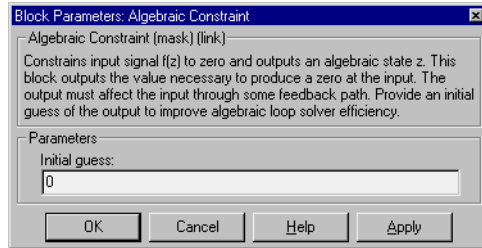
The solution is  $z_2 = 1$ ,  $z_1 = 0$ , as the Display blocks show.



## Data Type Support

An Algebraic Constraint block accepts and outputs real values of type double.

## Parameters and Dialog Box



### Initial guess

An initial guess of the solution value. The default is 0.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes
Zero Crossing	No

# Backlash

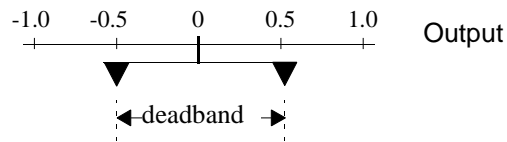
**Purpose** Model the behavior of a system with play.

**Library** Nonlinear

## Description



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



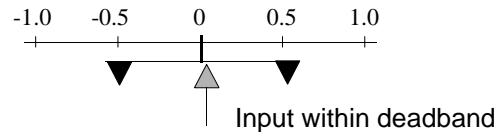
A system with play can be in one of three modes:

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

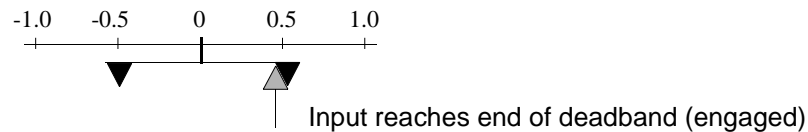
If the initial input is outside the deadband, the **Initial output** parameter value determines if the block is engaged in a positive or negative direction and the output at the start of the simulation is the input plus or minus half the deadband width.

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

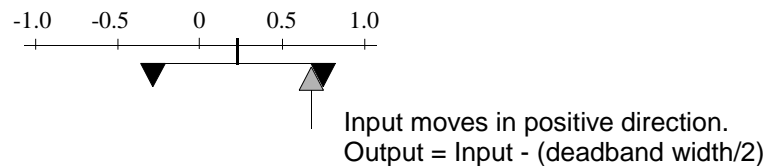
The figures below illustrate the block's operation when the initial input is within the deadband. The first figure shows the relationship between the input and the output while the system is in disengaged mode (and the default parameter values are not changed).



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



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



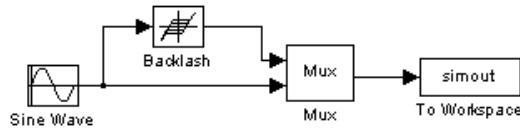
If the input reverses its direction, it disengages from the output. The output remains constant until the input either reaches the opposite end of the deadband or reverses its direction again and engages at the same end of the deadband. Now, as before, movement in the input causes equal movement in the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

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

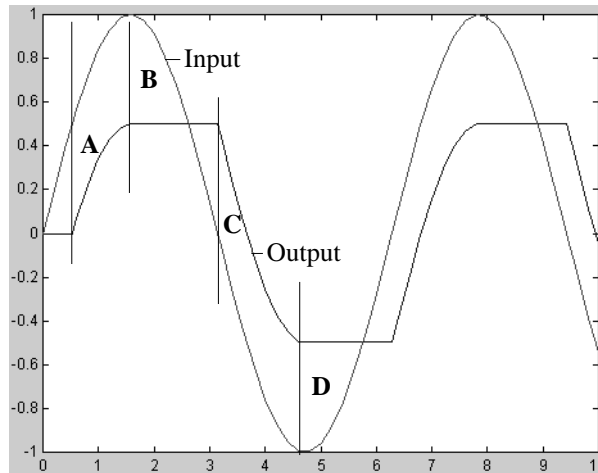
- 5 if the input,  $u$ , is between 4 and 6
- $u + 1$  if  $u < 4$
- $u - 1$  if  $u > 6$

# Backlash

This sample model and the plot that follows it show the effect of a sine wave passing through a Backlash block.



The Backlash block parameters are unchanged from their default values (the deadband width is 1 and the initial output is 0). Notice in the plotted output below that the Backlash block output is zero until the input reaches the end of the deadband (at 0.5). Now, the input and output are engaged and the output moves as the input does until the input changes direction (at 1.0). When the input reaches 0, it again engages the output at the opposite end of the deadband.



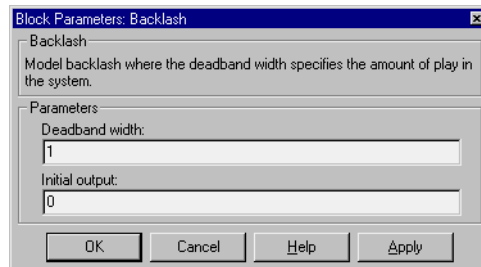
- A** Input engages in positive direction. Change in input causes equal change in output.
- B** Input disengages. Change in input does not affect output.
- C** Input engages in negative direction. Change in input causes equal change in output.
- D** Input disengages. Change in input does not affect output.

## Data Type Support

A Backlash block accepts and outputs real values of type double.



## Parameters and Dialog Box



### Deadband width

The width of the deadband. The default is 1.

### Initial output

The initial output value. The default is 0.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	Yes, to detect engagement with lower and upper thresholds

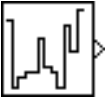
# Band-Limited White Noise

---

**Purpose** Introduce white noise into a continuous system.

**Library** Sources

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



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

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

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

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

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

## The Algorithm Used in the Block Implementation

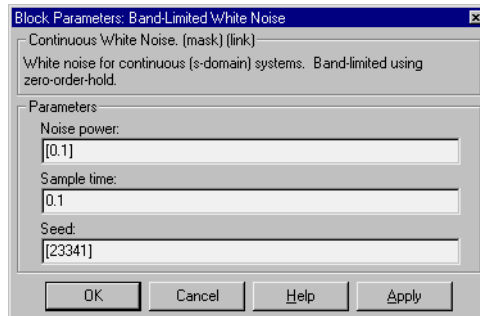
To produce the correct intensity of this noise, the covariance of the noise is scaled to reflect the implicit conversion from a continuous PSD to a discrete noise covariance. The appropriate scale factor is  $1/tc$ , where  $tc$  is the correlation time of the noise. This scaling ensures that the response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

white noise is infinite, the approximation used in this block has the property that the covariance of the block output is the **Noise Power** divided by  $tc$ .

## Data Type Support

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

## Parameters and Dialog Box



### Noise power

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

### Sample time

The correlation time of the noise. The default value is 0.1.

### Seed

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

## Characteristics

Sample Time	Discrete
Scalar Expansion	Of <b>Noise power</b> and <b>Seed</b> parameters and output
Vectorized	Yes
Zero Crossing	No

# Bus Selector

**Purpose** Select signals from an incoming bus.

**Library** Signals & Systems

**Description** The Bus Selector block accepts input from a Mux block or another Bus Selector block. This block has one input port. The number of output ports depends on the state of the **Muxed output** checkbox. If you check **Muxed output**, then the signals are combined at the output port and there is only one output port; otherwise, there is one output port for each selected signal.



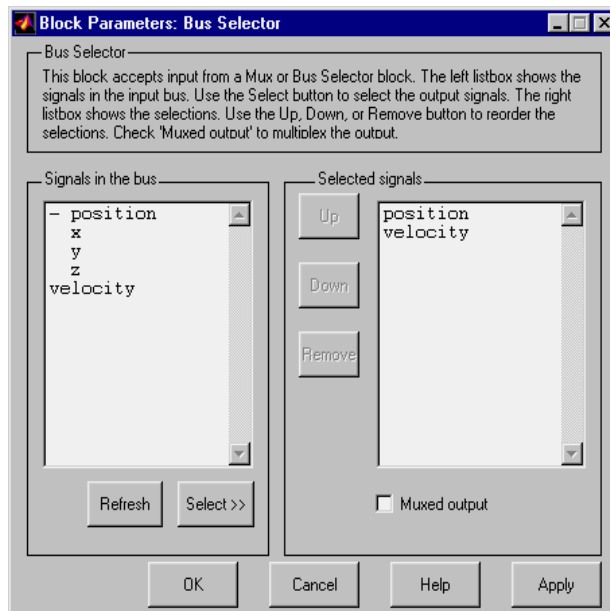
---

**Note** Simulink hides the name of a bus selector block when you copy it from the Simulink library to a model.

---

**Data Type Support** A Bus Selector block accepts and outputs real or complex values of any data type.

## Parameters and Dialog Box



## Signals in the bus

The **Signals in the bus** listbox shows the signals in the input bus. Use the **Select>>** button to select output signals from the **Signals in the bus** listbox.

## Selected signals

The **Selected signals** listbox shows the output signals. You can order the signals by using the **Up**, **Down**, and **Remove** buttons. Port connectivity is maintained when the signal order is changed.

If an output signal listed in the **Selected signals** listbox is not an input to the Bus Selector block, the signal name will be preceded by ???.

The signal label at the output port is automatically set by the block except when you check the **Muxed output** checkbox. If you try to change this label, you will get an error message stating that you cannot change the signal label of a line connected to the output of a Bus Selector block.

# Chirp Signal

**Purpose** Generate a sine wave with increasing frequency.

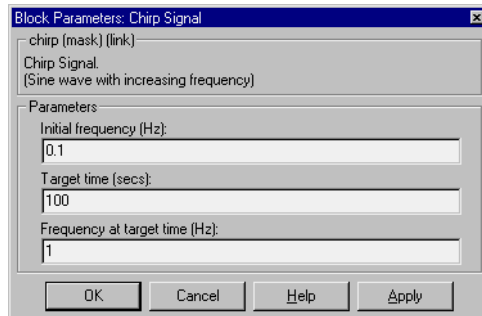
**Library** Sources

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



**Data Type Support** A Chirp Signal block outputs a real-valued signal of type double.

## Parameters and Dialog Box



### Initial frequency

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

### Target time

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

### Frequency at target time

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

<b>Characteristics</b>	Sample Time	Continuous
	Scalar Expansion	Of parameters

Vectorized	Yes
Zero Crossing	No

# Clock

---

**Purpose** Display and provide the simulation time.

**Library** Sources

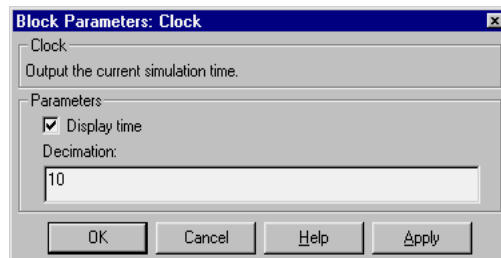
**Description** The Clock block outputs the current simulation time at each simulation step. This block is useful for other blocks that need the simulation time.



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

**Data Type Support** A Clock block outputs a real-valued signal of type double.

## Parameters and Dialog Box



### Display time

Use the **Display time** check box to display the current simulation time inside the Clock block icon, which will then have the following appearance.

### Decimation

The **Decimation** parameter value is the increment at which the clock gets updated; it can be any positive integer. For example, if the decimation is 1000, then for a fixed integration step of 1 millisecond, the clock will update at 1 second, 2 seconds, and so on.

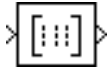
<b>Characteristics</b>	Sample Time	Continuous
	Scalar Expansion	N/A
	Vectorized	No
	Zero Crossing	No



**Purpose** Implement a truth table.

**Library** Math

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



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

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

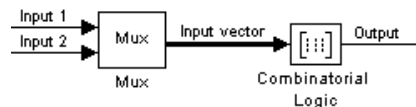
$$\text{number of rows} = 2^{\text{(number of inputs)}}$$

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

$$\text{row index} = 1 + u(m) \cdot 2^0 + u(m-1) \cdot 2^1 + \dots + u(1) \cdot 2^{m-1}$$

## Example of Two-Input AND Function

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



The table below indicates the combination of inputs that generate each output. The input signal labeled “Input 1” corresponds to the column in the table labeled Input 1. Similarly, the input signal “Input 2” corresponds to the column

# Combinatorial Logic

with the same name. The combination of these values determines which row of the Output column of the table gets passed as block output.

For example, if the input vector is [1 0], the input references the third row ( $2^{1*1} + 1$ ). So, the output value is 0.

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

## Example of Circuit

This sample circuit has three inputs: the two bits (**a** and **b**) to be summed and a carry-in bit (**c**). It has two outputs, the carry-out bit (**c'**) and the sum bit (**s**). Here is the truth table and the outputs associated with each combination of input values for this circuit:

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

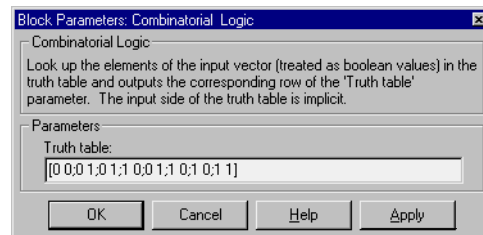
To implement this adder with the Combinatorial Logic block, you enter the 8-by-2 matrix formed by columns **c'** and **s** as the **Truth table** parameter.

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

## Data Type Support

A Combinatorial Logic block accepts real signals of type `boolean` or `double` and outputs the same type as the input. The elements of the truth table can be of type `boolean` or `double`. If the elements are of type `double`, they may have any values, not just “boolean” (0 or 1) values. If the data type of the truth table elements differs from the data type of the output signal, Simulink converts the truth table to the output type before computing the output.

## Parameters and Dialog Box



### Truth table

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

## Characteristics

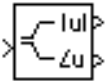
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Vectorized	Yes; the output width is the number of columns of the <b>Truth table</b> parameter
Zero Crossing	No

# Complex to Magnitude-Angle

**Purpose** Compute the magnitude and/or phase angle of a complex signal.

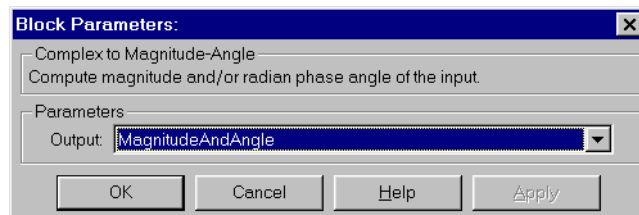
**Library** Math

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



**Data Type Support** See the description above.

## Parameters and Dialog Box



## Output

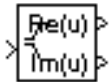
Determines the output of this block. Choose from the following values: `MagnitudeAndAngle` (outputs the input signal's magnitude and phase angle in radians), `Magnitude` (outputs the input's magnitude), `Angle` (outputs the input's phase angle in radians).

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	No
	Vectorized	No
	Zero Crossing	No

**Purpose** Output the real and imaginary parts of a complex input signal.

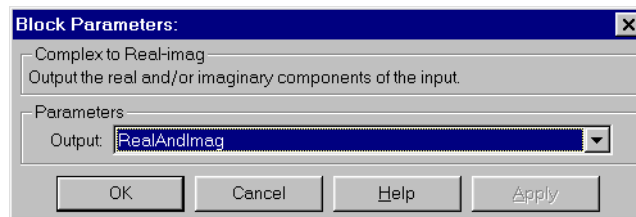
**Library** Math

**Description** The Complex to Real-Imag block accepts a complex-valued signal of type `double`. It outputs the real and/or imaginary part of the input signal, depending on the setting of the **Output** parameter. The outputs are real values of type `double`. The input may be a vector of complex signals, in which case the output signals are also vectors. The real signal vector contains the real parts of the corresponding complex input elements. The imaginary output similarly contains the imaginary parts of the input elements.



**Data Type Support** See the description above.

### Parameters and Dialog Box



### Output

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

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	No
	Vectorized	No
	Zero Crossing	No

# Configurable Subsystem

---

**Purpose** Represents any block selected from a user-specified library of blocks.

**Library** Signals & Systems

**Description** A Configurable Subsystem block can represent any block contained in a specified library of blocks. The Configurable Subsystem's dialog box lets you specify which block it represents and the values of the parameters of the represented block.



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

A Configurable Subsystem block's appearance changes depending on which block it represents. Initially, a Configurable Subsystem block represents nothing. In this state, it has no ports and displays the icon shown at the left of this paragraph. When you select a library and block, the Configurable Subsystem shows the icon and a set of input and output ports corresponding to input and output ports in the selected library.

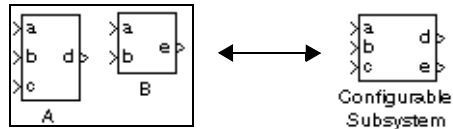
Simulink uses the following rules to map library ports to Configurable Subsystem block ports:

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

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

For example, suppose that a library contains two blocks A and B and that block A has input ports labeled a, b, and c and an output port labeled d and that block

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



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

---

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

---

## Data Type Support

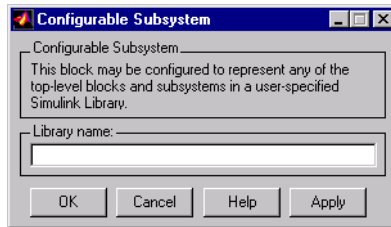
A Configurable Subsystem block accepts and outputs signals of the same types as are accepted or output by the block that it currently represents.

# Configurable Subsystem

---

## Parameters and Dialog Box

A Configurable Subsystem's dialog box changes, depending on whether the Configurable Subsystem currently represents a library and which block, if any, the Configurable Subsystem represents. Initially a Configurable Subsystem does not represent anything; its dialog box displays only an empty **Library name** parameter.



### Library name

The relative path name of the library of blocks that this Configurable Subsystem can represent, for example, simulink/Math.

To specify the library that you want the Configurable Subsystem to represent, enter the library's name in the **Library name** field.

---

**Note** You cannot use the block dialog box to change the library represented by an existing Configurable Subsystem block. You can, however, change the library by setting the block's Library parameter to the name of the new library, using the `set_param` model creation command.

---

---

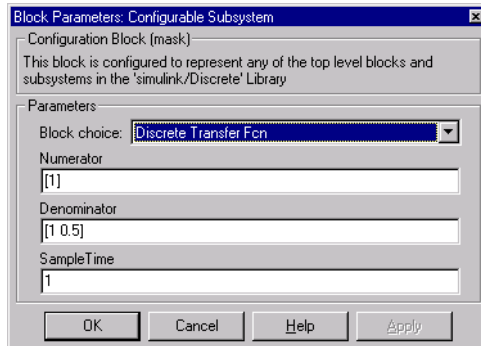
**Note** If you add or remove blocks or ports in a library, you must recreate any Configurable Subsystem blocks that use the library.

---

When you specify the library, a new set of parameters replaces the **Library name** field. The new set comprises a **Block choice** field, an **Open subsystems when selected** field, and the parameters, if any, of the block currently represented by the Configurable Subsystem. The following figure shows the



dialog box for a Configurable Subsystem block that represents the Simulink Discrete block library.



## Block choice

The block that this Configurable Subsystem block current represents.

## Open subsystems when selected

Checking this option causes Simulink to open a nonmasked block when you select it. This parameter appears only when the selected block is a masked block.

---

**Note** The other parameter fields in the dialog shown above are those of the Discrete Transfer Function block, which the Configurable Subsystem block represents in this example.

---

## Characteristics

A Configurable Subsystem block has the characteristics of the block that it currently represents.

# Constant

---

**Purpose** Generate a constant value.

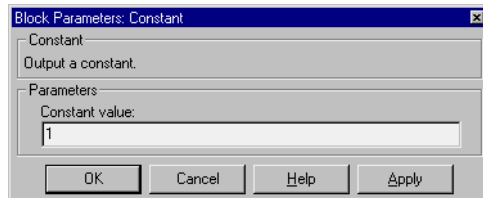
**Library** Sources

**Description** The Constant block generates a specified real or complex value independent of time. The block generates one output, which can be scalar or vector, depending on the length of the **Constant value** parameter.



**Data Type Support** A Constant block outputs a signal whose numeric type (complex or real) and data type are the same as that of the block's **Constant value** parameter.

## Parameters and Dialog Box



### Constant value

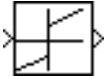
The output of the block. If a vector, the output is a vector of constants with the specified values. The default value is 1.

<b>Characteristics</b>	Sample Time	Constant
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Model discontinuity at zero, with linear gain elsewhere.

**Library** Nonlinear

**Description** The Coulomb and Viscous Friction block models Coulomb (static) and viscous (dynamic) friction. The block models a discontinuity at zero and a linear gain otherwise. The offset corresponds to the Coulombic friction; the gain corresponds to the viscous friction. The block is implemented as



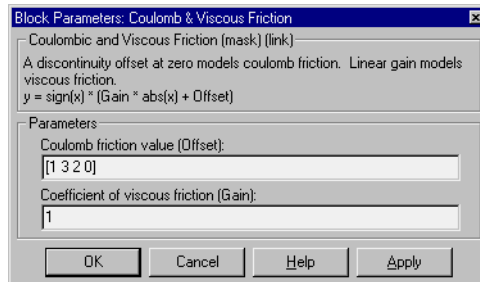
$$y = \text{sign}(u) * (\text{Gain} * \text{abs}(u) + \text{Offset})$$

where  $y$  is the output,  $u$  is the input, and  $\text{Gain}$  and  $\text{Offset}$  are block parameters.

The block accepts one input and generates one output.

**Data Type Support** A Coulomb and Viscous Friction block accepts and outputs real signals of type double.

## Parameters and Dialog Box



### Coulomb friction value

The offset, applied to all input values. The default is [1 3 2 0].

### Coefficient of viscous friction

The signal gain at nonzero input points. The default is 1.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	No

# Data Store Memory

Vectorized	Yes
Zero Crossing	Yes, at the point where the static friction is overcome

**Purpose** Define a data store.

**Library** Signals & Systems

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



Each data store must be defined by a Data Store Memory block. The location of the Data Store Memory block that defines a data store determines the Data Store Read and Data Store Write blocks that can access the data store:

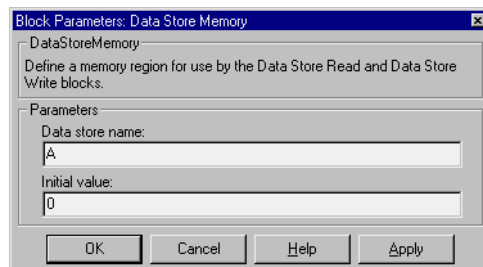
- If the Data Store Memory block is in the *top-level system*, the data store can be accessed by Data Store Read and Data Store Write blocks located anywhere in the model.
- If the Data Store Memory block is in a *subsystem*, the data store can be accessed by Data Store Read and Data Store Write blocks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

You initialize the data store by specifying values in the **Initial value** parameter. The size of the value determines the size of the data store. An error occurs if a Data Store Write block does not write the same amount of data.

## Data Type Support

A Data Store Memory block stores real signals of type double.

## Parameters and Dialog Box



**Data store name**

The name of the data store being defined. The default is A.

**Initial value**

The initial values of the data store. The default value is 0.

<b>Characteristics</b>	Sample Time	N/A
	Vectorized	Yes

# Data Store Read

---

**Purpose** Read data from a data store.

**Library** Signals & Systems

**Description** The Data Store Read block reads data from a named data store, passing the data as output. The data was previously initialized by a Data Store Memory block and (possibly) written to that data store by a Data Store Write block.

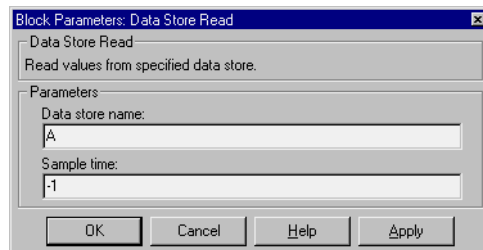


The data store from which the data is read is determined by the location of the Data Store Memory block that defines the data store. For more information, see Data Store Memory on page 8-36.

More than one Data Store Read block can read from the same data store.

**Data Type Support** A Data Store Read block outputs a real signal of type double.

## Parameters and Dialog Box



### Data store name

The name of the data store from which this block reads data.

### Sample time

The sample time, which controls when the block writes to the data store. The default, -1, indicates that the sample time is inherited.

<b>Characteristics</b>	Sample Time	Continuous or discrete
	Vectorized	Yes

**Purpose** Write data to a data store.

**Library** Signals & Systems

**Description** The Data Store Write block writes the block input to a named data store.



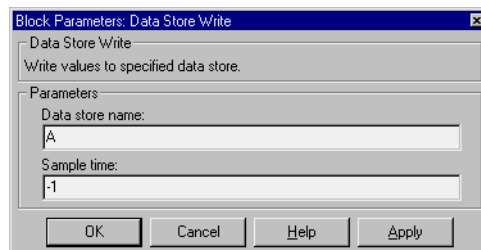
Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.

The data store to which this block writes is determined by the location of the Data Store Memory block that defines the data store. For more information, see Data Store Memory on page 8-36. The size of the data store is set by the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

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

**Data Type Support** A Data Store Write block accepts a real signal of type double.

## Parameters and Dialog Box



**Data store name** The name of the data store to which this block writes data.

**Sample time** The sample time, which controls when the block writes to the data store. The default, -1, indicates that the sample time is inherited.

# Data Store Write

---

<b>Characteristics</b>	Sample Time	Continuous or discrete
	Vectorized	Yes



**Purpose** Convert input signal to specified data type.

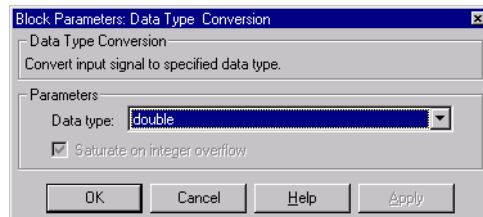
**Library** Signals & Systems

**Description** The Data Type Conversion block converts an input signal to the data type specified by the block's **Data type** parameter. The input can be any real or complex-valued signal. If the input is real, the output is real. If the input is complex, the output is complex.



**Data Type Support** See block description above.

## Parameters and Dialog Box



### Data type

Specifies the type to which to convert the input signal. The auto option converts the input signal to the type required by the input port to which the Data Type Conversion block's output port is connected.

### Saturate on integer overflow

This parameter is enable only for integer output. If selected, this option causes the output of the Data Type Conversion block to saturate on integer overflow. In particular, if the output data type is an integer type, the block output is the maximum value representable by the output type or the converted output, whichever is smaller in the absolute sense. If the option is not selected, Simulink takes the action specified by **Data overflow** event option on the **Diagnostics** page of the **Simulation Parameters** dialog box (see "The Diagnostics Page" on page 4–24).

# Data Type Conversion

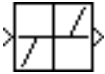
---

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of parameters
	Vectorized	Yes
	Zero Crossing	Yes, to detect when the limits are reached

**Purpose** Provide a region of zero output.

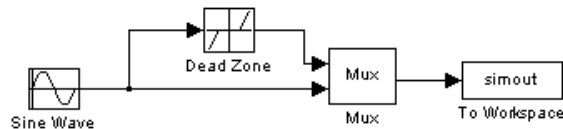
**Library** Nonlinear

**Description** The Dead Zone block generates zero output within a specified region, called its dead zone. The lower and upper limits of the dead zone are specified as the **Start of dead zone** and **End of dead zone** parameters. The block output depends on the input and dead zone:

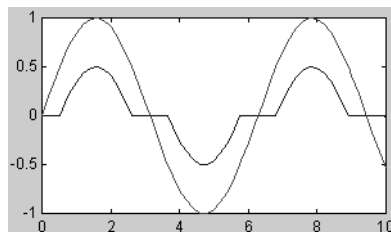


- If the input is within the dead zone (greater than the lower limit and less than the upper limit), the output is zero.
- If the input is greater than or equal to the upper limit, the output is the input minus the upper limit.
- If the input is less than or equal to the lower limit, the output is the input minus the lower limit.

This sample model uses lower and upper limits of -0.5 and +0.5, with a sine wave as input.



This plot shows the effect of the Dead Zone block on the sine wave. While the input (the sine wave) is between -0.5 and 0.5, the output is zero.



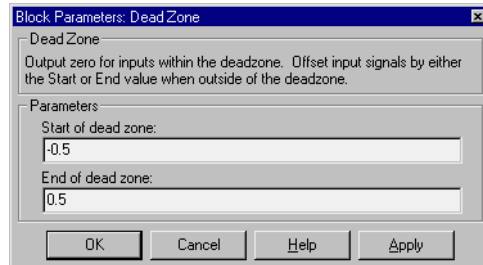
## Data Type Support

A Dead Zone block accepts and outputs a real signal of type double.

# Dead Zone

---

## Parameters and Dialog Box



### Start of dead zone

The lower limit of the dead zone. The default is -0.5.

### End of dead zone

The upper limit of the dead zone. The default is 0.5.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	Yes, to detect when the limits are reached

**Purpose** Separate a vector signal into output signals.

**Library** Signals & Systems

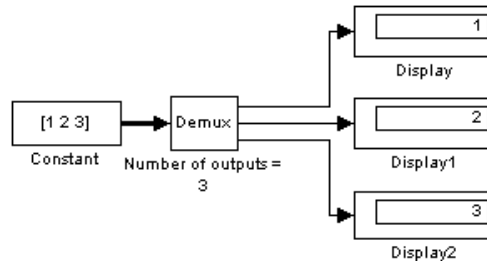
**Description** The Demux block separates a vector input signal into output lines, each of which can carry a scalar or vector signal. Simulink determines the number and widths of the output signals by the **Number of outputs** parameter.



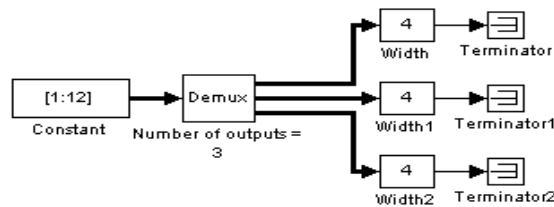
### Scalar Number of Outputs

If the **Number of outputs** parameter field contains a scalar value, the block separates the input signal into that number of output signals. The widths of the output signals depend on the width of the input vector and the number of outputs:

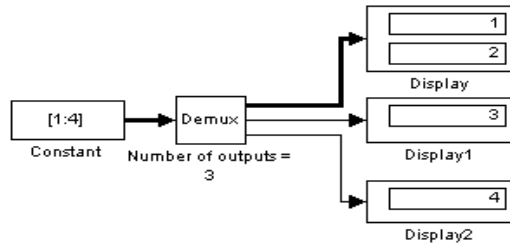
- If the input signal width is equal to the number of outputs, the block separates the input signal vector into scalar signals. In this model, the Demux block separates a three-element vector signal into three scalar signals. The **Number of outputs** parameter is 3.



- If the input signal width is evenly divisible by the number of outputs, the block separates the input signal into vector signals of equal width. In this model, the Demux block separates a 12-element vector signal into three vector signals, each with a width of four elements.



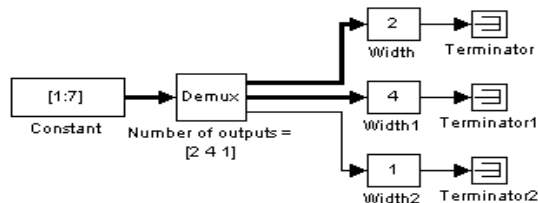
- If the input signal width is not evenly divisible by the number of outputs (and they're not the same), the block separates the input signal into vector signals of unequal width and Simulink issues a warning message. In this model, the Demux block separates a four-element vector signal into three signals. The first signal contains the first two elements of the input signal.



## Vector Number of Outputs

If the **Number of outputs** parameter is a vector, the number of output lines is equal to the number of elements in the vector. The output signal widths depend on the input vector width and the values of the elements of the parameter. You can explicitly size output signals or let Simulink determine their widths.

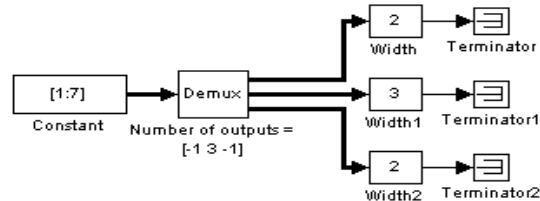
- If the **Number of outputs** vector elements are all positive values, the block generates signals with the specified widths. In this model, the input signal is a vector of width 7 and the **Number of outputs** parameter is  $[2\ 4\ 1]$ .



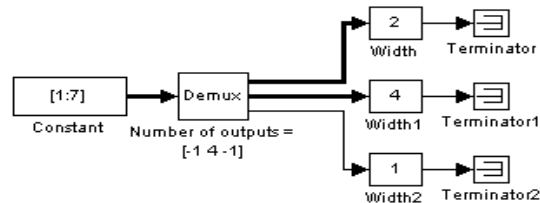
- If the **Number of outputs** vector elements include positive and  $-1$  values, the block generates output signals with the specified widths for those outputs having positive values and dynamically sizes those outputs having  $-1$  values.

In this model, the input signal is a vector of width seven and the **Number of outputs** parameter is  $[-1\ 3\ -1]$ . In this example, Simulink explicitly generates a three-element vector signal as the second output and

dynamically sizes the other two outputs by dividing the remaining input elements as evenly as possible. In this case, the four elements divide equally.



In the next example, the **Number of outputs** is specified as [-1 4 -1]. This parameter causes Simulink to generate unequal output vectors.



- If the **Number of outputs** vector elements are all -1, the number of outputs is equal to the number of vector elements, and the widths are dynamically sized. Specifying the parameter in this way is the same as specifying the parameter as a scalar whose value is the number of elements. For example, entering [-1 -1 -1] is the same as specifying a parameter value of 3.

Simulink draws the specified number of output ports on the block, resizing the block if necessary. When the number of ports is increased or decreased, ports are added or removed from the bottom of the block icon.

### Using a Variable to Provide the Number of Outputs Parameter

When you specify the **Number of outputs** parameter as a variable, Simulink issues an error message if the variable is undefined in the workspace.

---

**Note** Simulink hides the name of a bus selector block when you copy it from the Simulink library to a model.

---

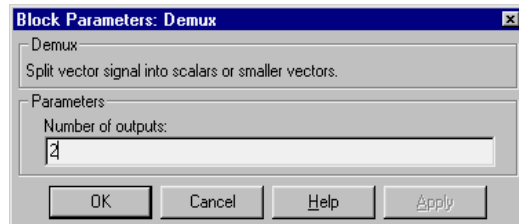
# Demux

---

## Data Type Support

A Demux block accepts and outputs signals of any numeric (complex or real) and data type.

## Parameters and Dialog Box



### Number of outputs

The number and width of outputs. The total of the output widths must match the width of the input line.



**Purpose** Output the time derivative of the input.

**Library** Continuous

**Description** The Derivative block approximates the derivative of its input by computing



$$\frac{\Delta u}{\Delta t}$$

where  $\Delta u$  is the change in input value and  $\Delta t$  is the change in time since the previous simulation time step. The block accepts one input and generates one output. The value of the input signal before the start of the simulation is assumed to be zero. The initial output for the block is zero.

The accuracy of the results depends on the size of the time steps taken in the simulation. Smaller steps allow a smoother and more accurate output curve from this block. Unlike blocks that have continuous states, the solver does not take smaller steps when the input changes rapidly.

When the input is a discrete signal, the continuous derivative of the input is an impulse when the value of the input changes, otherwise it is 0. You can obtain the discrete derivative of a discrete signal using

$$y(k) = \frac{1}{\Delta t}(u(k) - u(k-1))$$

and taking the  $z$ -transform

$$\frac{Y(z)}{u(z)} = \frac{1 - z^{-1}}{\Delta t} = \frac{z - 1}{\Delta t \cdot z}$$

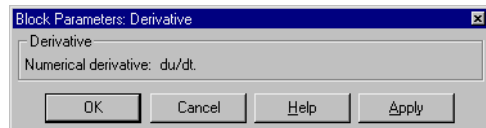
Using `linmod` to linearize a model that contains a Derivative block can be troublesome. For information about how to avoid the problem, see “Linearization” on page 5–4.

**Data Type Support** A Derivative block accepts and outputs a real signal of type double.

# Derivative

---

## Dialog Box



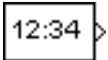
## Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	N/A
States	0
Vectorized	Yes
Zero Crossing	No

**Purpose** Output simulation time at the specified sampling interval.

**Library** Sources

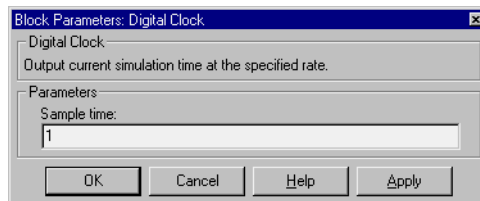
**Description** The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the output is held at the previous value.



Use this block rather than the Clock block (which outputs continuous time) when you need the current time within a discrete system.

**Data Type Support** A Digital Clock block outputs a real signal of type double.

## Parameters and Dialog Box



### Sample time

The sampling interval. The default value is 1 second.

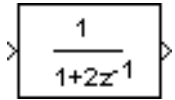
<b>Characteristics</b>	Sample Time	Discrete
	Scalar Expansion	No
	Vectorized	No
	Zero Crossing	No

# Discrete Filter

**Purpose** Implement IIR and FIR filters.

**Library** Discrete

## Description



The Discrete Filter block implements Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. You specify the coefficients of the numerator and denominator polynomials in ascending powers of  $z^{-1}$  as vectors using the **Numerator** and **Denominator** parameters. The order of the denominator must be greater than or equal to the order of the numerator. See Discrete Transfer Fcn on page 8-65 for more information about coefficients.

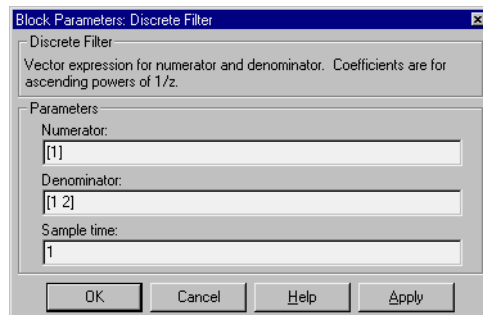
The Discrete Filter block represents the method often used by signal processing engineers, who describe digital filters using polynomials in  $z^{-1}$  (the delay operator). The Discrete Transfer Fcn block represents the method often used by control engineers, who represent a discrete system as polynomials in  $z$ . The methods are identical when the numerator and denominator are the same length. A vector of  $n$  elements describes a polynomial of degree  $n-1$ .

The block icon displays the numerator and denominator according to how they are specified. For a discussion of how Simulink displays the icon, see Transfer Fcn on page 8-203.

## Data Type Support

A Discrete Filter block accepts and outputs a real signal of type double.

## Parameters and Dialog Box



### Numerator

The vector of numerator coefficients. The default is [ 1 ].

## Denominator

The vector of denominator coefficients. The default is [1 2].

## Sample time

The time interval between samples.

<b>Characteristics</b>	Direct Feedthrough	Only if the lengths of the <b>Numerator</b> and <b>Denominator</b> parameters are equal
	Sample Time	Discrete
	Scalar Expansion	No
	States	Length of <b>Denominator</b> parameter -1
	Vectorized	No
	Zero Crossing	No

# Discrete Pulse Generator

---

**Purpose** Generate pulses at regular intervals.

**Library** Sources

**Description** The Discrete Pulse Generator block generates a series of pulses at regular intervals.

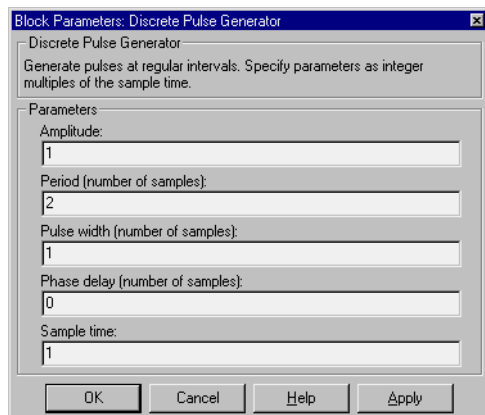


The pulse width is the number of sample periods the pulse is high. The period is the number of sample periods the pulse is high and low. The phase delay is the number of sample periods before the pulse starts. The phase delay can be positive or negative but must not be larger than the period. The sample time must be greater than zero.

Use the Discrete Pulse Generator block for discrete or hybrid systems. To generate continuous signals, use the Pulse Generator block (see Pulse Generator on page 8-146).

**Data Type Support** A Discrete Pulse Generator block accepts and outputs a real signal of type double.

## Parameters and Dialog Box



### Amplitude

The amplitude of the pulse. The default is 1.

### Period

The pulse period in number of samples. The default is 2.

**Pulse width**

The number of sample periods that the pulse is high. The default is 1.

**Phase delay**

The delay before each pulse is generated, in number of samples. The default is 0.

**Sample time**

The sample period. The default is 1.

<b>Characteristics</b>	Sample Time	Discrete
	Scalar Expansion	Of parameters
	Vectorized	Yes
	Zero Crossing	No

# Discrete State-Space

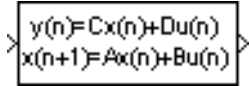
**Purpose** Implement a discrete state-space system.

**Library** Discrete

**Description** The Discrete State-Space block implements the system described by

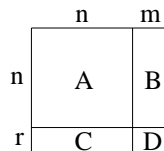
$$x(n+1) = Ax(n) + Bu(n)$$

$$y(n) = Cx(n) + Du(n)$$



where  $u$  is the input,  $x$  is the state, and  $y$  is the output. The matrix coefficients must have these characteristics, as illustrated in the diagram below:

- **A** must be an  $n$ -by- $n$  matrix, where  $n$  is the number of states.
- **B** must be an  $n$ -by- $m$  matrix, where  $m$  is the number of inputs.
- **C** must be an  $r$ -by- $n$  matrix, where  $r$  is the number of outputs.
- **D** must be an  $r$ -by- $m$  matrix.



The block accepts one input and generates one output. The input vector width is determined by the number of columns in the B and D matrices. The output vector width is determined by the number of rows in the C and D matrices.

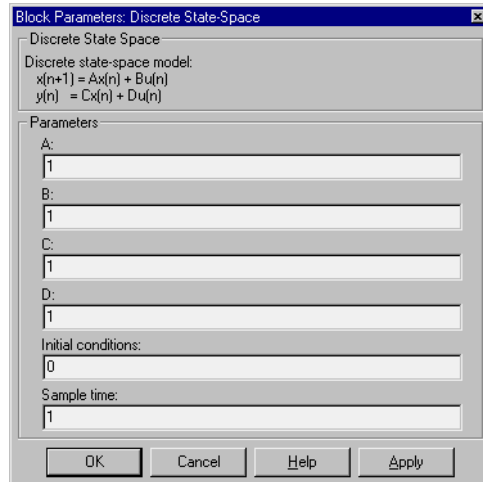
Simulink converts a matrix containing zeros to a sparse matrix for efficient multiplication.

## Data Type Support

A Discrete State Space block accepts and outputs a real signal of type `double`.



## Parameters and Dialog Box



### A, B, C, D

The matrix coefficients, as defined in the above equations.

### Initial conditions

The initial state vector. The default is 0.

### Sample time

The time interval between samples.

## Characteristics

Direct Feedthrough	Only if $D \neq 0$
Sample Time	Discrete
Scalar Expansion	Of the initial conditions
States	Determined by the size of $A$
Vectorized	Yes
Zero Crossing	No

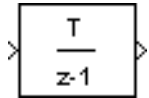
# Discrete-Time Integrator

---

**Purpose** Perform discrete-time integration of a signal.

**Library** Discrete

**Description** The Discrete-Time Integrator block can be used in place of the Integrator block when constructing a purely discrete system.



The Discrete-Time Integrator block allows you to:

- Define initial conditions on the block dialog box or as input to the block.
- Output the block state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

These features are described below.

## Integration Methods

The block can integrate using these methods: Forward Euler, Backward Euler, and Trapezoidal. For a given step  $k$ , Simulink updates  $y(k)$  and  $x(k+1)$ .  $T$  is the sampling period (delta  $T$  in the case of triggered sampling time). Values are clipped according to upper or lower limits. In all cases,  $x(0)=IC$  (clipped if necessary):

- Forward Euler method (the default), also known as Forward Rectangular, or left-hand approximation. For this method,  $1/s$  is approximated by  $T/(z-1)$ . This gives us  $y(k) = y(k-1) + T * u(k-1)$ .

Let  $x(k) = y(k)$ , then we have:

$$\begin{aligned}x(k+1) &= x(k) + T * u(k) \quad (\text{clip if necessary}) \\y(k) &= x(k)\end{aligned}$$

With this method, input port 1 does not have direct feedthrough.

- Backward Euler method, also known as Backward Rectangular or right-hand approximation. For this method,  $1/s$  is approximated by  $Tz/(z-1)$ . This gives us  $y(k) = y(k-1) + T * u(k)$ .

Let  $x(k) = y(k-1)$ , then we have:

$$\begin{aligned} x(k+1) &= y(k) \\ y(k) &= x(k) + T * u(k) \quad (\text{clip if necessary}) \end{aligned}$$

With this method, input port 1 has direct feedthrough.

- Trapezoidal method. For this method,  $1/s$  is approximated by  $T/2 * (z+1)/(z-1)$ . This gives us  $y(k) = y(k-1) + T/2 * (u(k) + u(k-1))$ .

When  $T$  is fixed (equal to the sampling period), let

$x(k) = y(k-1) + T/2 * u(k-1)$ , then we have:

$$\begin{aligned} x(k+1) &= y(k) + T/2 * u(k) \quad (\text{clip if necessary}) \\ y(k) &= x(k) + T/2 * u(k) \quad (\text{clip if necessary}) \end{aligned}$$

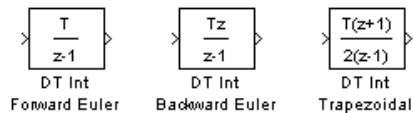
Here,  $x(k+1)$  is the best estimate of the next output. It isn't quite the state, in the sense that  $x(k) \neq y(k)$ .

When  $T$  is variable (that is, obtained from the triggering times), we have:

$$\begin{aligned} x(k+1) &= y(k) \\ y(k) &= x(k) + T/2 * (u(k) + u(k-1)) \quad (\text{clip if necessary}) \end{aligned}$$

With this method, input port 1 has direct feedthrough.

The block icon reflects the selected integration method, as this figure shows.



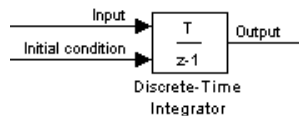
## Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

# Discrete-Time Integrator

---

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** parameter field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input, as shown in this figure.



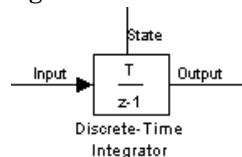
## Using the State Port

In two known situations, you must use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example of this situation, see the `bounce` model.
- When you want to pass the state from one conditionally executed subsystem to another, which may cause timing problems. For an example of this situation, see the `clutch` model.

You can correct these problems by passing the state through the state port rather than the output port. Although the values are the same, Simulink generates them at slightly different times, which protects your model from these problems. You output the block state by selecting the **Show state port** check box.

By default, the state port appears on the top of the block, as shown in this figure.



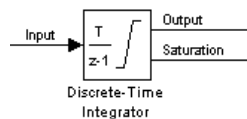
## Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. Doing so causes the block to function as a limited integrator. When the output

is outside the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than the **Lower saturation limit** and the input is negative, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than the **Upper saturation limit** and the input is positive, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port, as shown in this figure.



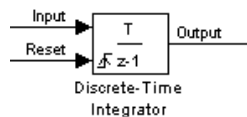
The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

When the **Limit output** option is selected, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

## Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type, as shown in this figure.



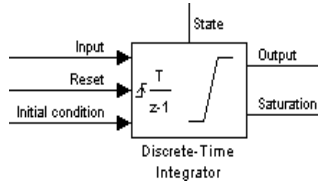
# Discrete-Time Integrator

Select **rising** to trigger the state reset when the reset signal has a rising edge. Select **falling** to trigger the state reset when the reset signal has a falling edge. Select **either** to trigger the reset when either a rising or falling signal occurs.

The reset port has direct feedthrough. If the block output is fed back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the block state into the reset port instead. To access the block's state, select the **Show state port** check box.

## Choosing All Options

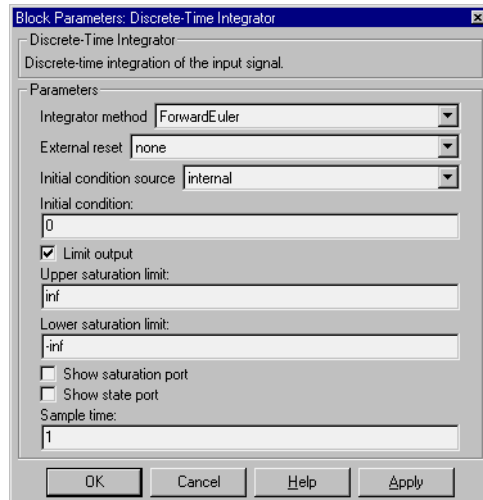
When all options are selected, the icon looks like this.



## Data Type Support

A Discrete-Time Integrator block accepts and outputs real signals of type double.

## Parameters and Dialog Box



## Integrator method

The integration method. The default is ForwardEuler.

## External reset

Resets the states to their initial conditions when a trigger event (**rising**, **falling**, or **either**) occurs in the reset signal.

## Initial condition source

Gets the states' initial conditions from the **Initial condition** parameter (if set to **internal**) or from an external block (if set to **external**).

## Initial condition

The states' initial conditions. Set the **Initial condition source** parameter value to **internal**.

## Limit output

If checked, limits the states to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

## Upper saturation limit

The upper limit for the integral. The default is `inf`.

## Lower saturation limit

The lower limit for the integral. The default is `-inf`.

## Show saturation port

If checked, adds a saturation output port to the block.

## Show state port

If checked, adds an output port to the block for the block's state.

## Sample time

The time interval between samples. The default is 1.

## Characteristics

Direct Feedthrough	Yes, of the reset and external initial condition source ports
Sample Time	Discrete
Scalar Expansion	Of parameters
States	Inherited from driving block and parameter

# Discrete-Time Integrator

---

Vectorized

Yes

Zero Crossing

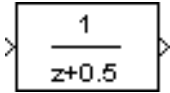
One for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation



**Purpose** Implement a discrete transfer function.

**Library** Discrete

**Description** The Discrete Transfer Fcn block implements the  $z$ -transform transfer function described by the following equations



$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^n + num_1 z^{n-1} + \dots + num_m z^{n-m}}{den_0 z^n + den_1 z^{n-1} + \dots + den_n}$$

where  $m+1$  and  $n+1$  are the number of numerator and denominator coefficients, respectively.  $num$  and  $den$  contain the coefficients of the numerator and denominator in descending powers of  $z$ .  $num$  can be a vector or matrix,  $den$  must be a vector, and both are specified as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

Block input is scalar; output width is equal to the number of rows in the numerator.

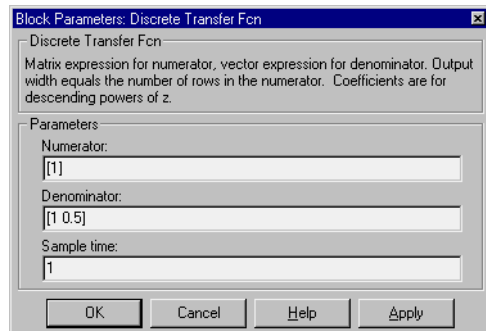
The Discrete Transfer Fcn block represents the method typically used by control engineers, representing discrete systems as polynomials in  $z$ . The Discrete Filter block represents the method typically used by signal processing engineers, who describe digital filters using polynomials in  $z^{-1}$  (the delay operator). The two methods are identical when the numerator is the same length as the denominator.

The Discrete Transfer Fcn block displays the numerator and denominator within its icon depending on how they are specified. See Transfer Fcn on page 8-203 for more information.

**Data Type Support** A Discrete Transfer Function block accepts and outputs real signals of type double.

# Discrete Transfer Fcn

## Parameters and Dialog Box



### Numerator

The row vector of numerator coefficients. A matrix with multiple rows can be specified to generate multiple output. The default is [ 1 ].

### Denominator

The row vector of denominator coefficients. The default is [ 1 0.5 ].

### Sample time

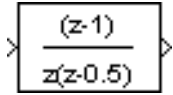
The time interval between samples. The default is 1.

<b>Characteristics</b>	Direct Feedthrough	Only if the lengths of the <b>Numerator</b> and <b>Denominator</b> parameters are equal
	Sample Time	Discrete
	Scalar Expansion	No
	States	Length of <b>Denominator</b> parameter -1
	Vectorized	No
	Zero Crossing	No

**Purpose** Implement a discrete transfer function specified in terms of poles and zeros.

**Library** Discrete

**Description** The Discrete Zero-Pole block implements a discrete system with the specified zeros, poles, and gain in terms of the delay operator  $z$ . A transfer function can be expressed in factored or zero-pole-gain form, which, for a single-input, single-output system in MATLAB, is



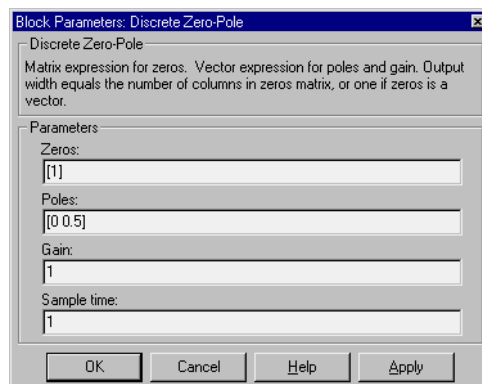
$$H(z) = K \frac{Z(z)}{P(z)} = K \frac{(z - Z_1)(z - Z_2) \dots (z - Z_m)}{(z - P_1)(z - P_2) \dots (z - P_n)}$$

where  $Z$  represents the zeros vector,  $P$  the poles vector, and  $K$  the gain. The number of poles must be greater than or equal to the number of zeros ( $n \geq m$ ). If the poles and zeros are complex, they must be complex conjugate pairs.

The block icon displays the transfer function depending on how the parameters are specified. See Zero-Pole on page 8-222 for more information.

**Data Type Support** A Discrete Zero-Pole block accepts and outputs real signals of type double.

## Parameters and Dialog Box



### Zeros

The matrix of zeros. The default is [1].

# Discrete Zero-Pole

---

## Poles

The vector of poles. The default is [0 0.5].

## Gain

The gain. The default is 1.

## Sample time

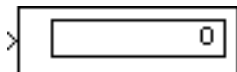
The time interval between samples.

<b>Characteristics</b>	Direct Feedthrough	Yes, if the number of zeros and poles are equal
	Sample Time	Discrete
	Scalar Expansion	No
	States	Length of <b>Poles</b> vector
	Vectorized	No
	Zero Crossing	No

**Purpose** Show the value of the input.

**Library** Sinks

**Description** The Display block shows the value of its input.



You can control the display format by selecting a **Format** choice:

- **short**, which displays a 5-digit scaled value with fixed decimal point
- **long**, which displays a 15-digit scaled value with fixed decimal point
- **short\_e**, which displays a 5-digit value with a floating decimal point
- **long\_e**, which displays a 16-digit value with a floating decimal point
- **bank**, which displays a value in fixed dollars and cents format (but with no \$ or commas)

To use the block as a floating display, select the **Floating display** check box. The block's input port disappears and the block displays the value of the signal on a selected line. If you select the **Floating display** option, you must turn off Simulink's buffer reuse feature. See "Disable optimized I/O storage" on page 4-25 for more information.

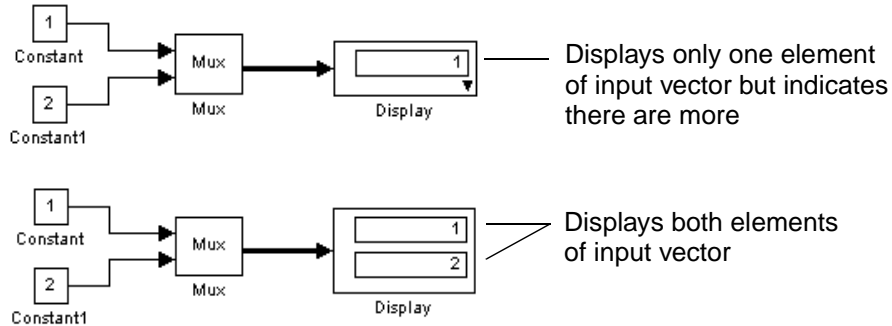
The amount of data displayed and the time steps at which the data is displayed are determined by block parameters:

- The **Decimation** parameter enables you to display data at every  $n$ th sample, where  $n$  is the decimation factor. The default decimation, 1, displays data at every time step.
- The **Sample time** parameter enables you to specify a sampling interval at which to display points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of  $-1$  causes the block to ignore sampling interval when determining which points to display.

If the block input is a vector, you can resize the block to show more than just the first element. You can resize the block vertically or horizontally; the block adds display fields in the appropriate direction. A black triangle indicates that the block is not displaying all input vector elements. For example, the figure below shows a model that passes a vector to a Display block. The top model

# Display

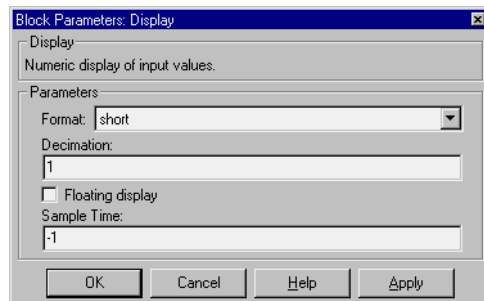
shows the block before it is resized; notice the black triangle. The bottom model shows the resized block displaying both input elements.



## Data Type Support

A Display block accepts and outputs real or complex signals of any data type.

## Parameters and Dialog Box



### Format

The format of the data displayed. The default is short.

### Decimation

How often to display data. The default value, 1, displays every input point.

### Floating display

If checked, the block's input port disappears, which enables the block to be used as a floating Display block.

### Sample time

The sample time at which to display points.

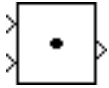
<b>Characteristics</b>	Sample Time	Inherited from driving block
	Vectorized	Yes

# Dot Product

**Purpose** Generate the dot product.

**Library** Math

**Description** The Dot Product block generates the dot product of its two input vectors. The scalar output,  $y$ , is equal to the MATLAB operation



$$y = u1' * u2$$

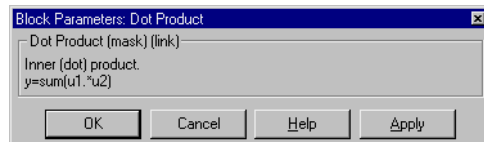
where  $u1$  and  $u2$  represent the vector inputs. If both inputs are vectors, they must be the same length. The elements of the input vectors may be real- or complex-valued signals of data type `double`. The signal type (complex or real) of the output depends on the signal types of the inputs.

Input 1	Input 2	Output
real	real	real
real	complex	complex
complex	real	complex
complex	complex	complex

To perform element-by-element multiplication without summing, use the Product block.

**Data Type Support** A Dot Product block accepts and outputs signals of type `double`.

## Dialog Box



**Characteristics**

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes



States	0
Vectorized	Yes
Zero Crossing	No

# Enable

---

**Purpose** Add an enabling port to a subsystem.

**Library** Signals & Systems

**Description** Adding an Enable block to a subsystem makes it an enabled subsystem. An enabled subsystem executes while the input received at the Enable port is greater than zero.



At the start of simulation, Simulink initializes the states of blocks inside an enabled subsystem to their initial conditions. When an enabled subsystem restarts (executes after having been disabled), the **States when enabling** parameter determines what happens to the states of blocks contained in the enabled subsystem:

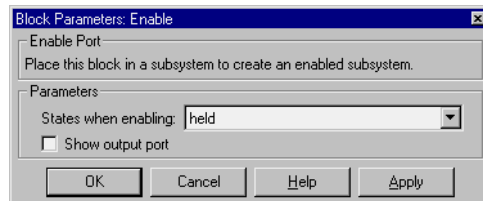
- **reset** resets the states to their initial conditions (zero if not defined).
- **held** holds the states at their previous values.

You can output the enabling signal by selecting the **Show output port** check box. Selecting this option allows the system to process the enabling signal. The width of the signal is the width of the enabling signal.

A subsystem can contain no more than one Enable block.

**Data Type Support** The data type of the input of the Enable port may be boolean or double. See Chapter 7, “Conditionally Executed Subsystems” for more information about enabled subsystems.

## Parameters and Dialog Box



### States when enabling

Specifies how to handle internal states when the subsystem becomes re-enabled.

## Show output port

If checked, Simulink draws the Enable block output port and outputs the enabling signal.

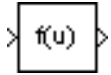
<b>Characteristics</b>	Sample Time	Determined by the signal at the enable port
	Vectorized	Yes

# Fcn

**Purpose** Apply a specified expression to the input.

**Library** Functions & Tables

**Description** The Fcn block applies the specified C language style expression to its input. The expression can be made up of one or more of these components:



- $u$  — the input to the block. If  $u$  is a vector,  $u(i)$  represents the  $i$ th element of the vector;  $u(1)$  or  $u$  alone represents the first element.
- Numeric constants
- Arithmetic operators (+ - \* /)
- Relational operators (== != > < >= <=) — The expression returns 1 if the relation is TRUE; otherwise, it returns 0.
- Logical operators (&& || !) — The expression returns 1 if the relation is TRUE; otherwise, it returns 0.
- Parentheses
- Mathematical functions — abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, hypot, ln, log, log10, pow, power, rem, sgn, sin, sinh, sqrt, tan, and tanh.
- Workspace variables — Variable names that are not recognized in the list of items above are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g.,  $A(1,1)$  instead of  $A$  for the first element in the matrix).

The rules of precedence obey the C language standards:

- 1 ( )
- 2 + - (unary)
- 3 pow (exponentiation)
- 4 !
- 5 \* /
- 6 + -
- 7 > < <= >=
- 8 = !=
- 9 &&
- 10 ||

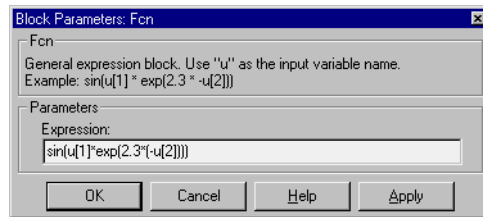
The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the Math Function block. If a block is a vector and the function operates on input elements individually (for example, the `sin` function), the block operates on only the first vector element.

## Data Type Support

A Fcn block accepts and outputs signals of type double.

## Parameters and Dialog Box



## Expression

The C language style expression applied to the input. Expression components are listed above. The expression must be mathematically well formed (i.e., matched parentheses, proper number of function arguments, etc.).

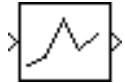
Characteristics	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	No
	Vectorized	No
	Zero Crossing	No

# First-Order Hold

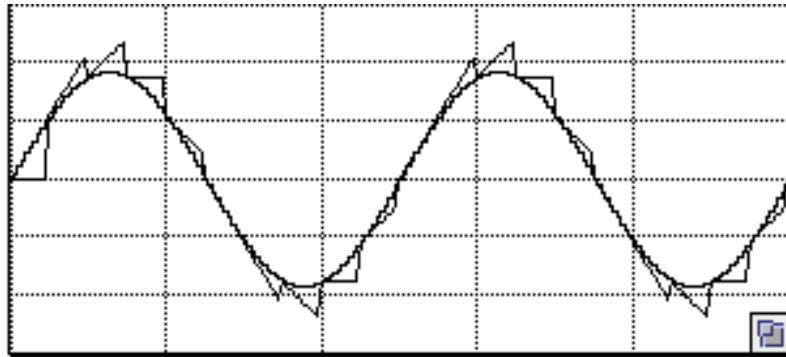
**Purpose** Implement a first-order sample-and-hold.

**Library** Discrete

**Description** The First-Order Hold block implements a first-order sample-and-hold that operates at the specified sampling interval. This block has little value in practical applications and is included primarily for academic purposes.

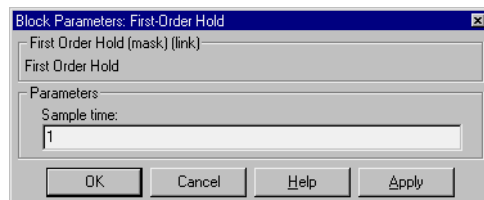


You can see the difference between the Zero-Order Hold and First-Order Hold blocks by running the demo program fohdemo. This figure compares the output from a Sine Wave block and a First-Order Hold block.



**Data Type Support** A First-Order Hold block accepts and outputs signals of type double.

## Parameters and Dialog Box



**Sample time** The time interval between samples.

<b>Characteristics</b>	Direct Feedthrough	No
	Sample Time	Continuous
	Scalar Expansion	No
	States	1 continuous and 1 discrete per input element
	Vectorized	Yes
	Zero Crossing	No

# From

**Purpose** Accept input from a Goto block.

**Library** Signals & Systems

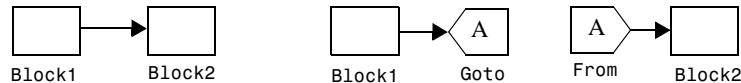
## Description



The From block accepts a signal from a corresponding Goto block, then passes it as output. The data type of the output is the same as that of the input from the Goto block. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. To associate a Goto block with a From block, enter the Goto block's tag in the **Goto tag** parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.



Associated Goto and From blocks can appear anywhere in a model with this exception: if either block is in a conditionally executed subsystem, the other block must be either in the same subsystem or in a subsystem below it in the model hierarchy (but not in another conditionally executed subsystem).

However, if a Goto block is connected to a state port, the signal can be sent to a From block inside another conditionally executed subsystem. For more information about conditionally executed subsystems, see Chapter 7.

The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see Goto on page 8-91, and Goto Tag Visibility on page 8-94. The block icon indicates the visibility of the Goto block tag:

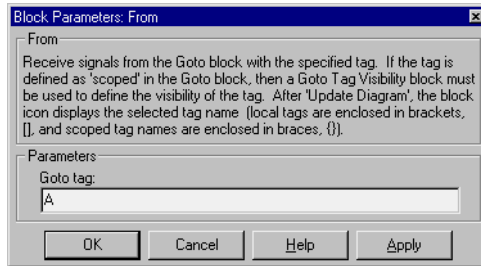
- A local tag name is enclosed in square brackets ([]).
- A scoped tag name is enclosed in braces ({}).
- A global tag name appears without additional characters.



## Data Type Support

A From block outputs signals of any real or complex data type.

## Parameters and Dialog Box



### Goto tag

The tag of the Goto block passing the signal to this From block.

## Characteristics

Sample Time

Inherited from block driving the Goto block

Vectorized

Yes

# From File

---

**Purpose** Read data from a file.

**Library** Sources

**Description** The From File block outputs data read from the specified file. The block icon displays the name of the file supplying the data.

untitled.mat

The file must contain a matrix of two or more rows. The first row must contain monotonically increasing time points. Other rows contain data points that correspond to the time point in that column. The matrix is expected to have this form.

$$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u_{1_1} & u_{1_2} & \dots & u_{1_{final}} \\ \dots & & & \\ u_{n_1} & u_{n_2} & \dots & u_{n_{final}} \end{bmatrix}$$

The width of the output depends on the number of rows in the file. The block uses the time data to determine its output, but does not output the time values. This means that in a matrix containing  $m$  rows, the block outputs a vector of length  $m-1$ , consisting of data from all but the first row of the appropriate column.

If an output value is needed at a time that falls between two values in the file, the value is linearly interpolated between the appropriate values. If the required time is less than the first time value or greater than the last time value in the file, Simulink extrapolates using the first two or last two points to compute a value.

If the matrix includes two or more columns at the same time value, the output is the data point for the first column encountered. For example, for a matrix that has this data:

```
time values:    0 1 2 2
data points:   2 3 4 5
```

At time 2, the output is 4, the data point for the first column encountered at that time value.

Simulink reads the file into memory at the start of the simulation. As a result, you cannot read data from the same file named in a To File block in the same model.

## Using Data Saved by a To File or a To Workspace Block

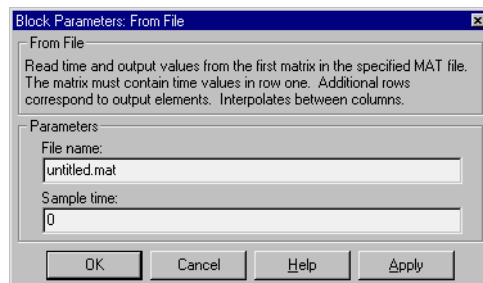
The From File block can read data written by a To File block without any modifications. To read data written by a To Workspace block and saved to a file:

- The data must include the simulation times. The easiest way to include time data in the simulation output is to specify a variable for time on the **Workspace I/O** page of the **Simulation Parameters** dialog box. See Chapter 4 for more information.
- The form of the data as it is written to the workspace is different from the form expected by the From File block. Before saving the data to a file, transpose it. When it is read by the From File block, it will be in the correct form.

## Data Type Support

A From File block outputs real signals of type double.

## Parameters and Dialog Box



### File name

The name of the file that contains the data used as input. The default file name is `untitled.mat`.

### Sample time

Sample rate of data read from the file.

# From File

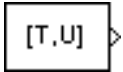
---

<b>Characteristics</b>	Sample Time	Inherited from driven block
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Read data from the workspace.

**Library** Sources

**Description**



The From Workspace block reads data from the MATLAB workspace. The block’s **Data** parameter specifies the workspace data via a MATLAB expression that evaluates to a matrix or structure containing a table of signal values and time steps. The format of the matrix or structure is the same as that used to load input data from the workspace (see “Loading Input from the Base Workspace” on page 4–17). The From Workspace icon displays the expression in the **Data** parameter.

If the input table does not specify the times of the input data values, each value is assumed to occur at  $t = (n-1) * st$  where  $n$  is the  $n$ th input value and  $st$  is the block’s sample time.

The output of a From Workspace block at each time step depends on the settings of the block’s **Interpolate data** and **Hold final data value** parameters. The following table summarizes the output for the various combinations of parameter settings.

Intrp. Option	Hold Option	Block Output $t_i < t < t_f$	Block Output $t > t_f$
On	Off	Interpolated between data values	Extrapolated from final data value
On	On	Interpolated between data values	Final data value
Off	Off	Most recent data value	Zero
Off	On	Most recent data value	Final data value

If the input table contains more than one entry for the same time step, Simulink uses the signals specified by the last entry. For example, suppose the input table has this data:

```
time:    0 1 2 2
signal:  2 3 4 5
```

# From Workspace

At time 2, the output is 5, the signal value for the last entry for time 2.

---

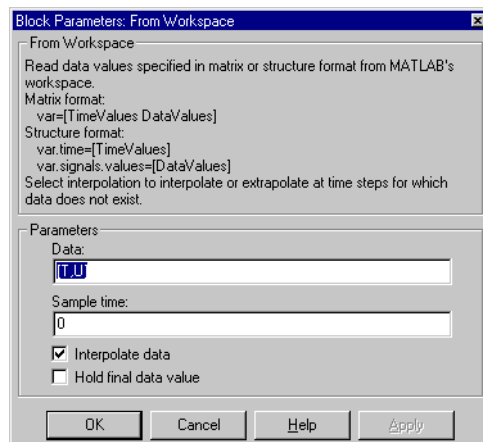
**Note** A From Workspace block can directly read the output of a To Workspace block (see To Workspace on page 8-199) if the output is in structure or structure-with-time format (see “Loading Input from the Base Workspace” on page 4–17 for a description of these formats). To read a matrix written by a To Workspace block requires that a time column be added to the matrix.

---

## Data Type Support

A From Workspace block can output a real or complex signal of any data type.

## Parameters and Dialog Box



## Data

An expression that evaluates to a matrix or a structure containing a table of simulation times and corresponding signal values. For example, suppose that the workspace contains a column vector of times named *T* and a matrix of corresponding signal values named *U*. Then the default expression for this parameter, `[ T , U ]`, yields a matrix containing the required input table. If the required signal-versus-time matrix or structure already exists in the workspace, simply enter the name of the structure or matrix in this field.

**Sample time**

Sample rate of data from workspace.

**Interpolate data**

This option causes the block to linearly interpolate (or extrapolate, if the **Hold final data value** parameter is off) at time steps for which no corresponding workspace data exists. Otherwise, the current output equals the output at the most recent time for which data exists.

**Hold final data value**

This option causes the block to hold its output to the last value for which data is available.

<b>Characteristics</b>	Sample Time	Inherited from driven block
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No

# Function-Call Generator

---

**Purpose** Execute a function-call subsystem at a specified rate.

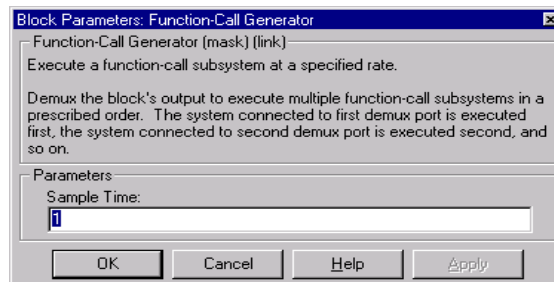
**Library** Signals & Systems

**Description** The Function-Call Generator block executes a function-call subsystem (for example, a Stateflow state chart configured as a function-call system) at the rate specified by the block's **Sample time** parameter. To execute multiple function-call subsystems in a prescribed order, first connect a Function-Call Generator block to a Demux block that has as many output ports as there are function-call subsystems to be controlled. Then connect the outputs of the Demux block to the systems to be controlled. The system connected to the first demux port executes first, the system connected to the second demux port executes second, and so on.



**Data Type Support** A Function-Call block outputs a real signal of type double.

## Parameters and Dialog Box



**Sample time** The time interval between samples.

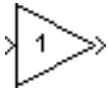
<b>Characteristics</b>	Direct Feedthrough	No
	Sample Time	User-specified
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No



**Purpose** Multiply block input.

**Library** Math

**Description**



The Gain block generates its output by multiplying its input by a specified constant, variable, or expression. You can enter the gain as a numeric value, or as a variable or expression. To multiply the input by a matrix, use the Matrix Gain block (see Matrix Gain on page 8-123).

The Gain block icon displays the value entered in the **Gain** parameter field if the block is large enough. If the gain is specified as a variable, the block displays the variable name, although if the variable is specified in parentheses, the block evaluates the variable each time the block is redrawn and displays its value. If the **Gain** parameter value is too long to be displayed in the block, the string –K– is displayed.

To modify the gain during a simulation using a slider control (see Slider Gain on page 8-183).

**Data Type Support**

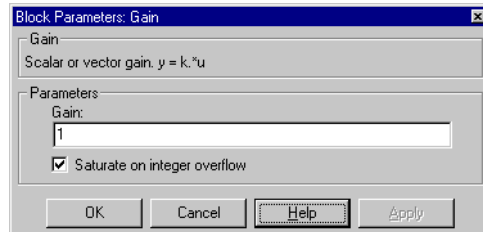
A Gain block accepts a real- or complex-valued scalar or vector of any data type except boolean and outputs a signal of the same data type as its input. The elements of an input vector must be of the same type. A Gain block's Gain parameter can also be a real- or complex-valued scalar or vector of any data type. A Gain block observes the following type rules:

- If the input is real and the gain is complex, the output is complex.
- If the gain parameter's data type differs from the input signal's data type and the input data type can represent the gain, Simulink converts the gain to the input type before computing the output. Otherwise, Simulink halts the simulation and signals an error. For example, if the input data type is `uint8` and the gain is `-1`, an error results. If typecasting the gain parameter to the input data type results in a loss of precision, Simulink issues a warning and continues the simulation.
- If the output data type is an integer type and the gain block's **Saturate on integer overflow** option is selected, the block saturates if the output exceeds the maximum value representable by the block's output data type. In other words, the block outputs one plus the maximum positive or minimum negative value representable by the output data type. For example, if the

# Gain

output type is `int8`, the actual output is 128 if the computed output is greater than 128 and -128 if the computed output is less than -128.

## Parameters and Dialog Box



## Gain

The gain, specified as a scalar, vector, variable name, or expression. The default is 1. If not specified, the data type of the **Gain** parameter is `double`.

## Saturate on integer overflow

If selected, this option causes the output of the Gain block to saturate on integer overflow. In particular, if the output data type is an integer type, the block output is the maximum value representable by the output type or the computed output, whichever is smaller in the absolute sense. If the option is not selected, Simulink takes that action specified by the **Data overflow** event option on the **Diagnostics** page of the **Simulation Parameters** dialog box (see “The Diagnostics Page” on page 4–24).

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of input and <b>Gain</b> parameter
States	0
Vectorized	Yes
Zero Crossing	No

**Purpose** Pass block input to From blocks.

**Library** Signals & Systems

**Description**



The Goto block passes its input to its corresponding From blocks. The input can be a real- or complex-valued signal or vector of any data type. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them.

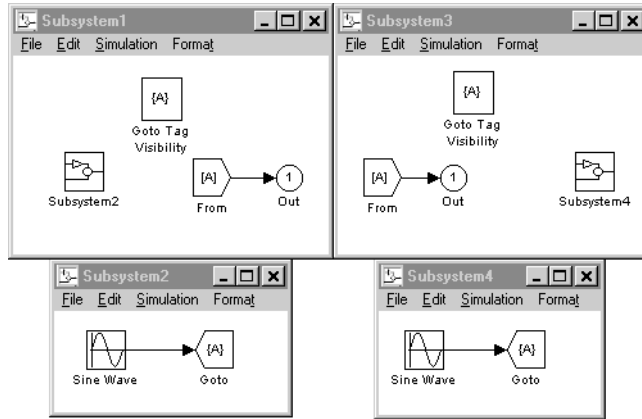
A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. For limitations on the use of From and Goto blocks, see From on page 8-80. Goto blocks and From blocks are matched by the use of Goto tags, defined as the **Tag** parameter.

The **Tag visibility** parameter determines whether the location of From blocks that access the signal is limited:

- **local**, the default, means that From and Goto blocks using the tag must be in the same subsystem. A local tag name is enclosed in square brackets ([]).
- **scoped** means that From and Goto blocks using the same tag must be in the same subsystem or in any subsystem below the Goto Tag Visibility block in the model hierarchy. A scoped tag name is enclosed in braces ({}).
- **global** means that From and Goto blocks using the same tag can be anywhere in the model.

Use local tags when the Goto and From blocks using the same tag name reside in the same subsystem. You must use global or scoped tags when the Goto and From blocks using the same tag name reside in different subsystems. When you define a tag as global, all uses of that tag access the same signal. A tag

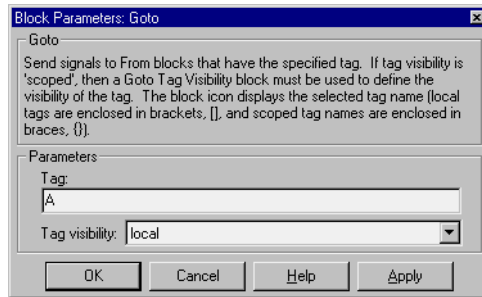
defined as scoped can be used in more than one place in the model. This example shows a model that uses two scoped tags with the same name (A).



## Data Type Support

A Goto block accepts real or complex signals of any data type.

## Parameters and Dialog Box



## Tag

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

## Tag visibility

The scope of the Goto block tag: **local**, **scoped**, or **global**. The default is **local**.

<b>Characteristics</b>	Sample Time	Inherited from driving block
	Vectorized	Yes

# Goto Tag Visibility

**Purpose** Define scope of Goto block tag.

**Library** Signals & Systems

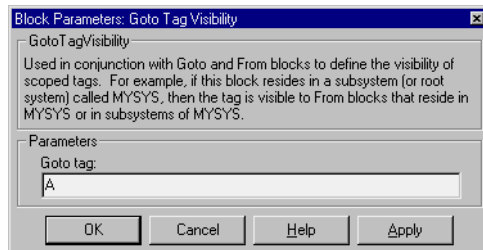
**Description** The Goto Tag Visibility block defines the accessibility of Goto block tags that have **scoped** visibility. The tag specified as the **Goto tag** parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.



A Goto Tag Visibility block is required for Goto blocks whose **Tag visibility** parameter value is **scoped**. It is not used if the tag visibility is either **local** or **global**. The block icon shows the tag name enclosed in braces ({}).

**Data Type Support** Not applicable.

## Parameters and Dialog Box



### Goto tag

The Goto block tag whose visibility is defined by the location of this block.

**Characteristics** Sample Time N/A

Vectorized N/A

**Purpose** Ground an unconnected input port.

**Library** Signals & Systems

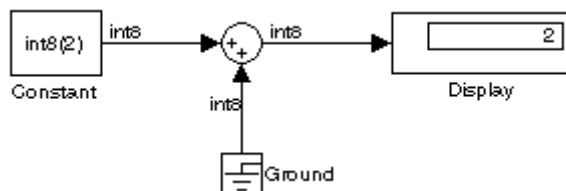
**Description**



The Ground block can be used to connect blocks whose input ports are not connected to other blocks. If you run a simulation with blocks having unconnected input ports, Simulink issues warning messages. Using Ground blocks to “ground” those blocks avoids warning messages. The Ground block outputs a signal with zero value. The data type of the signal is the same as that of the port to which it is connected.

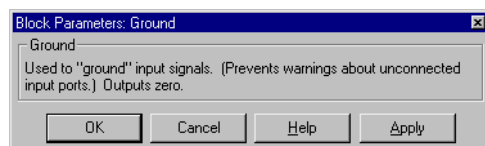
**Data Type Support**

A Ground block outputs a signal of the same numeric type (real or complex) and data type as the port to which it is connected. For example, consider the following model.



In this example, the output of the constant block determines the data type (int8) of the port to which the ground block is connected. That port in turn determines the type of the signal output by the ground block.

**Parameters and Dialog Box**



<b>Characteristics</b>	Sample Time	Inherited from driven block
	Vectorized	Yes

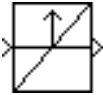
# Hit Crossing

---

**Purpose** Detect crossing point.

**Library** Signals & Systems

## Description



The Hit Crossing block detects when the input reaches the **Hit crossing offset** parameter value in the direction specified by the **Hit crossing direction** parameter. This block locates transitions to, from, and through the offset. The block finds the crossing point to within machine tolerance.

The block accepts one input of type `double`. If the **Show output port** check box is selected, the block output indicates when the crossing occurs. If the input signal is exactly the value of the offset value, the block outputs a value of 1 at that time step. If the input signals at two adjacent points bracket the offset value (but neither value is exactly equal to the offset), the block outputs a value of 1 at the second time step. If the **Show output port** check box is *not* selected, the block ensures that the simulation finds the crossing point but does not generate output.

The Hit Crossing block serves as an “Almost Equal” block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block may be more convenient than adding logic to your model to detect this condition.

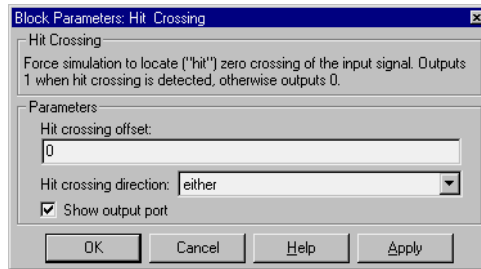
The `hardstop` and `clutch` demos illustrate the use of the Hit Crossing block. In the `hardstop` demo, the Hit Crossing block is in the Friction Model subsystem. In the `clutch` demo, the Hit Crossing block is in the Lockup Detection subsystem.

## Data Type Support

A Hit Crossing block outputs a signal of type `boolean` unless `boolean compatibility mode` is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45) in which case, the block outputs a signal of type `double`.



## Parameters and Dialog Box



### Hit crossing offset

The value whose crossing is to be detected.

### Hit crossing direction

The direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

### Show output port

If checked, draw an output port.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Yes
	Vectorized	Yes
	Zero Crossing	Yes, to detect the crossing

# IC

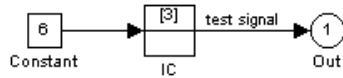
**Purpose** Set the initial value of a signal.

**Library** Signals & Systems

**Description** The IC block sets the initial condition of the signal connected to its output port.



For example, these blocks illustrate how the IC block initializes a signal labeled “test signal.”

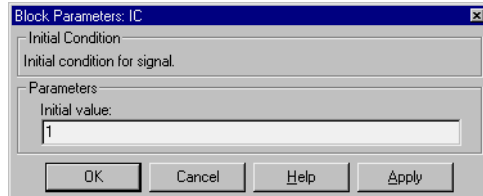


At  $t = 0$ , the signal value is 3. Afterwards, the signal value is 6.

The IC block is also useful in providing an initial guess for the algebraic state variables in the loop. For more information, see Chapter 10.

**Data Type Support** A IC block accepts and outputs a signal of type double.

## Dialog Box



### Initial value

The initial value for the signal. The default is 1.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Parameter only
	States	0
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Create an input port for a subsystem or an external input.

**Library** Signals & Systems

**Description** Inports are the links from outside a system into the system.



Simulink assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, it is assigned the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is *not* renumbered unless its current number conflicts with an Inport block already in the system. If the copied Inport block port number is not in sequence, you must renumber the block or you will get an error message when you run the simulation or update the block diagram.

If the Inport block provides a vector signal, you can specify the width of the input to the Inport block as the **Port width** parameter or let Simulink determine it automatically by providing a value of -1 (the default).

The **Sample time** parameter is the rate at which the signal is coming into the system. The default (-1) causes the block to inherit its sample time from the block driving it. It may be appropriate to set this parameter for Inport blocks in the top-level system or in models where Inport blocks are driven by blocks whose sample time cannot be determined.

## Inport Blocks in a Subsystem

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem.

The Inport block associated with an input port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose **Port**

**number** parameter is 1 gets its signal from the block connected to the top-most port on the Subsystem block.

If you renumber the **Port number** of an Inport block, the block becomes connected to a different input port, although the block continues to receive its signal from the same block outside the subsystem.

The Inport block name appears in the Subsystem block icon as a port label. To suppress display of the label, select the Inport block and choose **Hide Name** from the **Format** menu. Then, choose **Update Diagram** from the **Edit** menu.

## Inport Blocks in a Top-Level System

Inport blocks in a top-level system have two uses: to supply external inputs from the workspace, which you can do by using either the **Simulation Parameters** dialog box or the `sim` command, and to provide a means for analysis functions to perturb the model:

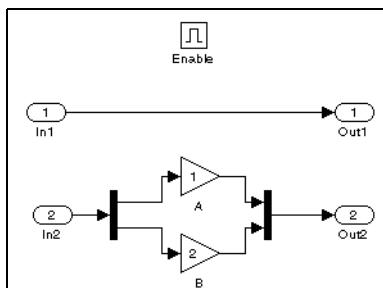
- To supply external inputs from the workspace, using the Simulation Parameters dialog (see “Loading Input from the Base Workspace” on page 4-17) or the `ut` argument of the `sim` command (see `sim` on page 4-30).
- To provide a means for perturbation of the model by the `linmod` and `trim` analysis functions. Inport blocks define the points where inputs are injected into the system. For information about using Inport blocks with analysis commands, see Chapter 5.

## Data and Numeric Type Support

An inport accepts real- or complex-valued signals of any data type. The data type and numeric type of the output of an inport is the same as that of the corresponding input signal. You must specify the signal type and data type of an external (i.e., workspace) input to a root-level inport, using the inport’s **Signal type** and **Data type** parameters.

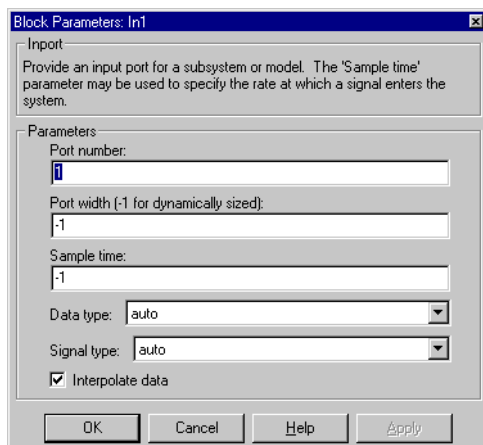
The elements of a signal vector connected to a root-level inport must be of the same numeric type and data type. Signal elements connected to a subsystem inport may be of differing numeric and data types except in one instance. If the subsystem contains an Enable or Trigger block and the inport is connected

directly to an output, the input elements must be of the same type. For example, consider the follow enabled subsystem.



In this example, the elements of a signal vector connected to In1 must be of the same type. The elements connected to In2, however, may be of differing types.

## Parameters and Dialog Box



### Port number

The port number of the Inport block.

### Port width

The width of the input signal to the Inport. Specify -1 to have it automatically determined.

### Sample time

The rate at which the signal is coming into the system.

# Inport

---

---

**Note** The next three parameters apply only to root-level inports. They do not appear on subsystem inport dialogs.

---

**Data type**

The data type of the external input.

**Signal type**

The signal type (real or complex) of the external input.

**Interpolate data**

Selecting this option causes this block to interpolate or extrapolate output at time steps for which no corresponding workspace data exists.

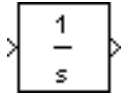
**Characteristics**

Sample Time	Inherited from driving block
Vectorized	Yes

**Purpose** Integrate a signal.

**Library** Continuous

**Description** The Integrator block integrates its input. The output of the integrator is simply its state, the integral. The Integrator block allows you to:



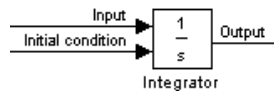
- Define initial conditions on the block dialog box or as input to the block.
- Output the block state.
- Define upper and lower limits on the integral.
- Reset the state depending on an additional reset input.

Use the Discrete-Time Integrator block (see Discrete-Time Integrator on page 8-58), when constructing a purely discrete system.

## Defining Initial Conditions

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** parameter field.
- To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input, as shown in this figure.



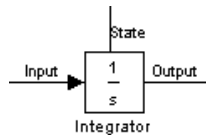
## Using the State Port

In two known situations, you must use the state port instead of the output port:

- When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example of this situation, see the bounce model.
- When you want to pass the state from one conditionally executed subsystem to another, which may cause timing problems. For an example of this situation, see the clutch model.

# Integrator

You can correct these problems by passing the state through the state port rather than the output port. Although the values are the same, Simulink generates them at slightly different times, which protects your model from these problems. You output the block state by selecting the **Show state port** check box. By default, the state port appears on the top of the block, as shown in this figure.

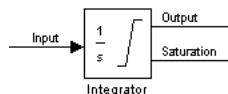


## Limiting the Integral

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. Doing so causes the block to function as a limited integrator. When the output is outside the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than the **Lower saturation limit** and the input is negative, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than the **Upper saturation limit** and the input is positive, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port, as shown on this figure.



The signal has one of three values:

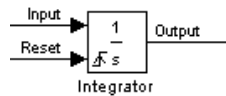
- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.



When this option is selected, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

## Resetting the State

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type, as shown in this figure.



Select **rising** to trigger the state reset when the reset signal has a rising edge. Select **falling** to trigger the state reset when the reset signal has a falling edge. Select **either** to trigger the reset when either a rising or falling signal occurs.

The reset port has direct feedthrough. If the block output is fed back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the block state into the reset port instead. To access the block's state, select the **Show state port** check box.

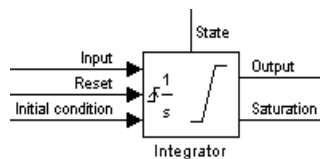
## Specifying the Absolute Tolerance for the Block State

When your model contains states having vastly different magnitudes, defining the absolute tolerance for the model might not provide sufficient error control. To define the absolute tolerance for an Integrator block's state, provide a value for the **Absolute tolerance** parameter. If the block has more than one state, the same value is applied to all states.

For more information about error control, see “Error Tolerances” on page 4-13.

## Choosing All Options

When all options are selected, the icon looks like this.

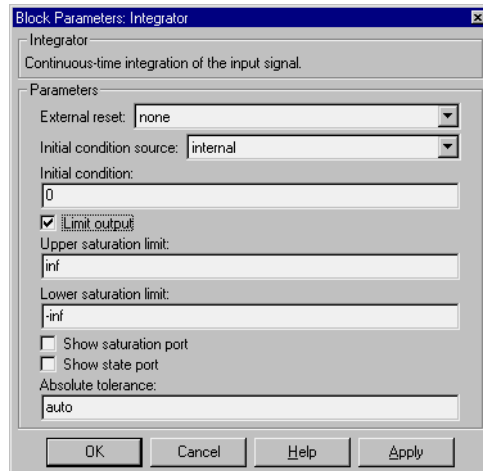


# Integrator

## Data Type Support

An Integrator block accepts and outputs signals of type `double` on its data ports. Its external reset port accepts signals of type `double` or `boolean`.

## Parameters and Dialog Box



### External reset

Resets the states to their initial conditions when a trigger event (**rising**, **falling**, or **either**) occurs in the reset signal.

### Initial condition source

Gets the states' initial conditions from the **Initial condition** parameter (if set to **internal**) or from an external block (if set to **external**).

### Initial condition

The states' initial conditions. Set the **Initial condition source** parameter value to **internal**.

### Limit output

If checked, limits the states to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

### Upper saturation limit

The upper limit for the integral. The default is `inf`.

### Lower saturation limit

The lower limit for the integral. The default is `-inf`.

**Show saturation port**

If checked, adds a saturation output port to the block.

**Show state port**

If checked, adds an output port to the block for the block's state.

**Absolute tolerance**

Absolute tolerance for the block's states.

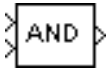
<b>Characteristics</b>	Direct Feedthrough	Yes, of the reset and external initial condition source ports
	Sample Time	Continuous
	Scalar Expansion	Of parameters
	States	Inherited from driving block or parameter
	Vectorized	Yes
	Zero Crossing	If the <b>Limit output</b> option is selected, one for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation

# Logical Operator

**Purpose** Perform the specified logical operation on the input.

**Library** Math

**Description** The Logical Operator block performs any of these logical operations on its inputs: AND, OR, NAND, NOR, XOR, and NOT. The output depends on the number of inputs, their vector size, and the selected operator. The output is 1 if TRUE and 0 if FALSE. The block icon shows the selected operator.

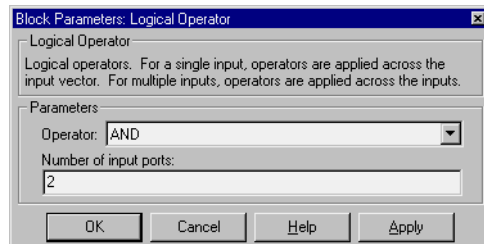


- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the operation is performed between corresponding elements of the vectors to produce a vector output.
- For a single vector input, the block applies the operation (except the NOT operator)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

When configured as a multi-input XOR gate, this block performs an addition modulo two operation as mandated by the IEEE standard for logic elements.

**Data Type Support** An Logical Operator block accepts signals of type `boolean` on its input ports, unless boolean compatibility mode is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45), in which case the block also accepts inputs of type `double`. A nonzero input of type `double` is treated as TRUE (1), a zero input as FALSE (0). All inputs must be of the same type. The output of the block is of the same type as the input.

## Parameters and Dialog Box



**Operator**

The logical operator to be applied to the block inputs. Valid choices are the operators listed above.

**Number of input ports**

The number of block inputs. The value must be appropriate for the selected operator.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of inputs
	Vectorized	Yes
	Zero Crossing	No

# Look-Up Table

**Purpose** Perform piecewise linear mapping of the input.

**Library** Functions & Tables

**Description** The Look-Up Table block maps an input to an output using linear interpolation of the values defined in the block's parameters.



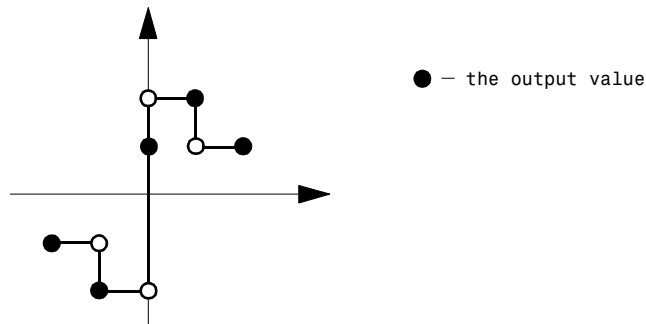
You define the table by specifying (either as row or column vectors) the **Vector of input values** and **Vector of output values** parameters. The block produces an output value by comparing the block input with values in the input vector:

- If it finds a value that matches the block's input, the output is the corresponding element in the output vector.
- If it does not find a value that matches, it performs linear interpolation between the two appropriate elements of the table to determine an output value. If the block input is less than the first or greater than the last input vector element, the block extrapolates using the first two or the last two points.

To map two inputs to an output, use the Look-Up Table (2-D) block. For more information, see Look-Up Table (2-D) on page 8-113.

To create a table with step transitions, repeat an input value with different output values. For example, these input and output parameter values create the input/output relationship described by the plot that follows:

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]



This example has three step discontinuities: at  $u = -1$ ,  $0$ , and  $+1$ .

When there are two points at a given input value, the block generates output according to these rules:

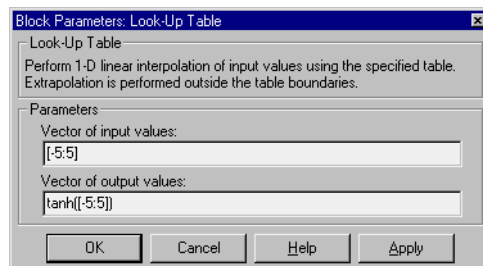
- When  $u$  is less than zero, the output is the value connected with the point first encountered when moving away from the origin in a negative direction. In this example, when  $u$  is -1,  $y$  is -2, marked with a solid circle.
- When  $u$  is greater than zero, the output is the value connected with the point first encountered when moving away from the origin in a positive direction. In this example, when  $u$  is 1,  $y$  is 2, marked with a solid circle.
- When  $u$  is at the origin and there are two output values specified for zero input, the actual output is their average. In this example, if there were no point at  $u = 0$  and  $y = 1$ , the output would be 0, the average of the two points at  $u = 0$ . If there are three points at zero, the block generates the output associated with the middle point. In this example, the output at the origin is 1.

The Look-Up Table block icon displays a graph of the input vector versus the output vector. When a parameter is changed on the block's dialog box, the graph is automatically redrawn when you press the **Apply** or **Close** button.

## Data Type Support

A Look-Up Table block accepts and outputs signals of type double.

## Parameters and Dialog Box



### Vector of input values

The vector of values containing possible block input values. This vector must be the same size as the output vector. The input vector must be monotonically increasing.

# Look-Up Table

---

## Vector of output values

The vector of values containing block output values. This vector must be the same size as the input vector.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No



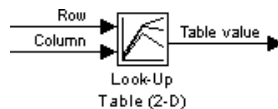
**Purpose** Perform piecewise linear mapping of two inputs.

**Library** Functions & Tables

**Description** The Look-Up Table (2-D) block maps the block inputs to an output using linear interpolation of a table of values defined by the block's parameters.



You define the possible output values as the **Table** parameter. You define the values that correspond to its rows and columns with the **Row** and **Column** parameters. The block generates an output value by comparing the block inputs with the **Row** and the **Column** parameters. The first input identifies a row, and the second input identifies a column, as shown by this figure.



The block generates output based on the input values:

- If the inputs match row and column parameter values, the output is the table value at the intersection of the row and column.
- If the inputs do not match row and column parameter values, the block generates output by linearly interpolating between the appropriate table values. If either or both block inputs are less than the first or greater than the last row or column parameter values, the block extrapolates from the first two or last two points.

If either the **Row** or **Column** parameter has a repeating value, the block chooses a value using the technique described for the Look-Up Table block.

The Look-Up Table block allows you to map a single input value into a vector of output values (see Look-Up Table on page 8-110).

## Example

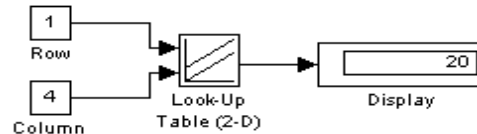
In this example, the block parameters are defined as:

```
Row:      [ 1  2 ]
Column:   [ 3  4 ]
Table:    [10 20; 30 40]
```

# Look-Up Table (2-D)

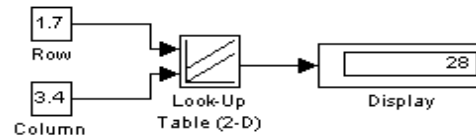
The first figure shows the block outputting a value at the intersection of block inputs that match row and column values. The first input is 1 and the second input is 4. These values select the table value at the intersection of the first row (row parameter value 1) and second column (column parameter value 4).

	3	4
1	10	20
2	30	40



In the second figure, the first input is 1.7 and the second is 3.4. These values cause the block to interpolate between row and column values, as shown in the table at the left. The value at the intersection (28) is the output value.

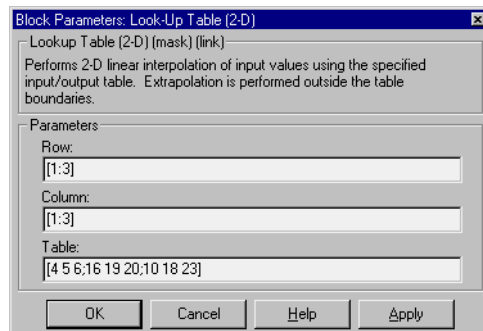
	3	3.4	4
1	10	14	20
1.7	24	28	34
2	30	34	40



## Data Type Support

A Look-Up Table (2-D) block accepts and outputs signals of type double.

## Parameters and Dialog Box



### Row

The row values for the table, entered as a vector. The vector values must increase monotonically.

**Column**

The column values for the table, entered as a vector. The vector values must increase monotonically.

**Table**

The table of output values. The matrix size must match the dimensions defined by the **Row** and **Column** parameters.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving blocks
	Scalar Expansion	Of one input if the other is a vector
	Vectorized	Yes
	Zero Crossing	No

# Magnitude-Angle to Complex

**Purpose** Convert a magnitude and/or a phase angle signal to a complex signal.

**Library** Math

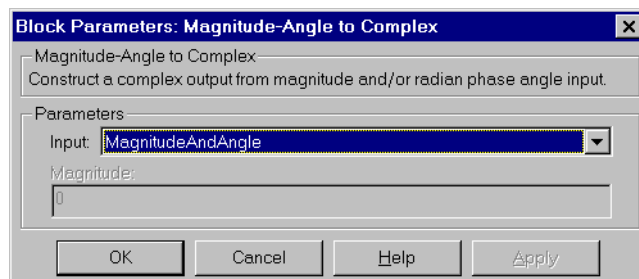
**Description** The Magnitude-Angle to Complex block converts magnitude and/or phase angle inputs to a complex-valued output signal. The inputs must be real-valued signals of type `double`. The angle input is assumed to be in radians. The data type of the complex output signal is `double`.



The inputs may be both vectors of equal size, or one input may be a vector and the other a scalar. If the block has a vector input, the output is a vector of complex signals. The elements of a magnitude input vector are mapped to magnitudes of the corresponding complex output elements. An angle input vector is similarly mapped to the angles of the complex output signals. If one input is a scalar, it is mapped to the corresponding component (magnitude or angle) of all the complex output signals.

**Data Type Support** See block description above.

## Parameters and Dialog Box



### Input

Specifies the kind of input: a magnitude input, an angle input, or both.

### Angle (Magnitude)

If the input is an angle signal, specifies the constant magnitude of the output signal. If the input is a magnitude, specifies the constant phase angle in radians of the output signal.

# Magnitude-Angle to Complex

---

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of the input when the function requires two inputs
	Vectorized	Yes
	Zero Crossing	No

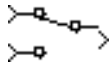
# Manual Switch

---

**Purpose** Switch between two inputs.

**Library** Nonlinear

## Description



The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click on the block icon (there is no dialog box). The selected input is propagated to the output, while the unselected input is discarded. You can set the switch before the simulation is started or throw it while the simulation is executing to interactively control the signal flow. The Manual Switch block retains its current state when the model is saved.

## Data Type Support

A Manual Switch block accepts all input types. Both inputs must be of the same numeric and data type. The block's output has the same numeric type (real or complex) and data type as its input.

## Parameters and Dialog Box

None

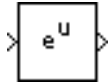
## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	No

**Purpose** Perform a mathematical function.

**Library** Math

**Description** The Math Function block performs numerous common mathematical functions.



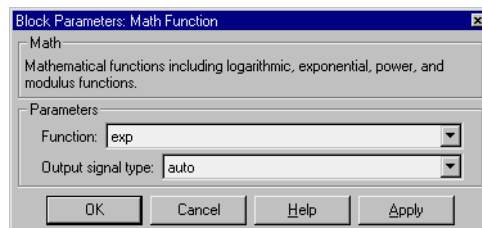
You can select one of these functions from the **Function** list: exp, log,  $10^u$ , log10, square, sqrt, pow, reciprocal, hypot, rem, and mod. The block output is the result of the function operating on the input or inputs.

The name of the function appears on the block icon. Simulink automatically draws the appropriate number of input ports.

Use the Math Function block instead of the Fcn block when you want vectorized output because the Fcn block can produce only scalar output.

**Data Type Support** A Math Function block accepts complex or real-valued signals or signal vectors of type double. The output signal type is real or complex, depending on the setting of the **Output signal type** parameter.

## Parameters and Dialog Box



## Function

The mathematical function.

# Math Function

## Output signal type

The dialog allows you to select the output signal type of the Math Function block as real, complex, or auto.

Function	Input	Output Signal Type		
	Signal	Auto	Real	Complex
Exp, log, log <sup>^</sup> , log10, square, sqrt, pow, reciprocal, conjugate	real complex	real complex	real error	complex complex
magnitude squared	real complex	real real	real real	complex complex
hypot, rem, mod	real complex	real error	real error	complex error

## Characteristics

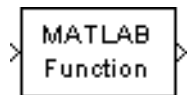
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs
Vectorized	Yes
Zero Crossing	No



**Purpose** Apply a MATLAB function or expression to the input.

**Library** Functions & Tables

**Description** The MATLAB Fcn block applies the specified MATLAB function or expression to the input. The specified function or expression is applied to the input. The output of the function must match the output width of the block or an error occurs.



Here are some sample valid expressions for this block:

```
sin
atan2(u(1), u(2))
u(1)^u(2)
```

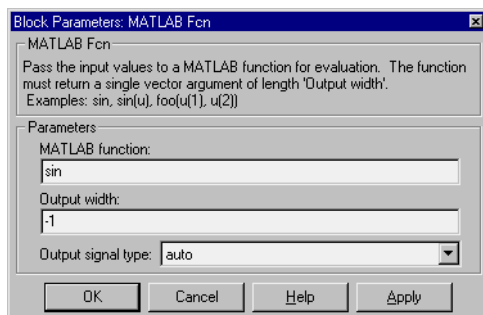
---

**Note** This block is slower than the Fcn block because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Fcn block or the Math Function block) instead, or writing the function as an M-file or MEX-file S-function, then accessing it using the S-Function block.

---

**Data Type Support** A MATLAB Fcn block accepts one complex- or real-valued input of type double and generates real or complex output of type double, depending on the setting of the **Output signal type** parameter.

## Parameters and Dialog Box



### MATLAB function

The function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

## Output width

The output width. If the output width is to be the same as the input width, specify -1. Otherwise, you must specify the correct width or an error will result.

## Output signal type

The dialog allows you to select the output signal type of the MATLAB Fcn as real, complex, or auto. A value of auto sets the block's output type to be the same as the type of the input signal. If the block has no input signal, auto sets the output type to the port type to which the output is connected.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Vectorized	Yes
Zero Crossing	No

**Purpose** Multiply the input by a matrix.

**Library** Math

**Description** The Matrix Gain block implements a matrix gain. It generates its output by multiplying its vector input by a specified matrix



$$y = Ku$$

where  $K$  is the gain and  $u$  is the input.

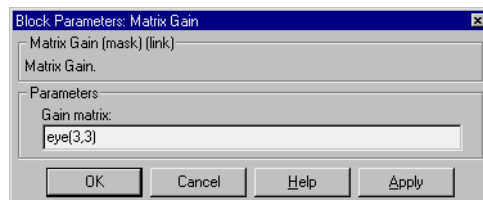
If the specified matrix has  $m$  rows and  $n$  columns, then the input to this block should be a vector of length  $n$ . The output is a vector of length  $m$ .

The block icon always displays  $K$ .

If the matrix contains zeros, Simulink converts the matrix gain to a sparse matrix for efficient multiplication.

**Data Type Support** A Matrix Gain block accepts and outputs real-valued signals of type double.

## Parameters and Dialog Box



### Gain matrix

The gain, specified as a matrix. The default is  $\text{eye}(3,3)$ .

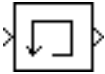
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Continuous
	Scalar Expansion	No
	States	0
	Vectorized	Yes
	Zero Crossing	No

# Memory

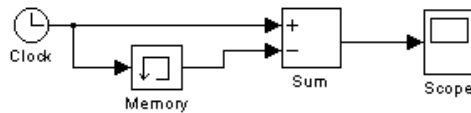
**Purpose** Output the block input from the previous integration step.

**Library** Continuous

**Description** The Memory block outputs its input from the previous time step, applying a one integration step sample-and-hold to its input signal.



This sample model (which, to provide more useful information, would be part of a larger model) demonstrates how to display the step size used in a simulation. The Sum block subtracts the time at the previous step, generated by the Memory block, from the current time, generated by the clock.



---

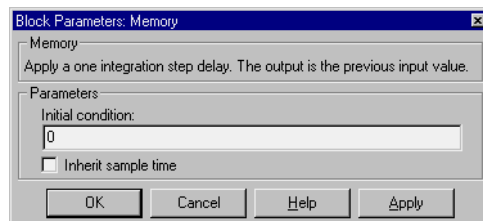
**Note** Avoid using the Memory block when integrating with ode15s or ode113, unless the input to the block does not change.

---

## Data Type Support

A Memory block accepts signals of any numeric type (complex or real) and data type, including user-defined types. If the input type is user-defined, the initial condition must be 0.

## Parameters and Dialog Box



### Initial condition

The output at the initial integration step.

### Inherit sample time

Check this box to cause the sample time to be inherited from the driving block.

<b>Characteristics</b>	Direct Feedthrough	No
	Sample Time	Continuous, but inherited if the <b>Inherit sample time</b> check box is selected
	Scalar Expansion	Of the <b>Initial condition</b> parameter
	Vectorized	Yes
	Zero Crossing	No

# Merge

**Purpose** Combine input lines into a scalar output line

**Library** Signals & Systems

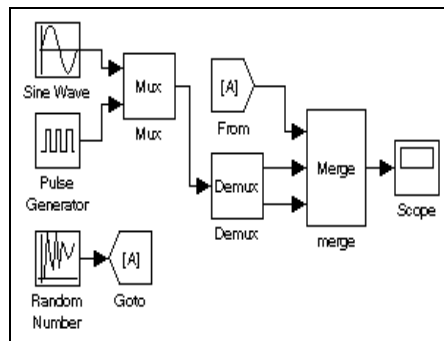
## Description



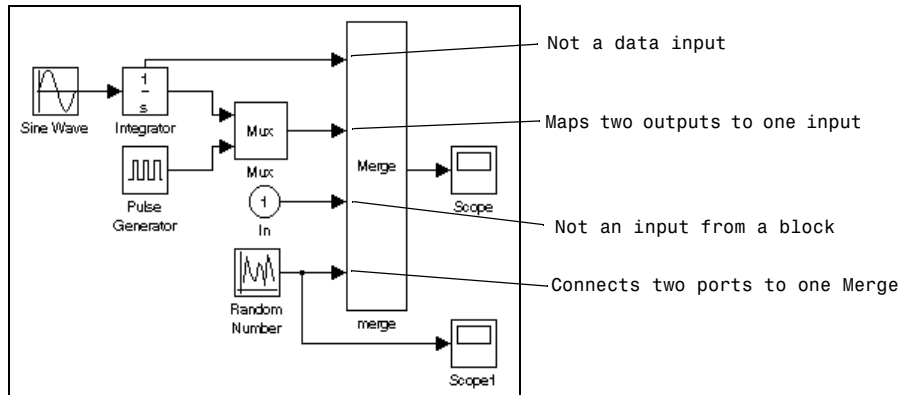
The Merge block combines its inputs into a single output line whose value at any time is equal to the most recently computed output of its driving blocks. You can specify any number of inputs by setting the block's **Number of Inputs** parameter. All inputs must be the same width, for example, all scalar or all vectors of width three. You can specify an initial output value by setting the blocks **Initial Output** parameter. If you do not specify an initial output and one or more of the driving blocks do, the Merge block's initial output equals the most recently evaluated initial output of the driving blocks.

Merge blocks facilitate creation of alternately executing subsystems. See "Creating Alternately Executing Subsystems" on page 7-12 for an application example.

Simulink restricts the kinds of connections you can make to the inputs of a Merge block. In particular, it permits only connections that establish a one-to-one mapping from the outputs of nonvirtual blocks to the inputs of a Merge block. For example, you can use a Go To/From block pair to connect the scalar or vector output of a nonvirtual block in one part of a diagram to the input of a Merge block in another part of the diagram. The following diagram illustrates valid ways to connect nonvirtual blocks to a Merge block.



You cannot use a Merge block to connect multiple nonvirtual outputs to a single input on a Merge block. The following diagram illustrates invalid ways to connect nonvirtual blocks to a Merge block.

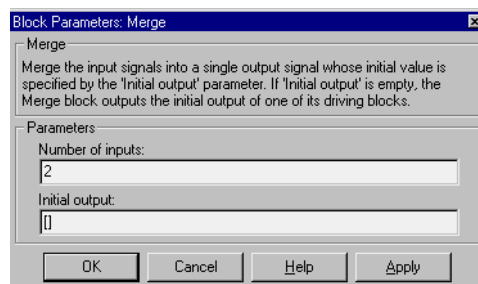


Simulink checks for invalid connections in a block diagram at the start of a simulation. If it detects an invalid connection, it stops and displays an error message.

## Data Type Support

A Merge block accepts signals of any numeric type (complex or real) and data type, including user-defined types. If the input type is user-defined, the initial condition must be 0.

## Parameters and Dialog Box



### Number of inputs

The number of input ports to merge. Ports may be scalar or vector.

# Merge

---

## Initial output

Initial value of output. If unspecified, the initial output equals the initial output, if any, of one of the driving blocks.

<b>Characteristics</b>	Sample Time	Inherited from the driving block
	Vectorized	Yes
	Scalar Expansion	No



**Purpose** Output the minimum or maximum input value.

**Library** Math

**Description** The MinMax block outputs either the minimum or the maximum element or elements of the input(s). You can choose which function to apply by selecting one of the choices from the **Function** parameter list.

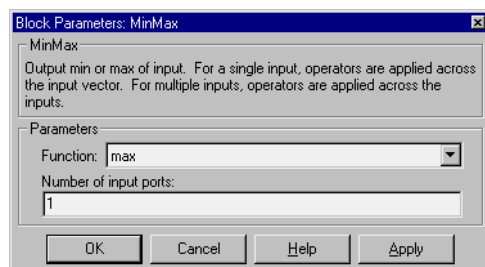


If the block has one input port, the block outputs a scalar that is the minimum or maximum element of the input vector.

If the block has more than one input port, the block performs an element-by-element comparison of the input vectors. Each element of the block output vector is the result of the comparison of the elements of the input vectors.

**Data Type Support** A MinMax block accepts and outputs real-valued signals of type double.

### Parameters and Dialog Box



**Function** The function (min or max) to apply to the input.

**Number of input ports** The number of inputs to the block.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from the driving block
	Scalar Expansion	Of the inputs

# MinMax

---

Vectorized

Yes

Zero Crossing

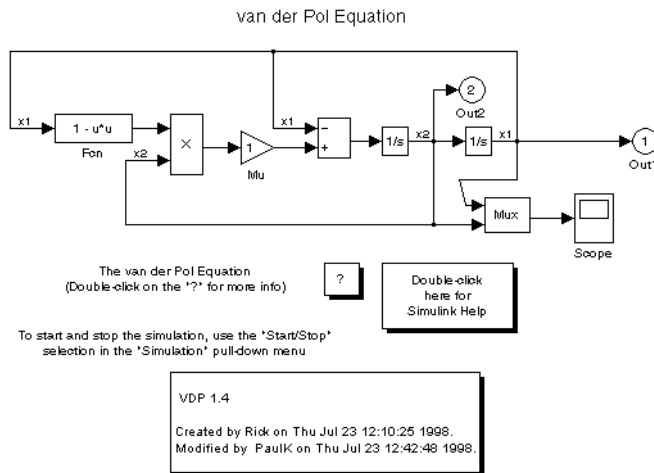
Yes, to detect minimum and maximum values

**Purpose** Display revision control information in a model.

**Library** Signals & Systems

**Description** The Model Info block displays revision control information about a model as an annotation block in the model's block diagram. The following diagram illustrates use of a Model Info block to display information about the vdp model.

Model Info Annotation

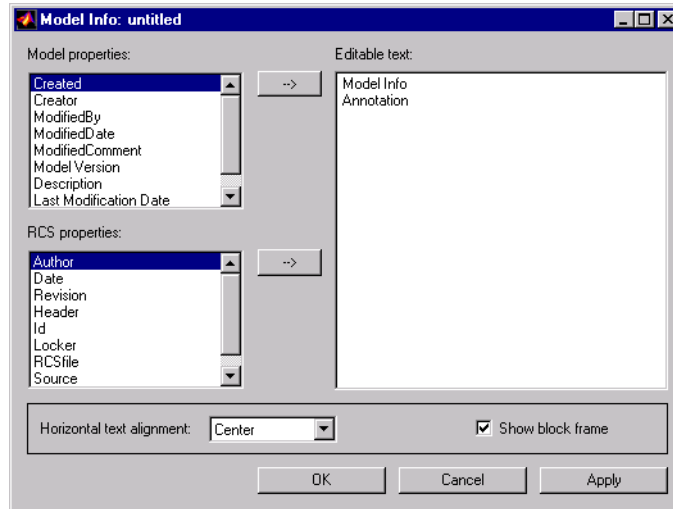


A Model Info block can show revision control information embedded in the model itself and/or information maintained by an external revision control or configuration management system. A Model Info block's dialog allows you to specify the content and format of the text displayed by the block.

**Data Type Support** Not applicable.

# Model Info

## Dialog Box



The Model Info block dialog box includes the following fields:

**Editable text.** Enter the text to be displayed by the Model Info block in this field. You can freely embed variables of the form %<propname>, where propname is the name of a model or revision control system property, in the entered text. The value of the property replaces the variable in the displayed text. For example, suppose that the current version of the model is 1.1. Then the entered text

```
Version %<ModelVersion>
```

appears as

```
Version 1.1
```

in the displayed text. The model and revision control system properties that you can reference in this way are listed in the **Model properties** and **Configuration manager properties** fields.

**Model properties.** Lists revision control properties stored in the model. Selecting a property and then selecting the adjacent arrow button enters the corresponding variable in the **Editable text** field. For example, selecting CreatedBy enters %<CreatedBy%> in the **Editable text** field. See “Version

Control Properties” on page 3–77 for a description of the usage of the properties specified in this field.

**RCS properties.** This field appears only if you previously specified an external configuration manager for this model (see “Configuration manager” on page 3–73). The title of the field changes to reflect the selected configuration manager (for example, **RCS Properties**). The field lists version control information maintained by the external system that you can include in the Model Info block. To include an item from the list, select it and then click the adjacent arrow button.

---

**Note** The selected item does not appear in the Model Info block until you check the model in or out of the repository maintained by the configuration manager and you have closed and reopened the model.

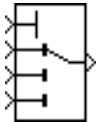
---

# Multiport Switch

**Purpose** Choose between block inputs.

**Library** Nonlinear

**Description** The Multiport Switch block chooses between a number of inputs.



The first (top) input is the control input and the other inputs are data inputs. The value of the control input determines which data input to pass through to the output port.

If the control input is not an integer value, the Multiport Switch truncates the value to the nearest integer and issues a warning. If the (truncated) control input is less than one or greater than the number of input ports, the switch issues an out-of-bounds error. Otherwise, the switch passes the data input that corresponds to the (truncated) control input. The following table summarizes the Multiport Switch's behavior.

<b>(Truncated) Control Input</b>	<b>Passes This Data Input</b>
Less than 1	Out of bounds error
1	First input
2	Second input
etc.	etc.
Greater than the number of data inputs	Out of bounds error

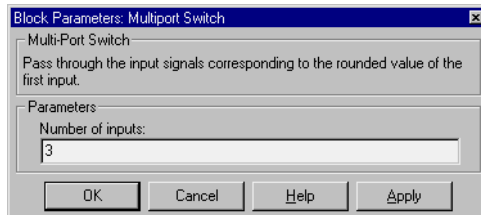
Data inputs can be scalar or vector. The control input can be a scalar or a vector. The block output is determined by these rules:

- If inputs are scalar, the output is a scalar.
- If the block has more than one data input, at least one of which is a vector, the output is a vector. Any scalar inputs are expanded to vectors.
- If the block has only one data input and that input is a vector, the block output is the element of the vector that corresponds to the truncated value of the control input.

## Data Type Support

The control input of a Multiport Switch block accepts a real-valued signal of any built-in data type except boolean. The data inputs accept real- or complex-valued inputs of any type. All data inputs must be of the same data and numeric type. The signal type of the block's output is the same as that of its data inputs.

## Parameters and Dialog Box



### Number of inputs

The number of data inputs to the block.

## Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block(s)
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	No

# Mux

---

**Purpose** Combine several input lines into a vector line.

**Library** Signals & Systems

**Description** The Mux block combines several input lines into one vector line. Each input line can carry a scalar or vector signal. The output of a Mux block is a vector.



You can assign names to the input signals by:

- Labeling the lines that represent the incoming signals
- Entering a comma-separated list of signal names as the value of the Mux block's **Number of inputs** parameter; for example, if you enter `position,velocity` in this parameter, the Mux block will have two inputs, named `position` and `velocity`.

The default name for an unlabeled line or unconnected port is `signalN`, where `N` is the input port number. This option is useful when you are defining a signal bus from which individual signals can be extracted by using the Bus Selector block.

If you define the **Number of inputs** parameter as a scalar, Simulink determines the input widths by checking the output ports of the blocks feeding the Mux block. If any input is a vector, all of its elements are combined by the block.

If it is necessary to define input widths explicitly, you can specify them as a vector. Include elements with `-1` values for those inputs whose widths are to be determined dynamically (during the simulation). If an input signal width does not match the expected width, Simulink displays an error message.

For example, `[4 1 2]` indicates three inputs forming a seven-element output vector: the first four output elements are from the first input, the fifth element comes from the second input, and the sixth and seventh elements come from the third input. If it is not important that these inputs have fixed widths, you could specify the **Number of inputs** as 3.

To specify three inputs where the first input vector must have four elements, you could specify `[4 -1 -1]`. Simulink determines the widths of the second and third inputs and sizes the output width accordingly.



Simulink draws the Mux block with the specified number of inputs. If you change the number of input ports, Simulink adds or removes them from the bottom of the block icon.

### Using a Variable to Provide the Number of Inputs Parameter

When you specify the **Number of inputs** parameter as a variable, Simulink issues an error message if the variable is undefined in the workspace.

---

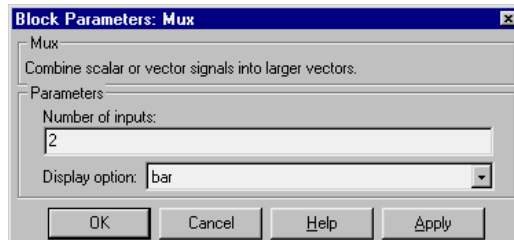
**Note** Simulink hides the name of a Mux block when you copy it from the Simulink block library to a model.

---

## Data Type Support

A Mux block accepts real or complex signals of any data type, including mixed-type vectors.

## Parameters and Dialog Box



### Number of inputs

The number and width of inputs. The width of the output line equals the sum of the widths of the input lines. You can enter a comma-separated list of signal names for this parameter field when the **Display option** parameter is names.

## Display option

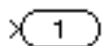
The appearance of the block icon in your model.

<b>Display Option</b>	<b>Appearance of Block in Model</b>
none	Mux appears inside block icon
names	Displays signal names next to each port
bar	Displays the block icon in a solid foreground color

**Purpose** Create an output port for a subsystem or an external output.

**Library** Signals & Systems

**Description** Outports are the links from a system to a destination outside the system.



Simulink assigns Outport block port numbers according to these rules:

- It automatically numbers the Outport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Outport block, it is assigned the next available number.
- If you delete an Outport block, other port numbers are automatically renumbered to ensure that the Outport blocks are in sequence and that no numbers are omitted.
- If you copy an Outport block into a system, its port number is *not* renumbered unless its current number conflicts with an Outport block already in the system. If the copied Outport block port number is not in sequence, you must renumber the block or you will get an error message when you run the simulation or update the block diagram.

### Output Blocks in a Subsystem

Output blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the Outport block whose **Port number** parameter is 1 sends its signal to the block connected to the top-most output port on the Subsystem block.

If you renumber the **Port number** of an Outport block, the block becomes connected to a different output port, although the block continues to send the signal to the same block outside the subsystem.

When you create a subsystem by selecting existing blocks, if more than one Outport block is included in the grouped blocks, Simulink automatically renumbers the ports on the blocks.

The Outputport block name appears in the Subsystem block icon as a port label. To suppress display of the label, select the Outputport block and choose **Hide Name** from the **Format** menu.

## Output Blocks in a Conditionally Executed Subsystem

When an Outputport block is in a triggered and/or enabled subsystem, you can specify what happens to its output when the subsystem is disabled: it can be **reset** to an initial value or **held** at its most recent value. The **Output when disabled** popup menu provides these options. The **Initial output** parameter is the value of the output before the subsystem executes and, if the **reset** option is chosen, while the subsystem is disabled.

## Output Blocks in a Top-Level System

Outputport blocks in a top-level system have two uses: to supply external outputs to the workspace, which you can do by using either the **Simulation Parameters** dialog box or the `sim` command, and to provide a means for analysis functions to obtain output from the system.

- To supply external outputs to the workspace, using the **Simulation Parameters** dialog box (see “Saving Output to the Workspace” on page 4-20) or the `sim` command (see `sim` on page 4-30). For example, if a system has more than one Outputport block, the following command

```
[t,x,y] = sim(...);
```

writes `y` as a matrix, with each column containing data for a different Outputport block. The column order matches the order of the port numbers for the Outputport blocks.

If you specify more than one variable name after the second (state) argument, data from each Outputport block is written to a different variable. For example, if the system has two Outputport blocks, to save data from Outputport block 1 to `speed` and the data from Outputport block 2 to `dist`, you could specify this command:

```
[t,x,speed,dist] = sim(...);
```

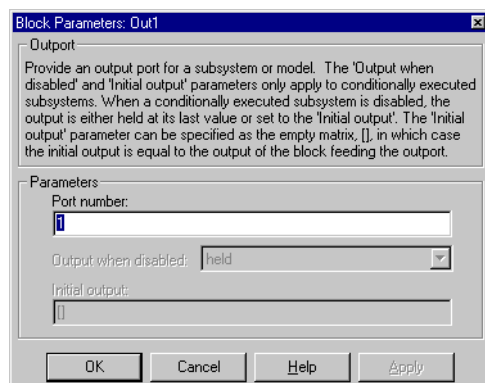
- To provide a means for the `linmod` and `trim` analysis functions to obtain output from the system. For more information about using Outputport blocks with analysis commands, see Chapter 5.

## Numeric and Data Type Support

An Outputport block accepts complex or real signals of any MATLAB data type as input. The numeric and data type of the block's output is the same as that of its input. The elements of a signal vector connected to an Outputport block can be of differing numeric and data types except in the following circumstance. If the output is in a conditionally executed subsystem and the initial output is not specified, all elements of an input vector must be of the same numeric and data type.

Simulink's data type conversion rules apply to an outputport's **Initial output** parameter. If the initial value is in the range of the block's output data type, Simulink converts the initial value to the output data type. If the conversion entails a loss of precision, Simulink issues a warning message. If the specified initial output is out of range of the output data type, Simulink halts the simulation and signals an error. Note that the block's output data type is the data type of the signal connected to its input.

## Parameters and Dialog Box



### Port number

The port number of the Outputport block.

### Output when disabled

For conditionally executed subsystems, what happens to the block output when the system is disabled.

### Initial output

For conditionally executed subsystems, the block output before the subsystem executes and while it is disabled.

# Output

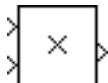
---

<b>Characteristics</b>	Sample Time	Inherited from driving block
	Vectorized	Yes

**Purpose** Generate the product or quotient of block inputs.

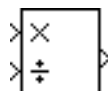
**Library** Math

**Description** The Product block multiplies or divides block inputs, depending on the value of the **Number of inputs** parameter:



- If the value is a combination of \* and / symbols, the number of block inputs is equal to the number of symbols. The block icon shows the appropriate symbol adjacent to each input port. The block output is the product of all inputs marked \* divided by all inputs marked /.

For example, this block icon is the result of entering \*/ as the parameter value.



- If the value is a scalar greater than 1, the block multiplies all inputs. If any input is a vector, the block output is an element-by-element product across the inputs. If all inputs are scalars, the output is a scalar. For a block having  $n$  inputs, if any input is a vector, each element of the output is generated as

$$y_i = u1_i \times u2_i \times \dots \times un_i$$

- If the value is 1, the block output is the scalar product of the elements of the input vector.

$$y = \prod u_i$$

This model represents using the Product block in this way. The solid line input signal indicates that the input is a vector.



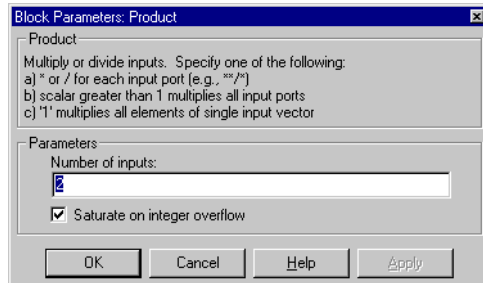
If necessary, Simulink resizes the block to show all input ports. If the number of inputs is changed, ports are added or deleted from the bottom of the block.

## Data Type Support

The Product block accepts real- or complex-valued signals of any data type. All input signals must be of the same data type. The output signal data type is the same as the input's.

# Product

## Parameters and Dialog Box



### Number of inputs

Either the number of inputs to the block or a combination of \* and / symbols. The default is 2.

### Saturate on integer overflow

If selected, this option causes the output of the Product block to saturate on integer overflow. In particular, if the output data type is an integer type, the block output is the maximum value representable by the output type or the computed output, whichever is smaller in the absolute sense. If the option is not selected, Simulink takes the action specified by the **Data overflow** event option on the **Diagnostics** page of the **Simulation Parameters** dialog (see “The Diagnostics Page” on page 4–24).

### Characteristics

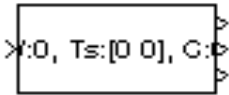
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Vectorized	Yes
Zero Crossing	No



**Purpose** Probe a line for its width, sample time, and/or complex signal flag.

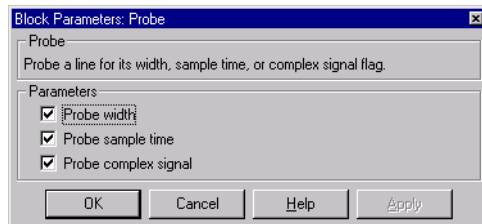
**Library** Signals & Systems

**Description** The Probe block outputs selected information about the signal on its input. The block can output the input signal's width, sample time, and/or a flag indicating whether the input is a complex-valued signal. The block has one input port. The number of output ports depends on the information that you select for probing, that is, signal width, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port. The block accepts real or complex-valued signals or vectors of any built-in data type. It outputs signals of type double. During simulation, the block's icon displays the probed data.



**Data Type Support** A Probe block accepts and outputs signals of type double.

## Parameters and Dialog Box



### Probe width

If checked, output width of probed line.

### Probe sample time

If checked, output sample time of probed line.

### Probe complex signal

If checked, output 1 if probed signal is complex; otherwise, 0.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Yes

# Pulse Generator

---

Vectorized	Yes
Zero Crossing	No

**Purpose** Generate pulses at regular intervals.

**Library** Sources

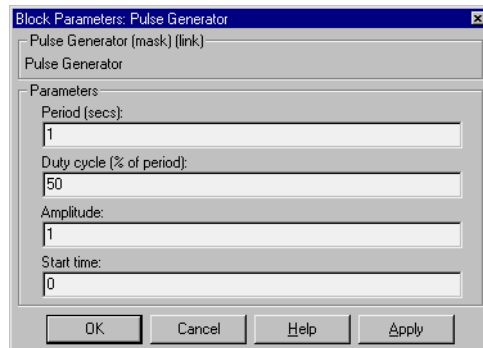
**Description** The Pulse Generator block generates a series of pulses at regular intervals.



Use the Pulse Generator block for continuous systems. To generate discrete signals, use the Discrete Pulse Generator block (see Discrete Pulse Generator on page 8-54).

**Data Type Support** A Pulse Generator block outputs signals of type double.

## Parameters and Dialog Box



### Period

The pulse period in seconds. The default is 1 second.

### Duty cycle

The duty cycle: the percentage of the pulse period that the signal is on. The default is 50 percent.

### Amplitude

The pulse amplitude. The default is 1.

**Start time**

The delay before the pulse is generated, in seconds. The default is 0 seconds.

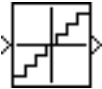
<b>Characteristics</b>	Sample Time	Inherited
	Scalar Expansion	Of parameters
	Vectorized	Yes
	Zero Crossing	No

# Quantizer

**Purpose** Discretize input at a specified interval.

**Library** Nonlinear

**Description** The Quantizer block passes its input signal through a stair-step function so that many neighboring points on the input axis are mapped to one point on the output axis. The effect is to quantize a smooth signal into a stair-step output. The output is computed using the round-to-nearest method, which produces an output that is symmetric about zero

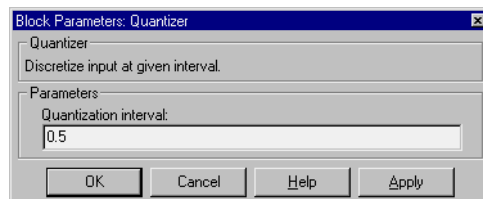


$$y = q * \text{round}(u/q)$$

where  $y$  is the output,  $u$  the input, and  $q$  the **Quantization interval** parameter.

**Data Type Support** A Quantizer block accepts and outputs signals of type double.

## Parameters and Dialog Box



### Quantization interval

The interval around which the output is quantized. Permissible output values for the Quantizer block are  $n*q$ , where  $n$  is an integer and  $q$  the **Quantization interval**. The default is 0.5.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of parameter
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Generate constantly increasing or decreasing signal.

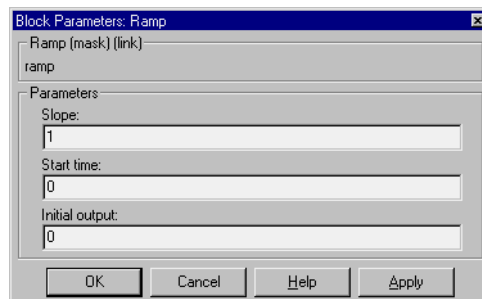
**Library** Sources

**Description** The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate.



**Data Type Support** A Ramp block outputs signals of type double.

## Parameters and Dialog Box



### Slope

The rate of change of the generated signal. The default is 1.

### Start time

The time at which the signal begins to be generated. The default is 0.

### Initial output

The initial value of the signal. The default is 0.

<b>Characteristics</b>	Sample Time	Inherited from driven block
	Scalar Expansion	Yes
	Vectorized	Yes
	Zero Crossing	Yes

# Random Number

---

**Purpose** Generate normally distributed random numbers.

**Library** Sources

**Description** The Random Number block generates normally distributed random numbers. The seed is reset to the specified value each time a simulation starts.



By default, the sequence produced has a mean of 0 and a variance of 1, although you can vary these parameters. The sequence of numbers is repeatable and can be produced by any Random Number block with the same seed and parameters. To generate a vector of random numbers with the same mean and variance, specify the **Initial seed** parameter as a vector.

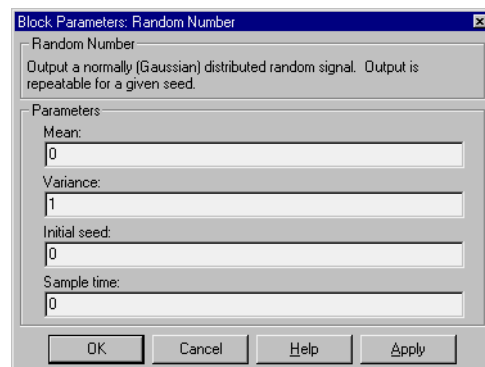
To generate uniformly distributed random numbers, use the Uniform Random Number block (see Uniform Random Number on page 8-212).

Avoid integrating a random signal because solvers are meant to integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

## Data Type Support

A Random Number block accepts and outputs signals of type double.

## Parameters and Dialog Box



### Mean

The mean of the random numbers. The default is 0.

### Variance

The variance of the random numbers. The default is 1.

**Initial seed**

The starting seed for the random number generator. The default is 0.

**Sample time**

The time interval between samples. The default is 0, causing the block to have continuous sample time.

<b>Characteristics</b>	Sample Time	Continuous or discrete
	Scalar Expansion	Of parameters
	Vectorized	Yes
	Zero Crossing	No

# Rate Limiter

**Purpose** Limit the rate of change of a signal.

**Library** Nonlinear

**Description** The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:



$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

$u(i)$  and  $t(i)$  are the current block input and time, and  $y(i-1)$  and  $t(i-1)$  are the output and time at the previous step. The output is determined by comparing  $rate$  to the **Rising slew rate** and **Falling slew rate** parameters:

- If  $rate$  is greater than the **Rising slew rate** parameter ( $R$ ), the output is calculated as:

$$y(i) = \Delta t \cdot R + y(i-1)$$

- If  $rate$  is less than the **Falling slew rate** parameter ( $F$ ), the output is calculated as:

$$y(i) = \Delta t \cdot F + y(i-1)$$

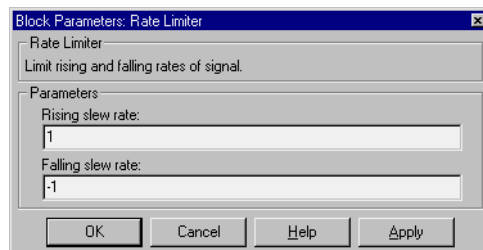
- If  $rate$  is between the bounds of  $R$  and  $F$ , the change in output is equal to the change in input:

$$y(i) = u(i)$$

## Data Type Support

A Rate Limiter block accepts and outputs signals of type double.

## Parameters and Dialog Box





**Rising slew rate**

The limit of the derivative of an increasing input signal.

**Falling slew rate**

The limit of the derivative of a decreasing input signal.

**Characteristics**

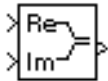
Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of input and parameters
Vectorized	Yes
Zero Crossing	No

# Real-Imag to Complex

**Purpose** Convert a magnitude and/or a phase angle signal to a complex signal.

**Library** Math

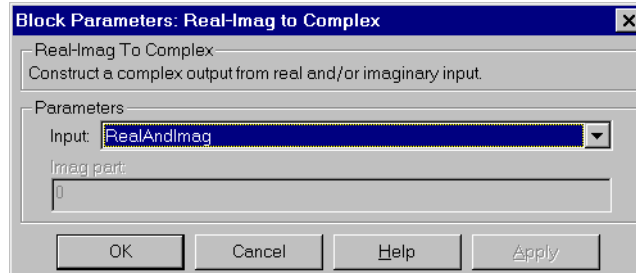
**Description** The Real-Imag to Complex block converts real and/or imaginary inputs to a complex-valued output signal. The inputs must be real-valued signals of type double. The data type of the complex output signal is double.



The inputs may be both vectors of equal size, or one input may be a vector and the other a scalar. If the block has a vector input, the output is a vector of complex signals. The elements of a real input vector are mapped to real parts of the corresponding complex output elements. An imaginary input vector is similarly mapped to the imaginary parts of the complex output signals. If one input is a scalar, it is mapped to the corresponding component (real or imaginary) of all the complex output signals.

**Data Type Support** See description above.

## Parameters and Dialog Box



**Input** Specifies the kind of input: a real input, an imaginary input, or both.

**Real (Imag) part** If the input is a real-part signal, this parameter specifies the constant imaginary part of the output signal. If the input is the imaginary part, this parameter specifies the constant real part of the output signal. Note that the title of this field changes to reflect its usage.

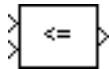
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of the input when the function requires two inputs
	Vectorized	Yes
	Zero Crossing	No

# Relational Operator

**Purpose** Perform the specified relational operation on the input.

**Library** Math

**Description** The Relational Operator block performs a relational operation on its two inputs and produces output according to the following table.



Operator	Output
==	TRUE if the first input is equal to the second input
~=	TRUE if the first input is not equal to the second input
<	TRUE if the first input is less than the second input
<=	TRUE if the first input is less than or equal to the second input
>=	TRUE if the first input is greater than or equal to the second input
>	TRUE if the first input is greater than the second input

If the result is TRUE, the output is 1; if FALSE, it is 0. You can specify inputs as scalars, vectors, or a combination of a scalar and a vector:

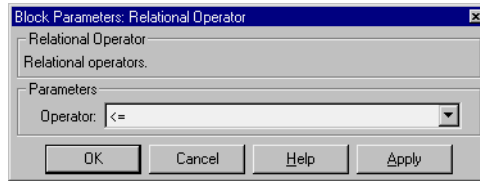
- For scalar inputs, the output is a scalar.
- For vector inputs, the output is a vector, where each element is the result of an element-by-element comparison of the input vectors.
- For mixed scalar/vector inputs, the output is a vector, where each element is the result of a comparison between the scalar and the corresponding vector element.

The block icon displays the selected operator.

## Data Type Support

A Relational Operator block accepts real signals of any data type. It outputs a signal of type `boolean`, unless `boolean compatibility mode` is enabled (see “Enabling Strict Boolean Type Checking” on page 3-45), in which case the block outputs a signal of type `double`.

## Parameters and Dialog Box



### Operator

The relational operator to be applied to the block inputs.

### Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of inputs
Vectorized	Yes
Zero Crossing	Yes, to detect when the output changes

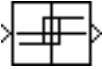
# Relay

---

**Purpose** Switch output between two constants.

**Library** Nonlinear

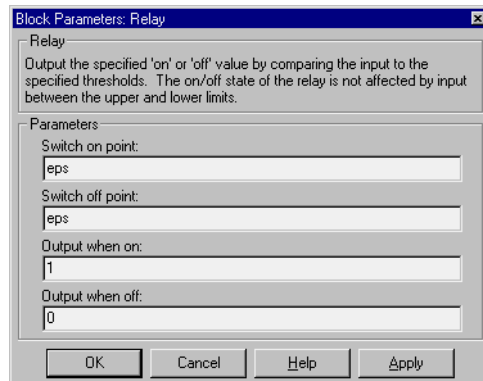
**Description** The Relay block allows the output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.



Specifying a **Switch on point** value greater than the **Switch off point** value models hysteresis, whereas specifying equal values models a switch with a threshold at that value. The **Switch on point** value must be greater than or equal to the **Switch off point**.

**Data Type Support** A Relay block accepts and outputs real signals of type double.

## Parameters and Dialog Box



### Switch on point

The on threshold for the relay. The default is eps.

### Switch off point

The off threshold for the relay. The default is eps.

### Output when on

The output when the relay is on. The default is 1.

## **Output when off**

The output when the relay is off. The default is 0.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Yes
	Vectorized	Yes
	Zero Crossing	Yes, to detect switch on and switch off points

# Repeating Sequence

---

**Purpose** Generate a repeatable arbitrary signal.

**Library** Sources

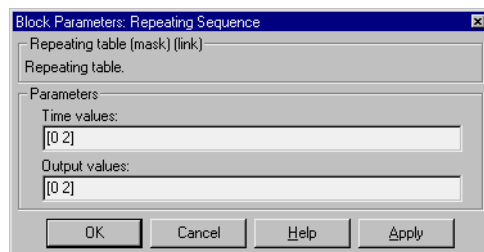
**Description** The Repeating Sequence block allows you to specify an arbitrary signal to be repeated regularly over time. When the simulation reaches the maximum time value in the **Time values** vector, the signal is repeated.



This block is implemented using the one-dimensional Look-Up Table block, performing linear interpolation between points.

**Data Type Support** A Repeating Sequence block outputs real signals of type double.

## Parameters and Dialog Box



### Time values

A vector of monotonically increasing time values. The default is [0 2].

### Output values

A vector of output values. Each corresponds to the time value in the same column. The default is [0 2].

<b>Characteristics</b>	Sample Time	Continuous
	Scalar Expansion	No
	Vectorized	No
	Zero Crossing	No



**Purpose** Perform a rounding function.

**Library** Math

**Description** The Rounding Function block performs common mathematical rounding functions.



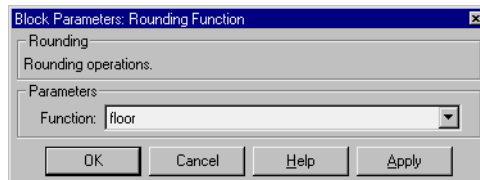
You can select one of these functions from the **Function** list: `floor`, `ceil`, `round`, and `fix`. The block output is the result of the function operating on the input or inputs. The Rounding Function block accepts and outputs real- or complex-valued signals of type `double`.

The name of the function appears on the block icon.

Use the Rounding Function block instead of the `Fcn` block when you want vectorized output because the `Fcn` block can produce only scalar output.

**Data Type Support** A Rounding Function block accepts and outputs real signals of type `double`.

## Parameters and Dialog Box



## Function

The rounding function.

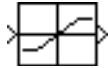
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	N/A
	Vectorized	Yes
	Zero Crossing	No

# Saturation

**Purpose** Limit the range of a signal.

**Library** Nonlinear

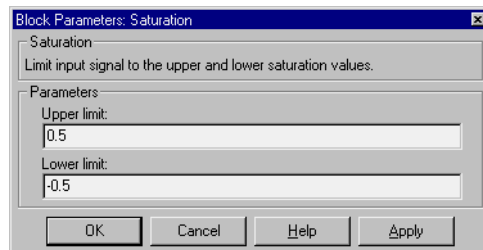
**Description** The Saturation block imposes upper and lower bounds on a signal. When the input signal is within the range specified by the **Lower limit** and **Upper limit** parameters, the input signal passes through unchanged. When the input signal is outside these bounds, the signal is clipped to the upper or lower bound.



When the parameters are set to the same value, the block outputs that value.

**Data Type Support** A Saturation block accepts and outputs real signals of type double.

## Parameters and Dialog Box



### Upper limit

The upper bound on the input signal. While the signal is above this value, the block output is set to this value.

### Lower limit

The lower bound on the input signal. While the signal is below this value, the block output is set to this value.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of parameters and input
	Vectorized	Yes
	Zero Crossing	Yes, to detect when the signal reaches a limit, and when it leaves the limit

**Purpose** Display signals generated during a simulation.

**Library** Sinks

**Description**

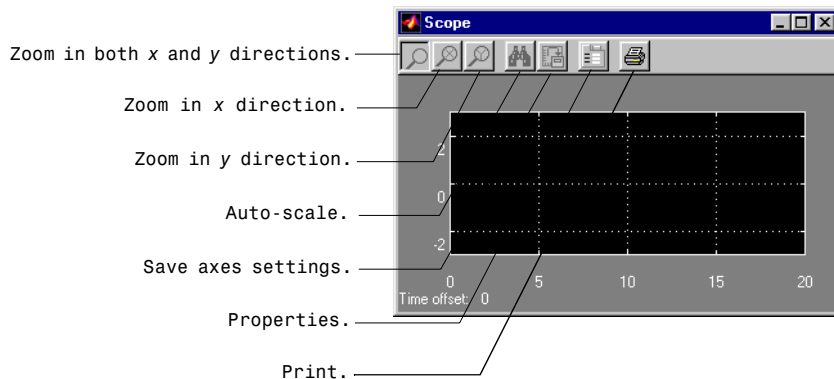


The Scope block displays its input with respect to simulation time. The Scope block can have multiple axes (one per port); all axes have a common time range with independent *y*-axes. The Scope allows you to adjust the amount of time and the range of input values displayed. You can move and resize the Scope window and you can modify the Scope's parameter values during the simulation.

When you start a simulation, Simulink does not open Scope windows, although it does write data to connected Scopes. As a result, if you open a Scope after a simulation, the Scope's input signal or signals will be displayed.

If the signal is continuous, the Scope produces a point-to-point plot. If the signal is discrete, the Scope produces a staircase plot.

The Scope provides toolbar buttons that enable you to zoom in on displayed data, display all the data input to the Scope, preserve axes settings from one simulation to the next, limit data displayed, and save data to the workspace. The toolbar buttons are labeled in this figure, which shows the Scope window as it appears when you open a Scope block.

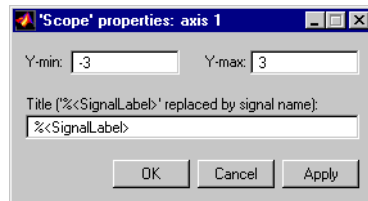


## Displaying Vector Signals

When displaying a vector signal, the Scope uses different colors in this order: yellow, magenta, cyan, red, green, and dark blue. When more than six signals are displayed, the Scope cycles through the colors in the order listed above.

## Y-Axis Limits

You set y-limits by right clicking on an axes and choosing **Properties...** The following dialog box appears.



### Y-min

Enter the minimum value for the y-axis.

### Y-max

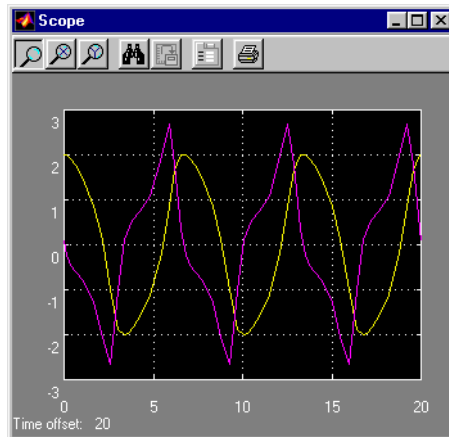
Enter the maximum value for the y-axis.

### Title

Enter the title of the plot. You can include a signal label in the title by typing %<SignalLabel> as part of the title string (%<SignalLabel> is replaced by the signal label).

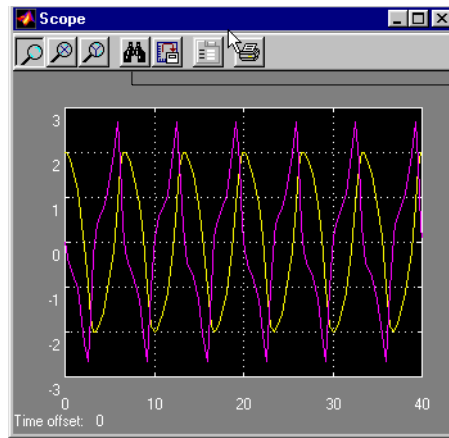
## Time Offset

This figure shows the Scope block displaying the output of the vdp model. The simulation was run for 40 seconds. Note that this scope shows the final 20 seconds of the simulation. The **Time offset** field displays the time corresponding to 0 on the horizontal axis. Thus, you have to add the offset to the fixed time range values on the  $x$ -axis to get the actual time.



## Auto-Scaling the Scope Axes

This figure shows the same output after pressing the **Auto-scale** toolbar button, which automatically scales both axes to display all stored simulation data. In this case, the  $y$ -axis was not scaled because it was already set to the appropriate limits.



The Auto-scale button

If you click on the **Auto-scale** button while the simulation is running, the axes are auto-scaled based on the data displayed on the current screen, and the auto-scale limits are saved as the defaults. This enables you to use the same limits for another simulation.

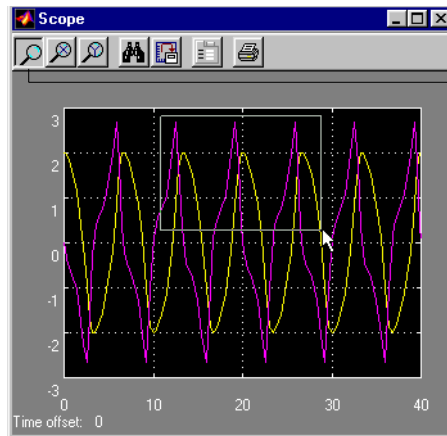
## Zooming

You can zoom in on data in both the  $x$  and  $y$  directions at the same time, or in either direction separately. The zoom feature is not active while the simulation is running.

To zoom in on data in both directions at the same time, make sure the left-most **Zoom** toolbar button is selected. Then, define the zoom region using a bounding box. When you release the mouse button, the Scope displays the data in that area. You can also click on a point in the area you want to zoom in on.

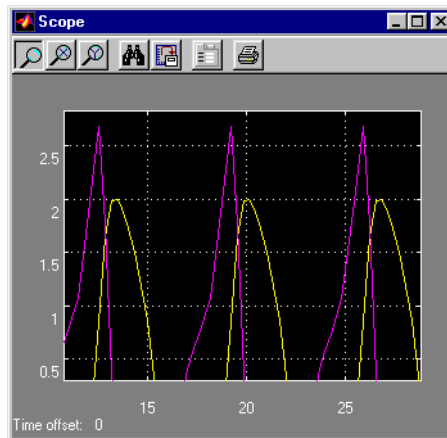
If the scope has multiple  $y$ -axes, and you zoom in on one set of  $x$ - $y$  axes, the  $x$ -limits on all sets of  $x$ - $y$  axes are changed so that they match, since all  $x$ - $y$  axes must share the same time base ( $x$ -axis).

This figure shows a region of the displayed data enclosed within a bounding box.



Zoom in both directions

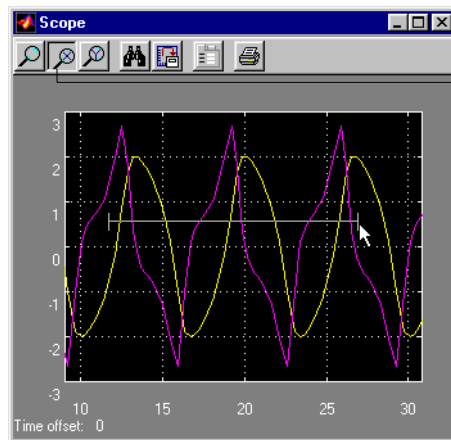
This figure shows the zoomed region, which appears after you release the mouse button.



To zoom in on data in just the x direction, click on the middle **Zoom** toolbar button. Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the mouse button, then moving the pointer

# Scope

to the other end of the region. This figure shows the Scope after defining the zoom region but before releasing the mouse button.



Zoom in x direction

When you release the mouse button, the Scope displays the magnified region. You can also click on a point in the area you want to zoom in on.

Zooming in the  $y$  direction works the same way except that you press the right-most **Zoom** toolbar button before defining the zoom region. Again, you can also click on a point in the area you want to zoom in on.

## Saving the Axes Settings

The **Save axes settings** toolbar button enables you to store the current  $x$ - and  $y$ -axis settings so you can apply them to the next simulation.



the Save axes settings button

You might want to do this after zooming in on a region of the displayed data so you can see the same region in another simulation. The time range is inferred from the current  $x$ -axis limits.

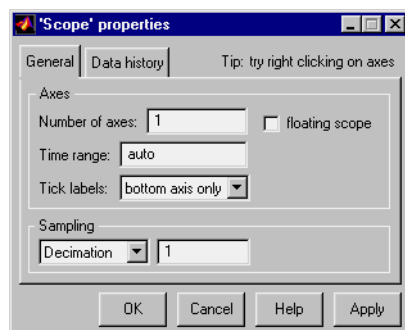


## Scope Properties

You can change axes limits, set the number of axes, time range, tick labels, sampling parameters, and saving options by choosing the **Properties** toolbar button.



When you click on the **Properties** button, this dialog box appears.



The dialog box has two tabs: **General** and **Data history**.

### General Parameters

You can set the axes parameters, time range, and tick labels in the **General** tab. You can also choose the **floating scope** option with this tab.

#### Number of axes

Set the number of  $y$ -axes in this data field. With the exception of the floating scope, there is no limit to the number of axes the Scope block can contain. All axes share the same time base ( $x$ -axis), but have independent  $y$ -axes. Note that the number of axes is equal to the number of input ports.

#### 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. Do not enter variable names in these fields.

## Tick labels

You can choose to have tick labels on all axes, on one axis, or on the bottom axis only in the **Tick labels** drop box.

## Floating scope

You can check the **Floating scope** check box if you want to have a floating scope. A floating Scope is a Scope block that can display the signals carried on one or more lines.

To add a floating Scope to a model, copy a Scope block into the model window, then open the block. Select the **Properties** button on the block's toolbar. Then, select the **General** tab and select the **Floating scope** check box.

To use a floating Scope during a simulation, first open the block. To display the signals carried on a line, select the line. Hold down the **Shift** key while clicking on another line to select multiple lines. It may be necessary to press the **Auto-scale data** button on the Scope's toolbar to find the signal and adjust the axes to the signal values. Note that floating scopes cannot have multiple axes.

A model can contain more than one floating Scope, although generally, it is not useful to have more than one floating Scope in a window because they will display the same signals.

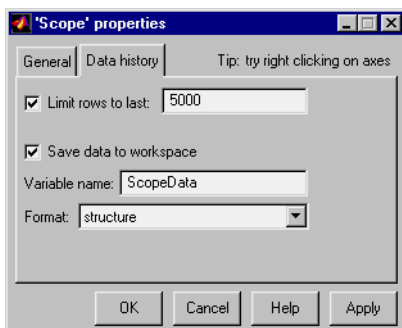
If you plan to use a floating scope during a simulation, you should disable buffer reuse. See “Disable optimized I/O storage” on page 4-25 for more information.

## Sampling

To specify a decimation factor, enter a number in the data field to the right of the **Decimation** choice. To display data at a sampling interval, select the **Sample time** choice and enter a number in the data field.

## Controlling Data Collection and Display

You can control the amount of data that the Scope stores and displays by setting fields on the **Data History** tab.



You can also choose to save data to the workspace in this tab. You apply the current parameters and options by clicking on the **Apply** or **OK** button. The values that appear in these fields are the values that will be used in the next simulation.

### Limit rows to last

You can limit the number of rows by checking the **Limit rows to last** check box and entering a value in its data field. The Scope relies on its data history for zooming and auto-scaling operations. If the number of rows is limited to 1,000 and the simulation generates 2,000 rows, only the last 1,000 are available for regenerating the display.

### Save data to workspace

You can automatically save the data collected by the Scope at the end of the simulation by checking the **Save data to workspace** check box. If you check this option, then the **Variable name** and **Format** fields become active.

### Variable name

Enter a variable name in the **Variable name** field. The specified name must be unique among all data logging variables being used in the model. Other data logging variables are defined on other Scope blocks, To Workspace blocks, and simulation return variables such as time, states, and outputs. Being able to save Scope data to the workspace means that it

# Scope

---

is not necessary to send the same data stream to both a Scope block and a To Workspace block.

## Format

Data can be saved in one of three formats: **Matrix**, **Structure**, or **Structure with time**. Use **Matrix** only for a Scope with one axes. For Scopes with more than one axes, use **Structure** if you do not want to store time data and use **Structure with time** if you want to store time data. See Data Logging.

## Printing the Contents of a Scope Window

To print the contents of a Scope window, open the **Scope Properties** dialog by clicking on the **Print** icon, the right-most icon on the Scope toolbar.



the Print icon

## Data Type Support

A Scope block accepts real signals, including homogenous vectors, of any type.

## Characteristics

Sample Time	Inherited from driving block or settable
States	0

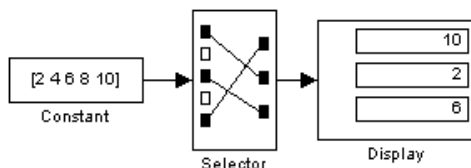
**Purpose** Select input elements.

**Library** Signals & Systems

**Description** The Selector block generates as output selective elements of the input vector.



The **Elements** parameter defines the order of the input vector elements in the output vector. The parameter must be specified as a vector unless only one element is being selected. For example, this model shows the Selector block icon and the output for an input vector of [2 4 6 8 10] and an **Elements** parameter value of [5 1 3].

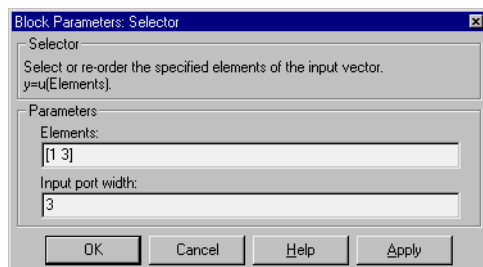


The block icon displays the ordering of input vector elements graphically. If the block is not large enough, it displays the block name.

The Selector block accepts signals of any signal and data type, including mixed-type signal vectors. The elements of the output vector have the same type as the corresponding selected input elements.

**Data Type Support** A Selector block accepts and outputs signals of any numeric type (real or complex) and data type.

## Parameters and Dialog Box



## Elements

The order that the input elements are to appear in the output vector.

# Selector

---

## **Input port width**

The number of elements in the input vector.

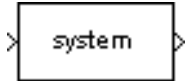
## **Characteristics**

Sample Time	Inherited from driving block
Vectorized	Yes

**Purpose** Access an S-function.

**Library** Functions & Tables

**Description** The S-Function block provides access to S-functions from a block diagram. The S-function named as the **S-function name** parameter can be an M-file or MEX-file written as an S-function.



The S-Function block allows additional parameters to be passed directly to the named S-function. The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example,

```
A, B, C, D, [eye(2,2);zeros(2,2)]
```

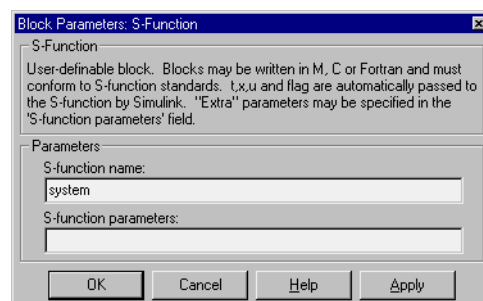
Note that although individual parameters can be enclosed in square brackets, the list of parameters must not be enclosed in square brackets.

The S-Function block displays the name of the specified S-function and is always drawn with one input port and one output port, regardless of the number of inputs and outputs of the contained subsystem.

Vector lines are used when the S-function contains more than one input or output. The input vector width must match the number of inputs contained in the S-function. The block directs the first element of the input vector to the first input of the S-function, the second element to the second input, and so on. Likewise, the output vector width must match the number of S-function outputs.

**Data Type Support** Depends on the implementation of the S-Function block.

## Parameters and Dialog Box



# S-Function

---

## **S-function name**

The S-function name.

## **S-function parameters**

Additional S-function parameters.

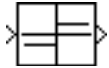
<b>Characteristics</b>	Direct Feedthrough	Depends on contents of S-function
	Sample Time	Depends on contents of S-function
	Scalar Expansion	Depends on contents of S-function
	Vectorized	Depends on contents of S-function
	Zero Crossing	No



**Purpose** Indicate the sign of the input.

**Library** Math

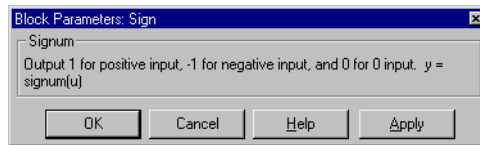
**Description** The Sign block indicates the sign of the input:



- The output is 1 when the input is greater than zero.
- The output is 0 when the input is equal to zero.
- The output is -1 when the input is less than zero.

**Data Type Support** A Sign block accepts and outputs real signals of type double.

**Dialog Box**



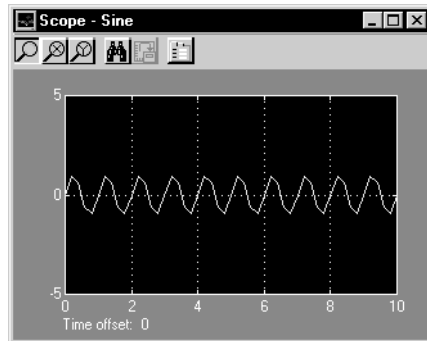
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	N/A
	Vectorized	Yes
	Zero Crossing	Yes, to detect when the input crosses through zero

# Signal Generator

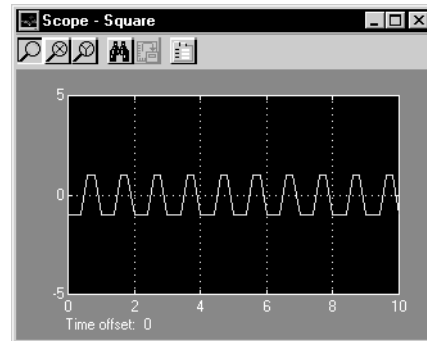
**Purpose** Generate various waveforms.

**Library** Sources

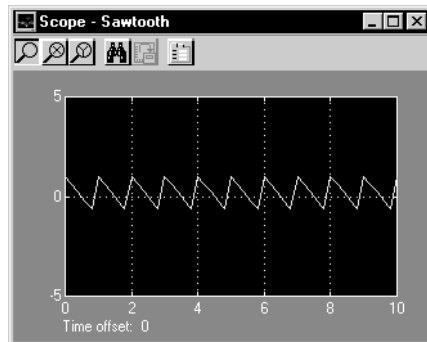
**Description** The Signal Generator block can produce one of three different waveforms: sine wave, square wave, and sawtooth wave. The signal parameters can be expressed in Hertz (the default) or radians per second. This figure shows each signal displayed on a Scope using default parameter values.



Sine Wave



Square Wave



Sawtooth Wave

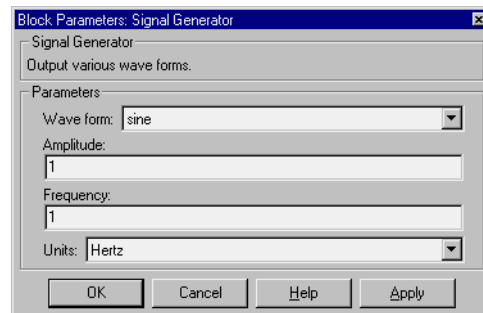
A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in a variety of ways, including inputting a Clock block signal to a MATLAB Fcn block and writing the equation for the particular wave.

You can vary the output settings of the Signal Generator block while a simulation is in progress. This is useful to determine quickly the response of a system to different types of inputs.

## Data Type Support

A Signal Generator block outputs real signals of type double.

## Parameters and Dialog Box



### Wave form

The wave form: a sine wave, square wave, or sawtooth wave. The default is a sine wave.

### Amplitude

The signal amplitude. The default is 1.

### Frequency

The signal frequency. The default is 1.

### Units

The signal units, Hertz or radians/sec. The default is Hertz.

## Characteristics

Sample Time	Inherited
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	No

# Sine Wave

---

**Purpose** Generate a sine wave.

**Library** Sources

**Description** The Sine Wave block provides a sinusoid. The block can operate in either continuous or discrete mode.



The output of the Sine Wave block is determined by:

$$y = \textit{Amplitude} \times \sin(\textit{frequency} \times \textit{time} + \textit{phase})$$

The value of the **Sample time** parameter determines whether the block operates in continuous mode or discrete mode:

- 0 (the default) causes the block to operate in continuous mode.
- >0 causes the block to operate in discrete mode.
- -1 causes the block to operate in the same mode as the block receiving the signal.

## Using the Sine Wave Block in Discrete Mode

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

Using the Sine Wave block in this way allows you to build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and, as a result, take longer to simulate.

The Sine Wave block in discrete mode uses an incremental algorithm rather than one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The incremental algorithm computes the sine based on the value computed at the previous sample time. This method makes use of the following identities:

$$\sin(t + \Delta t) = \sin(t)\cos(\Delta t) + \sin(\Delta t)\cos(t)$$

$$\cos(t + \Delta t) = \cos(t)\cos(\Delta t) - \sin(t)\sin(\Delta t)$$

These identities can be written in matrix form.

$$\begin{bmatrix} \sin(t + \Delta t) \\ \cos(t + \Delta t) \end{bmatrix} = \begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}$$

Since  $\Delta t$  is constant, the following expression is a constant.

$$\begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix}$$

Therefore the problem becomes one of a matrix multiply of the value of  $\sin(t)$  by a constant matrix to obtain  $\sin(t+\Delta t)$ . This algorithm may also be faster on computers that do not have hardware floating-point support for trigonometric functions.

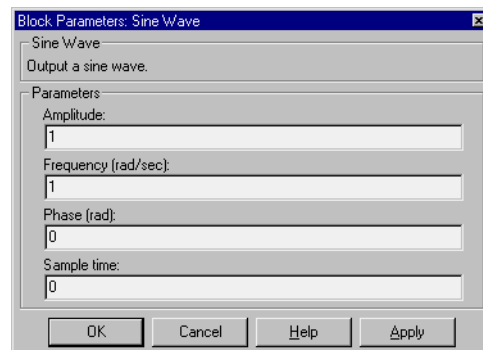
## Using the Sine Wave Block in Continuous Mode

A **Sample time** parameter value of zero causes the block to behave in continuous mode. When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.

## Data Type Support

A Sine Wave block accepts and outputs real signals of type double.

## Parameters and Dialog Box



## Amplitude

The amplitude of the signal. The default is 1.

# Sine Wave

---

## Frequency

The frequency, in radians/second. The default is 1 rad/sec.

## Phase

The phase shift, in radians. The default is 0 radians.

## Sample time

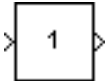
The sample period. The default is 0.

<b>Characteristics</b>	Sample Time	Continuous, discrete, or inherited
	Scalar Expansion	Of parameters
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Vary a scalar gain using a slider.

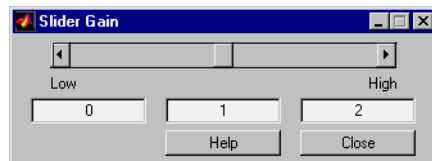
**Library** Math

**Description** The Slider Gain block allows you to vary a scalar gain during a simulation using a slider. The block accepts one input and generates one output.



**Data Type Support** Data type support for the Slider Gain block is the same as that for the Gain block (see Gain on page 8-89).

**Dialog Box**



## Low

The lower limit of the slider range. The default is 0.

## High

The upper limit of the slider range. The default is 2.

The edit fields indicate (from left to right) the lower limit, the current value, and the upper limit. You can change the gain in two ways: by manipulating the slider, or by entering a new value in the current value field. You can change the range of gain values by changing the lower and upper limits. Close the dialog box by clicking on the **Close** button.

If you click on the slider's left or right arrow, the current value changes by about 1% of the slider's range. If you click on the rectangular area to either side of the slider's indicator, the current value changes by about 10% of the slider's range.

To apply a vector gain to the block input, consider using the Gain block, described on page Gain on page 8-89. To apply a matrix gain, use the Matrix Gain block, described on Matrix Gain on page 8-123.

# Slider Gain

---

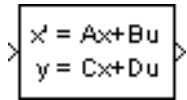
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of the gain
	States	0
	Vectorized	Yes
	Zero Crossing	No



**Purpose** Implement a linear state-space system.

**Library** Continuous

**Description** The State-Space block implements a system whose behavior is defined by:

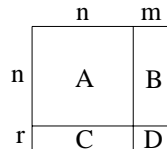


$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

where  $x$  is the state vector,  $u$  is the input vector, and  $y$  is the output vector. The matrix coefficients must have these characteristics, as illustrated in the diagram below:

- **A** must be an  $n$ -by- $n$  matrix, where  $n$  is the number of states.
- **B** must be an  $n$ -by- $m$  matrix, where  $m$  is the number of inputs.
- **C** must be an  $r$ -by- $n$  matrix, where  $r$  is the number of outputs.
- **D** must be an  $r$ -by- $m$  matrix.



The block accepts one input and generates one output. The input vector width is determined by the number of columns in the B and D matrices. The output vector width is determined by the number of rows in the C and D matrices.

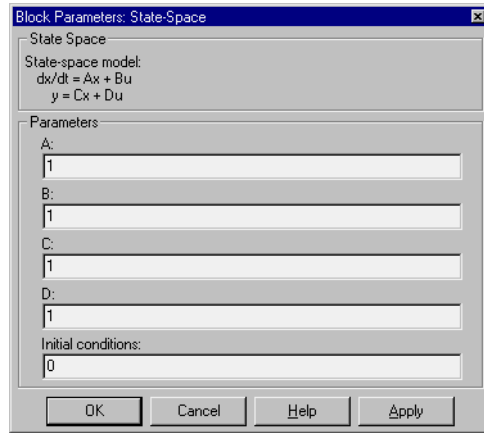
Simulink converts a matrix containing zeros to a sparse matrix for efficient multiplication.

## Data Type Support

A State-Space block accepts and outputs real signals of type double.

# State-Space

## Parameters and Dialog Box



### A, B, C, D

The matrix coefficients.

### Initial conditions

The initial state vector.

<b>Characteristics</b>	Direct Feedthrough	Only if $D \neq 0$
	Sample Time	Continuous
	Scalar Expansion	Of the initial conditions
	States	Depends on the size of A
	Vectorized	Yes
	Zero Crossing	No

**Purpose** Generate a step function.

**Library** Sources

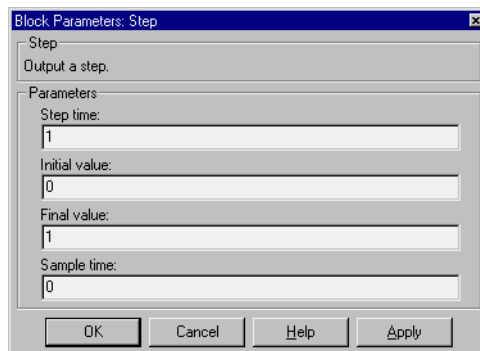
**Description** The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the **Step time** parameter value, the block's output is the **Initial value** parameter value. For simulation time greater than or equal to the **Step time**, the output is the **Final value** parameter value.



The Step block generates a scalar or vector output, depending on the length of the parameters.

**Data Type Support** A Step block outputs real signals of type double.

### Parameters and Dialog Box



#### Step time

The time, in seconds, when the output jumps from the **Initial value** parameter to the **Final value** parameter. The default is 1 second.

#### Initial value

The block output until the simulation time reaches the **Step time** parameter. The default is 0.

#### Final value

The block output when the simulation time reaches and exceeds the **Step time** parameter. The default is 1.

# Step

---

## Sample time

Sample rate of step.

## Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	Of parameters
Vectorized	Yes
Zero Crossing	Yes, to detect step times

**Purpose** Stop the simulation when the input is nonzero.

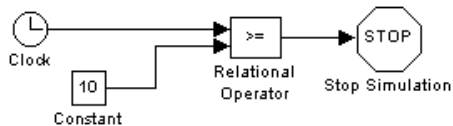
**Library** Sinks

**Description** The Stop Simulation block stops the simulation when the input is nonzero.



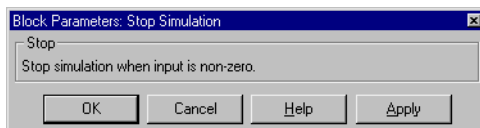
The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

You can use this block in conjunction with the Relational Operator block to control when the simulation stops. For example, this model stops the simulation when the input signal reaches 10.



**Data Type Support** A Stop Simulation block accepts real signals of type double.

**Dialog Box**



**Characteristics** Sample Time Inherited from driving block

Vectorized Yes

# Subsystem

---

**Purpose** Represent a system within another system.

**Library** Signals & Systems

**Description** A Subsystem block represents a system within another system. You create a subsystem in these ways:



- Copy the Subsystem block from the Connections library into your model. You can then add blocks to the subsystem by opening the Subsystem block and copying blocks into its window.
- Select the blocks and lines that are to make up the subsystem using a bounding box, then choose **Create Subsystem** from the **Edit** menu. Simulink replaces the blocks with a Subsystem block. When you open the block, the window displays the blocks you selected, adding Inport and Outport blocks to reflect signals entering and leaving the subsystem.

The number of input ports drawn on the Subsystem block's icon corresponds to the number of Inport blocks in the subsystem. Similarly, the number of output ports drawn on the block corresponds to the number of Outport blocks in the subsystem. If Inport and Outport block names are not hidden, they appear as port labels on the Subsystem block.

For more information about subsystems, see “Creating Subsystems” in Chapter 3.

**Data Type Support** See Inport on page 8-99 and Outport on page 8-139.

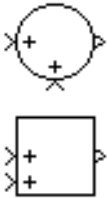
**Dialog Box** None

<b>Characteristics</b>	Sample Time	Depends on the blocks in the subsystem
	Vectorized	Depends on the blocks in the subsystem
	Zero Crossing	Yes, for enable and trigger ports if present

**Purpose** Generate the sum of inputs.

**Library** Math

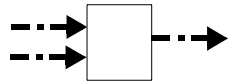
**Description** The Sum block adds scalar and/or vector inputs, or elements of a single vector input, depending on the number of block inputs:



- If the block has more than one input, the block output is an element-by-element sum across the inputs. If all inputs are scalars, the output is a scalar. For a block having  $n$  inputs, if any input is a vector, each element of the output is generated as

$$:y_i = u1_i + u2_i + \dots + un_i$$

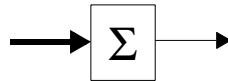
This model represents using the Sum block in this way. The broken lines indicate that each signal can be a scalar or vector. The output is a scalar only if all inputs are scalars.



- If the block has one vector input, the block output is the scalar sum of the elements of the input:

$$y = \Sigma u_i$$

This model represents using the Sum block in this way. The solid line input signal indicates that the input is a vector.




---

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

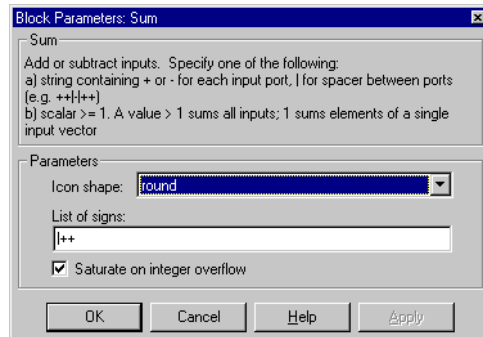
---

**Data Type Support**

The Sum block accepts real- or complex-valued signals of any data type. All the inputs must be of the same data type. The output data type is the same as the input data type.

# Sum

## Parameters and Dialog Box



### Icon shape

You can choose a circular or rectangular shape for the Sum block in the **Icon shape** drop box. If the Sum block has multiple inputs, it may be more convenient to have a circular shape than a rectangular shape.

### List of signs

The **List of signs** parameter can have a constant or a combination of +, -, and | symbols. Specifying a constant causes Simulink to redraw the block with that number of ports, all with positive polarity. A combination of plus and minus signs specifies the polarity of each port, where the number of ports equals the number of symbols used.

The Sum block draws plus and minus signs beside the appropriate ports and redraws its ports to match the number of signs specified in the **List of signs** parameter. If the number of signs is changed, ports are added or deleted from the icon. If necessary, Simulink resizes the block to show all input ports. You can also manipulate the position of the input ports by inserting spacers (|) between the signs in the **List of signs** parameter. The spacers create extra space between the ports. For example, ++|-- will create an extra space between the second + port and the first - port:

### Saturate on integer overflow

If selected, this option causes the output of the Sum block to saturate on integer overflow. In particular, if the output data type is an integer type, the block output is the maximum value representable by the output type or the computed output, whichever is smaller in the absolute sense. If the option is not selected, Simulink takes the action specified by **Data**



---

**overflow** event option on the **Diagnostics** page of the **Simulation Parameters** dialog (see “The Diagnostics Page” on page 4–24).

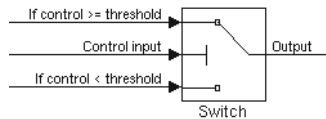
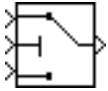
<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving blocks
	Scalar Expansion	Yes
	States	0
	Vectorized	Yes
	Zero Crossing	No

# Switch

**Purpose** Switch between two inputs.

**Library** Nonlinear

**Description** The Switch block propagates one of two inputs to its output depending on the value of a third input, called the control input. If the signal on the control (second) input is greater than or equal to the **Threshold** parameter, the block propagates the first input; otherwise, it propagates the third input. This figure shows the use of the block ports.

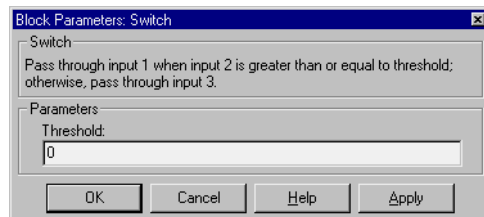


To drive the switch with a logic input (i.e., 0 or 1), set the threshold to 0.5.

## Data Type Support

A Switch block accepts real- or complex-valued signals of any data type as switched inputs (inputs 1 and 3). Both switched inputs must be of the same type. The block output signal has the data type of the selected input. The data type of the threshold input must be `bool` or `double`.

## Parameters and Dialog Box



### Threshold

The value of the control (the second input) at which the switch flips to its other state. You can specify this parameter as either a scalar or a vector equal in width to the input vectors.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Yes

Vectorized	Yes
Zero Crossing	Yes, to detect when the switch condition occurs

# Terminator

---

**Purpose** Terminate an unconnected output port.

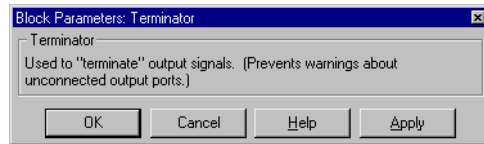
**Library** Signals & Systems

**Description** The Terminator block can be used to cap blocks whose output ports are not connected to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks avoids warning messages.



**Data Type Support** A Terminator block accepts signals of any numeric type or data type.

**Parameters and Dialog Box**



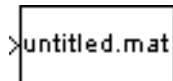
**Characteristics**

Sample Time	Inherited from driving block
Vectorized	Yes

**Purpose** Write data to a file.

**Library** Sinks

**Description**



The To File block writes its input to a matrix in a MAT-file. The block writes one column for each time step: the first row is the simulation time; the remainder of the column is the input data, one data point for each element in the input vector. The matrix has this form:

$$\begin{bmatrix} t_1 & t_2 & \dots & t_{final} \\ u1_1 & u1_2 & \dots & u1_{final} \\ \dots & \dots & \dots & \dots \\ un_1 & un_2 & \dots & un_{final} \end{bmatrix}$$

The From File block can use data written by a To File block without any modifications. However, the form of the matrix expected by the From Workspace block is the transpose of the data written by the To File block.

The block writes the data as well as the simulation time after the simulation is completed. The block icon shows the name of the specified output file.

The amount of data written and the time steps at which the data is written are determined by block parameters:

- The **Decimation** parameter allows you to write data at every  $n$ th sample, where  $n$  is the decimation factor. The default decimation, 1, writes data at every time step.
- The **Sample time** parameter allows you to specify a sampling interval at which to collect points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining which points to write.

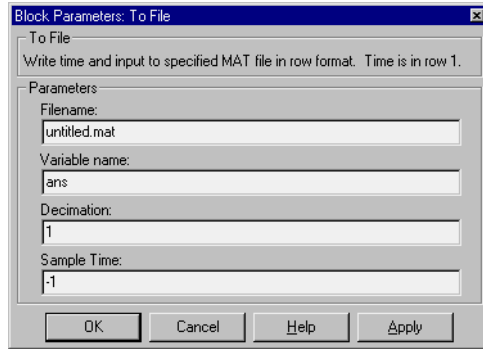
If the file exists at the time the simulation starts, the block overwrites its contents.

**Data Type Support**

A To File block accepts real signals of type double.

# To File

## Parameters and Dialog Box



### Filename

The name of the MAT-file that holds the matrix.

### Variable name

The name of the matrix contained in the named file.

### Decimation

A decimation factor. The default value is 1.

### Sample time

The sample time at which to collect points.

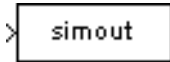
## Characteristics

Sample Time	Inherited from driving block
Vectorized	Yes

**Purpose** Write data to the workspace.

**Library** Sinks

**Description** The To Workspace block writes its input to the workspace. The block writes its output to a matrix or structure that has the name specified by the block's **Variable name** parameter. The **Save format** parameter determines the output format:



## Matrix

The matrix has this form:

$$\begin{bmatrix} u1_1 & u2_1 & \dots & un_1 \\ u1_2 & u2_2 & \dots & un_2 \\ \dots & & & \\ u1_{final} & u2_{final} & \dots & un_{final} \end{bmatrix}$$

The amount of data written and the time steps at which the data is written are determined by block parameters:

- The **Maximum number of rows** parameter indicates how many data rows to save. If the simulation generates more rows than the specified maximum, the simulation saves only the most recently generated rows. To capture all the data, set this value to `inf`.
- The **Decimation** parameter allows you to write data at every  $n$ th sample, where  $n$  is the decimation factor. The default decimation, 1, writes data at every time step.
- The **Sample time** parameter allows you to specify a sampling interval at which to collect points. This parameter is useful when using a variable-step solver where the interval between time steps may not be the same. The default value of -1 causes the block to inherit the sample time from the driving block when determining which points to write.

During the simulation, the block writes data to an internal buffer. When the simulation is completed or paused, that data is written to the workspace. The block icon shows the name of the matrix to which the data is written.

# To Workspace

---

## Structure

This format consists of a structure with three fields: time, signals, and blockName. The time field is empty. The blockName field contains the name of the To Workspace block. The signals field contains a structure with two fields: values and label. The values field contains the matrix of signal values.

## Structure with Time

This format is the same as Structure except that the time field contains a vector of simulation time steps.

## Using Saved Data with a From File Block

If the data written using a To Workspace block is to be saved and read later by a From File block, the time must be added to the data and the matrix must be transposed. For more information, see From File on page 8-82.

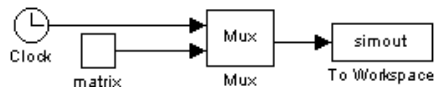
## Using Saved Data with a From Workspace Block

If the data written using a To Workspace block is intended to be “played back” in another simulation using a From Workspace block, the data must contain the simulation time values. The way you include times depends on the save format.

If the save format is a structure, you can include the simulation time by choosing Structure with Time as the value of **Save format**. The block stores the simulation times as a vector in the time member of the output structure.

If the save format is a matrix, you must add a column of simulation times to the matrix. You can add a column with time values in two ways:

- By multiplexing the output of a Clock block as the first element of the vector input line of the To Workspace block.



- By specifying time as a return value on the **Simulation Parameters** dialog box or from the command line, described in Chapter 4. When the simulation



is completed, you can concatenate the time vector (t) to the matrix using a command like this:

```
matrix = [t; matrix];
```

## Examples

In a simulation where the start time is 0, the **Maximum number of rows** is 100, the **Decimation** is 1, and the **Sample time** is 0.5. The To Workspace block collects a maximum of 100 points, at time values of 0, 0.5, 1.0, 1.5, ... seconds. Specifying a **Decimation** of 1 directs the block to write data at each step.

In a similar example, the **Maximum number of rows** is 100 and the **Sample time** is 0.5, but the **Decimation** is 5. In this example, the block collects up to 100 points, at time values of 0, 2.5, 5.0, 7.5, ... seconds. Specifying a **Decimation** of 5 directs the block to write data at every fifth sample. The sample time ensures that data is written at these points.

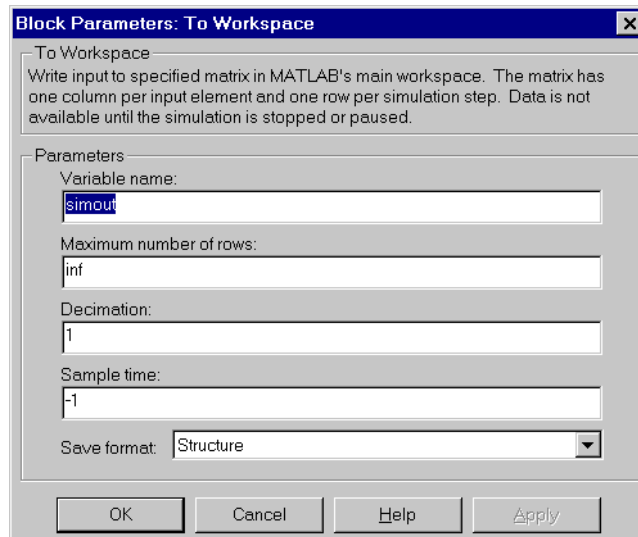
In another example, all parameters are as defined in the first example except that the **Maximum number of rows** is 3. In this case, only the last three rows collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

## Data Type Support

A To Workspace block can save input of any real or complex data type to the MATLAB workspace.

# To Workspace

## Parameters and Dialog Box



### Variable name

The name of the matrix that holds the data.

### Maximum number of rows

The maximum number of rows (one row per time step) to be saved. The default is 1000 rows.

### Decimation

A decimation factor. The default is 1.

### Sample time

The sample time at which to collect points.

### Save format

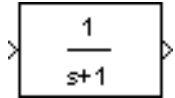
Format in which to save simulation output to the workspace. The default is as a structure.

<b>Characteristics</b>	Sample Time	Inherited
	Vectorized	Yes

**Purpose** Implement a linear transfer function.

**Library** Continuous

**Description** The Transfer Fcn block implements a transfer function where the input ( $u$ ) and output ( $y$ ) can be expressed in transfer function form as the following equation



$$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \dots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \dots + den(nd)}$$

where  $nn$  and  $nd$  are the number of numerator and denominator coefficients, respectively.  $num$  and  $den$  contain the coefficients of the numerator and denominator in descending powers of  $s$ .  $num$  can be a vector or matrix,  $den$  must be a vector, and both are specified as parameters on the block dialog box. The order of the denominator must be greater than or equal to the order of the numerator.

A Transfer Fcn block takes a scalar input. If the numerator of the block's transfer function is a vector, the block's output is also scalar. However, if the numerator is a matrix, the transfer function expands the input into an output vector equal in width to the number of rows in the numerator. For example, a two-row numerator results in a block with scalar input and vector output. The width of the output vector is two.

Initial conditions are preset to zero. If you need to specify initial conditions, convert to state-space form using `tf2ss` and use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system. For more information, type `help tf2ss` or consult the *Control System Toolbox User's Guide*.

## The Transfer Fcn Block Icon

The numerator and denominator are displayed on the Transfer Fcn block icon depending on how they are specified:

- If each is specified as an expression, a vector, or a variable enclosed in parentheses, the icon shows the transfer function with the specified coefficients and powers of  $s$ . If you specify a variable in parentheses, the variable is evaluated. For example, if you specify **Numerator** as `[3, 2, 1]` and

# Transfer Fcn

**Denominator** as (den) where den is [7,5,3,1], the block icon looks like this:

$$\frac{3s^2+2s+1}{7s^3+5s^2+3s+1}$$

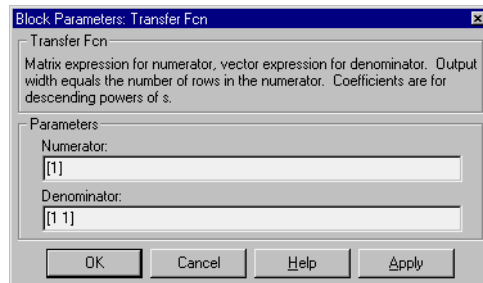
- If each is specified as a variable, the icon shows the variable name followed by “(s)”. For example, if you specify **Numerator** as num and **Denominator** as den, the block icon looks like this:

$$\frac{\text{num}(s)}{\text{den}(s)}$$

## Data Type Support

A Transfer Fcn block accepts and outputs signals of any data type.

## Parameters and Dialog Box



### Numerator

The row vector of numerator coefficients. A matrix with multiple rows can be specified to generate multiple output. The default is [ 1 ].

### Denominator

The row vector of denominator coefficients. The default is [ 1 1 ].

## Characteristics

Direct Feedthrough	Only if the lengths of the <b>Numerator</b> and <b>Denominator</b> parameters are equal
Sample Time	Continuous
Scalar Expansion	No
States	Length of <b>Denominator</b> -1

Vectorized	Yes, in the sense that the block expands scalar input into vector output when the transfer function numerator is a matrix. See block description above.
Zero Crossing	No

# Transport Delay

---

**Purpose** Delay the input by a given amount of time.

**Library** Continuous

**Description** The Transport Delay block delays the input by a specified amount of time. It can be used to simulate a time delay.



At the start of the simulation, the block outputs the **Initial input** parameter until the simulation time exceeds the **Time delay** parameter, when the block begins generating the delayed input. The **Time delay** parameter must be nonnegative.

The block stores input points and simulation times during a simulation in a buffer whose initial size is defined by the **Initial buffer size** parameter. If the number of points exceeds the buffer size, the block allocates additional memory and Simulink displays a message after the simulation that indicates the total buffer size needed. Because allocating memory slows down the simulation, define this parameter value carefully if simulation speed is an issue. For long time delays, this block might use a large amount of memory, particularly for a vectorized input.

When output is required at a time that does not correspond to the times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which may produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate its output value. To illustrate this point, consider a fixed-step simulation with a step size of 1 and the current time at  $t = 5$ . If the delay is 0.5, the block needs to generate a point at  $t = 4.5$ . Because the most recent stored time value is at  $t = 4$ , the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, it returns the discrete value at  $t - t_{delay}$ .

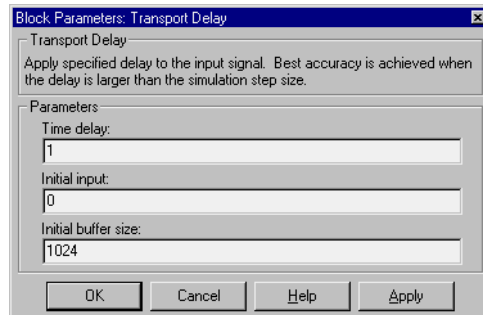
This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Using `linmod` to linearize a model that contains a Transport Delay block can be troublesome. For more information about ways to avoid the problem, see “Linearization” in Chapter 5.

## Data Type Support

A Transport Delay block accepts and outputs real signals of type `double`.

## Parameters and Dialog Box



### Time delay

The amount of simulation time that the input signal is delayed before propagating it to the output. The value must be nonnegative.

### Initial input

The output generated by the block between the start of the simulation and the **Time delay**.

### Initial buffer size

The initial memory allocation for the number of points to store.

## Characteristics

Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	Of input and all parameters except <b>Initial buffer size</b>
Vectorized	Yes
Zero Crossing	No

# Trigger

---

**Purpose** Add a trigger port to a subsystem.

**Library** Signals & Systems

## Description



Adding a Trigger block to a subsystem makes it a triggered subsystem. A triggered subsystem executes once on each integration step when the value of the signal that passes through the trigger port changes in a specifiable way (described below). A subsystem can contain no more than one Trigger block. For more information about triggered subsystems, see Chapter 7.

The **Trigger type** parameter allows you to choose the type of event that triggers execution of the subsystem:

- **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.
- **function-call** causes execution of the subsystem to be controlled by logic internal to an S-function (for more information, see “Function-Call Subsystems” in Chapter 7).

You can output the trigger signal by selecting the **Show output port** check box. Selecting this option allows the system to determine what caused the trigger. The width of the signal is the width of the triggering signal. The signal value is:

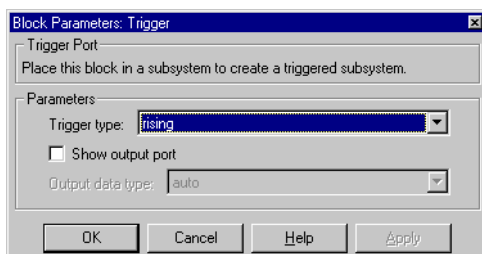
- 1 for a signal that causes a rising trigger
- -1 for a signal that causes a falling trigger
- 0 otherwise

## Data Type Support

A Trigger block accepts signals of type boolean or double.



## Parameters and Dialog Box



### Trigger type

The type of event that triggers execution of the subsystem

### Show output port

If checked, Simulink draws the Trigger block output port and outputs the trigger signal.

### Output data type

Specifies the data type (double or int8) of the trigger output. If you select auto, Simulink sets the data type to be the same as that of the port to which the output is connected. If the port's data type is not double or int8, Simulink signals an error.

## Characteristics

Sample Time	Determined by the signal at the trigger port
Vectorized	Yes

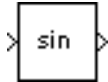
# Trigonometric Function

---

**Purpose** Perform a trigonometric function.

**Library** Math

**Description** The Trigonometric Function block performs numerous common trigonometric functions.



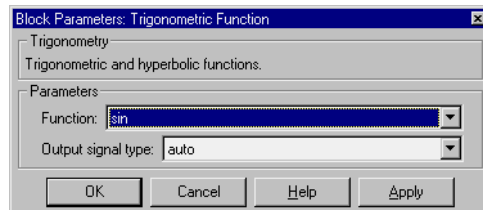
You can select one of these functions from the **Function** list: sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, and tanh. The block output is the result of the function operating on the input or inputs.

The name of the function appears on the block icon. Simulink automatically draws the appropriate number of input ports. The block accepts and outputs real or complex signals of type double.

Use the Trigonometric Function block instead of the Fcn block when you want vectorized output because the Fcn block can produce only scalar output.

**Data Type Support** A Trigonometric Function block accepts and outputs real or complex signals of type double.

## Parameters and Dialog Box



**Function** The trigonometric function.

**Output signal type** Type of signal (complex or real) to output.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	Of the input when the function requires two inputs

Vectorized	Yes
Zero Crossing	No

# Uniform Random Number

---

**Purpose** Generate uniformly distributed random numbers.

**Library** Sources

**Description** The Uniform Random Number block generates uniformly distributed random numbers over a specifiable interval with a specifiable starting seed. The seed is reset each time a simulation starts. The generated sequence is repeatable and can be produced by any Uniform Random Number block with the same seed and parameters. To generate a vector of random numbers, specify the **Initial seed** parameter as a vector.



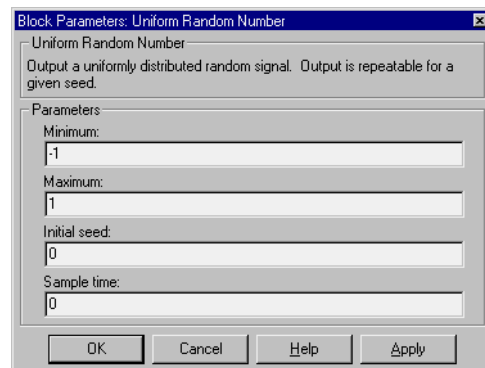
To generate normally distributed random numbers, use the Random Number block, described on Random Number on page 8-150.

Avoid integrating a random signal because solvers are meant to integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

## Data Type Support

A Uniform Random Number block outputs a real signal of type double.

## Parameters and Dialog Box



### Minimum

The minimum of the interval. The default is -1.

### Maximum

The maximum of the interval. The default is 1.

### Initial seed

The starting seed for the random number generator. The default is 0.

## Sample time

The sample period. The default is 0.

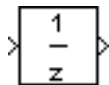
<b>Characteristics</b>	Sample Time	Continuous, discrete, or inherited
	Scalar Expansion	No
	Vectorized	Yes
	Zero Crossing	No

# Unit Delay

**Purpose** Delay a signal one sample period.

**Library** Discrete

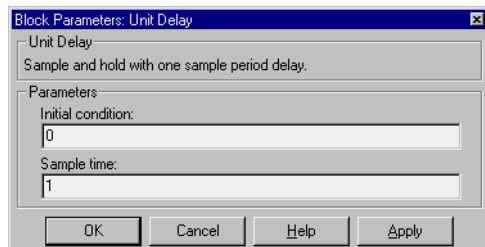
**Description** The Unit Delay block delays and holds its input signal by one sampling interval. If the input to the block is a vector, all elements of the vector are delayed by the same sample delay. This block is equivalent to the  $z^{-1}$  discrete-time operator.



If an undelayed sample-and-hold function is desired, use a Zero-Order Hold block, or if a delay of greater than one unit is desired, use a Discrete Transfer Fcn block. (See the description of the Transport Delay block for an example that uses the Unit Delay block.)

**Data Type Support** A Unit block accepts real or complex signals of data type, including user-defined types. If the data type of the input signal is user-defined, the initial condition must be 0

## Parameters and Dialog Box



### Initial condition

The block output for the first simulation period, during which the output of the Unit Delay block is undefined. Careful selection of this parameter can minimize unwanted output behavior during this time. The default is 0.

### Sample time

The time interval between samples. The default is 1.

<b>Characteristics</b>	Direct Feedthrough	No
	Sample Time	Discrete
	Scalar Expansion	Of the <b>Initial condition</b> parameter or the input

States	Inherited from driving block or parameters
Vectorized	Yes
Zero Crossing	No

# Variable Transport Delay

**Purpose** Delay the input by a variable amount of time.

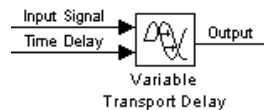
**Library** Continuous

## Description



The Variable Transport Delay block can be used to simulate a variable time delay. The block might be used to model a system with a pipe where the speed of a motor pumping fluid in the pipe is variable.

The block accepts two inputs: the first input is the signal that passes through the block; the second input is the time delay, as show in this icon.



The **Maximum delay** parameter defines the largest value the time delay input can have. The block clips values of the delay that exceed this value. The **Maximum delay** must be greater than or equal to zero. If the time delay becomes negative, the block clips it to zero and issues a warning message.

During the simulation, the block stores time and input value pairs in an internal buffer. At the start of the simulation, the block outputs the **Initial input** parameter until the simulation time exceeds the time delay input. Then, at each simulation step the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

When output is required at a time that does not correspond to the times of the stored input values, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point. This may result in less accurate results. The block cannot use the current input to calculate its output value because the block does not have direct feedthrough at this port. To illustrate this point, consider a fixed-step simulation with a step size of 1 and the current time at  $t = 5$ . If the delay is 0.5, the block needs to generate a point at  $t = 4.5$ . Because the most recent stored time value is at  $t = 4$ , the block performs forward extrapolation.

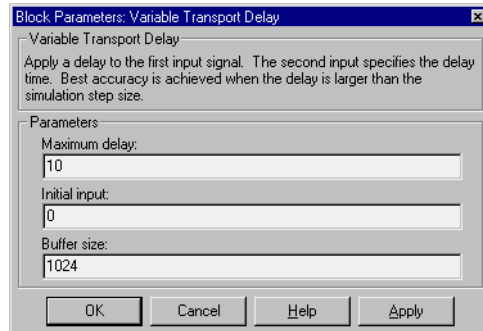
The Variable Transport Delay block does not interpolate discrete signals. Instead, it returns the discrete value at  $t - t_{delay}$ .



## Data Type Support

A Variable Transport Delay block accepts and outputs real signals of type double.

## Parameters and Dialog Box



### Maximum delay

The maximum value of the time delay input. The value cannot be negative. The default is 10.

### Initial input

The output generated by the block until the simulation time first exceeds the time delay input. The default is 0.

### Buffer size

The number of points the block can store. The default is 1024.

## Characteristics

Direct Feedthrough	Yes, of the time delay (second) input
Sample Time	Continuous
Scalar Expansion	Of input and all parameters except <b>Buffer size</b>
Vectorized	Yes
Zero Crossing	No

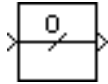
# Width

---

**Purpose** Output the width of the input vector.

**Library** Signals & Systems

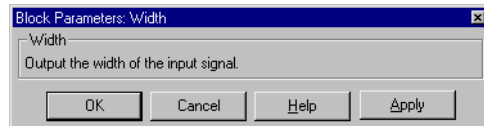
**Description** The Width block generates as output the width of its input vector.



The Width block accepts real- or complex-valued signals of any data type, including mixed-type signal vectors.

**Data Type Support** A Width block accepts and outputs real signals of type double.

## Parameters and Dialog Box



<b>Characteristics</b>	Sample Time	Constant
	Vectorized	Yes

**Purpose** Display an X-Y plot of signals using a MATLAB figure window.

**Library** Sinks

**Description** The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.



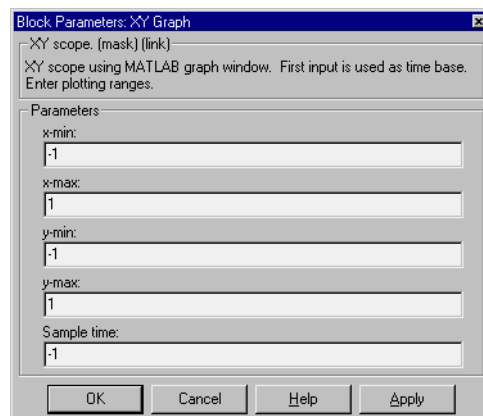
The block has two scalar inputs. The block plots data in the first input (the  $x$  direction) against data in the second input (the  $y$  direction). This block is useful for examining limit cycles and other two-state data. Data outside the specified range is not displayed.

Simulink opens a figure window for each XY Graph block in the model at the start of the simulation.

For a demo that illustrates the use of the XY Graph block, enter `lorenz`s in the command window.

**Data Type Support** An XY Graph block accepts real signals of type double.

## Parameters and Dialog Box



### **x-min**

The minimum x-axis value. The default is -1.

### **x-max**

The maximum x-axis value. The default is 1.

# XY Graph

---

## **y-min**

The minimum y-axis value. The default is -1.

## **y-max**

The maximum y-axis value. The default is 1.

## **Sample time**

The time interval between samples. The default is -1, which means that the sample time is determined by the driving block.

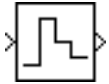
## **Characteristics**

Sample Time	Inherited from driving block
States	0

**Purpose** Implement zero-order hold of one sample period.

**Library** Discrete

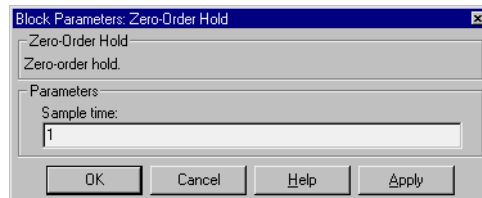
**Description** The Zero-Order Hold block implements a sample-and-hold function operating at the specified sampling rate. The block accepts one input and generates one output, both of which can be scalar or vector..



This block provides a mechanism for discretizing one or more signals or resampling the signal at a different rate. You can use it in instances where you need to model sampling without requiring one of the other more complex discrete function blocks. For example, it could be used in conjunction with a Quantizer block to model an A/D converter with an input amplifier.

**Data Type Support** A Zero-Order Hold block accepts real- or complex-valued signals of any data type.

## Parameters and Dialog Box



### Sample time

The time interval between samples. The default is 1.

<b>Characteristics</b>	Direct Feedthrough	Yes
	Sample Time	Discrete
	Scalar Expansion	Yes
	States	0
	Vectorized	Yes
	Zero Crossing	No

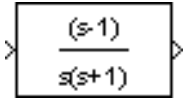
# Zero-Pole

---

**Purpose** Implement a transfer function specified in terms of poles and zeros.

**Library** Continuous

**Description** The Zero-Pole block implements a system with the specified zeros, poles, and gain in terms of the Laplace operator  $s$ .



A transfer function can be expressed in factored or zero-pole-gain form, which, for a single-input single-output system in MATLAB, is

$$H(s) = K \frac{Z(s)}{P(s)} = K \frac{(s - Z(1))(s - Z(2)) \dots (s - Z(m))}{(s - P(1))(s - P(2)) \dots (s - P(n))}$$

where  $Z$  represents the zeros vector,  $P$  the poles vector, and  $K$  the gain.  $Z$  can be a vector or matrix,  $P$  must be a vector,  $K$  can be a scalar or vector whose length equals the number of rows in  $Z$ . The number of poles must be greater than or equal to the number of zeros. If the poles and zeros are complex, they must be complex conjugate pairs.

Block input and output widths are equal to the number of rows in the zeros matrix.

## The Zero-Pole Block Icon

The Zero-Pole block displays the transfer function in its icon depending on how the parameters are specified:

- If each is specified as an expression or a vector, the icon shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the variable is evaluated.

For example, if you specify **Zeros** as [3,2,1], **Poles** as (poles), where poles is defined in the workspace as [7,5,3,1], and **Gain** as gain, the icon looks like this:

$$\frac{\text{gain}(s-3)(s-2)(s-1)}{(s-7)(s-5)(s-3)(s-1)}$$

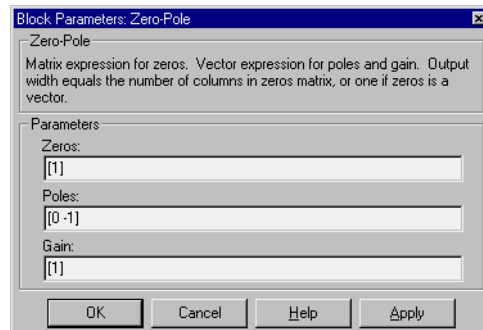
- If each is specified as a variable, the icon shows the variable name followed by “(s)” if appropriate. For example, if you specify **Zeros** as zeros, **Poles** as poles, and **Gain** as gain, the icon looks like this.

$$\frac{\text{gain}^*\text{zeros}(s)}{\text{poles}(s)}$$

## Data Type Support

A Zero-Pole block accepts real signals of type double.

## Parameters and Dialog Box



### Zeros

The matrix of zeros. The default is [1].

### Poles

The vector of poles. The default is [0 -1].

### Gain

The vector of gains. The default is [1].

# Zero-Pole

---

<b>Characteristics</b>	Direct Feedthrough	Only if the lengths of the <b>Poles</b> and <b>Zeros</b> parameters are equal
	Sample Time	Continuous
	Scalar Expansion	No
	States	Length of <b>Poles</b> vector
	Vectorized	No
	Zero Crossing	No



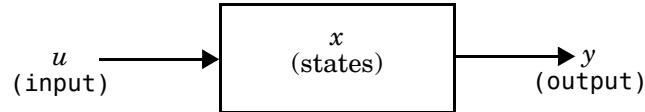
# Additional Topics

---

<b>How Simulink Works</b> . . . . .	9-2
Zero Crossings . . . . .	9-3
Algebraic Loops . . . . .	9-7
Invariant Constants . . . . .	9-11
<b>Discrete-Time Systems</b> . . . . .	9-13
Discrete Blocks . . . . .	9-13
Sample Time . . . . .	9-13
Purely Discrete Systems . . . . .	9-13
Multirate Systems . . . . .	9-14
Sample Time Colors . . . . .	9-15
Mixed Continuous and Discrete Systems . . . . .	9-17

## How Simulink Works

Each block within a Simulink model has these general characteristics: a vector of inputs,  $u$ , a vector of outputs,  $y$ , and a vector of states,  $x$ :



The state vector may consist of continuous states, discrete states, or a combination of both. The mathematical relationships between these quantities are expressed by these equations.

$$\begin{aligned}
 y &= f_o(t, x, u) && \text{output} \\
 x_{d_{k+1}} &= f_u(t, x, u) && \text{update} \\
 x'_c &= f_d(t, x, u) && \text{derivative}
 \end{aligned}$$

where  $x = \begin{bmatrix} x_c \\ x_d \end{bmatrix}$

Simulation consists of two phases: initialization and simulation. During the initialization phase:

- 1 The block parameters are passed to MATLAB for evaluation. The resulting numerical values are used as the actual block parameters.
- 2 The model hierarchy is flattened. Each subsystem that is not a conditionally executed subsystem is replaced by the blocks it contains.
- 3 Blocks are sorted into the order in which they need to be updated. The sorting algorithm constructs a list such that any block with direct feedthrough is not updated until the blocks driving its inputs are updated. It is during this step that algebraic loops are detected. For more information about algebraic loops, see “Algebraic Loops” on page 9-7.
- 4 The connections between blocks are checked to ensure that the vector length of the output of each block is the same as the input expected by the blocks it drives.

Now the simulation is ready to run. A model is simulated using numerical integration. Each of the supplied ODE solvers (simulation methods) depends on the ability of the model to provide the derivatives of its continuous states. Calculating these derivatives is a two-step process. First, each block's output is calculated in the order determined during the sorting. Then, in a second pass, each block calculates its derivatives based on the current time, its inputs, and its states. The resulting derivative vector is returned to the solver, which uses it to compute a new state vector at the next time point. Once a new state vector is calculated, the sampled data blocks and Scope blocks are updated.

## Zero Crossings

Simulink uses zero crossings to detect discontinuities in continuous signals. Zero crossings play an important role in:

- The handling of state events
- The accurate integration of discontinuous signals

## State Event Handling

A system experiences a *state event* when a change in the value of a state causes the system to undergo a distinct change. A simple example of a state event is a bouncing ball hitting the floor. When simulating such a system using a variable-step solver, the solver typically does not take steps that exactly correspond to the times that the ball makes contact with the floor. As a result, the ball is likely to overshoot the contact point, which results in the ball penetrating the floor.

Simulink uses zero crossings to ensure that time steps occur exactly (within machine precision) at the time state events occur. Because time steps occur at the exact time of contact, the simulation produces no overshoot and the transition from negative to positive velocity is extremely sharp (that is, there is no rounding of corners at the discontinuity). To see a bouncing ball demo, type `bounce` at the MATLAB prompt.

## Integration of Discontinuous Signals

Numerical integration routines are formulated on the assumption that the signals they are integrating are continuous and have continuous derivatives. If a discontinuity (state event) is encountered during an integration step, Simulink uses zero crossing detection to find the time at which the discontinuity occurs. An integration step is then taken up to the left edge of the

discontinuity. Finally, Simulink steps over the discontinuity and begins a new integration step on the next piece-wise continuous portion of the signal.

### Implementation Details

An example of a Simulink block that uses zero crossings is the Saturation block. Zero crossings detect 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*. If you need explicit notification of a zero crossing event, use the Hit Crossing block. See “Blocks with Zero Crossings” on page 9-6 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.

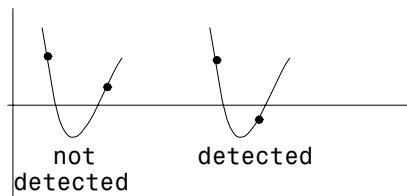
Zero crossing signals have a direction attribute, which can have these values:

- *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.

If the error tolerances are too large, it is possible for Simulink to fail to detect a zero crossing. For example, 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 will step over the crossing without detecting it.

This figure shows a signal that crosses zero. In the first instance, the integrator “steps over” the event. In the second, the solver detects the event.



If you suspect this is happening, tighten the error tolerances to ensure that the solver takes small enough steps. For more information, see “Error Tolerances” on page 4–13.

### Caveat

It is possible to create models that exhibit high frequency fluctuations about a discontinuity (chattering). Such systems typically are not physically realizable; a mass-less spring, for example. Because chattering causes repeated detection of zero crossings, the step sizes of the simulation become very small, essentially halting the simulation.

If you suspect that this behavior applies to your model, you can disable zero crossings by selecting the **Disable zero crossing detection** check box on the **Diagnostics** page of the **Simulation Parameters** dialog box. Although disabling zero crossing detection may alleviate the symptoms of this problem, you no longer benefit from the increased accuracy that zero crossing detection provides. A better solution is to try to identify the source of the underlying problem in the model.

## Blocks with Zero Crossings

**Table 9-1: Blocks with Intrinsic Zero Crossings**

<b>Block</b>	<b>Description of Zero Crossing</b>
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.
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).
Hit Crossing	One: to detect when the input crosses the threshold. These zero crossings are not affected by the <b>Disable zero crossing detection</b> check box in the <b>Simulation Parameters</b> dialog box.
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
Relay	One: if the relay is off, to detect the switch on point. If the relay is on, to detect the switch off point.
Relational Operator	One: to detect when the output changes.
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.
Step	One: to detect the step time.

**Table 9-1: Blocks with Intrinsic Zero Crossings (Continued)**

Block	Description of Zero Crossing
Subsystem	For conditionally executed subsystems: one for the enable port if present, and one for the trigger port, if present.
Switch	One: to detect when the switch condition occurs.

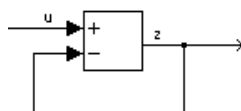
## Algebraic Loops

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:

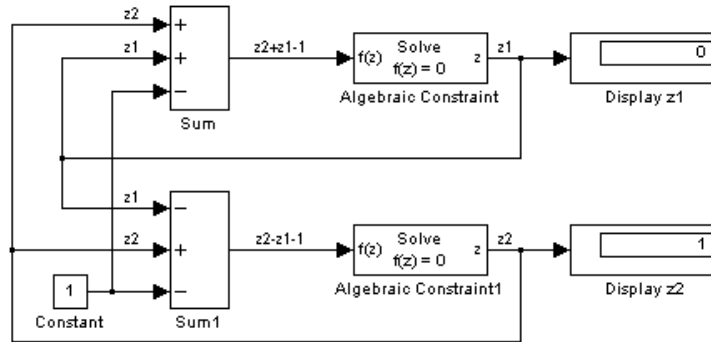
- The Elementary Math block
- The Gain block
- The Integrator block's initial condition ports
- The Product block
- The State-Space block when there is a nonzero D matrix
- The Sum block
- The Transfer Fcn block when the numerator and denominator are of the same order
- The Zero-Pole block when there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

To determine whether a block has direct feedthrough, consult the Characteristics table that describes the block, in Chapter 8.

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. (See “Non-Algebraic Direct-Feedthrough Loops” on page 9-9 for an example of an exception to this general rule.) 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 (see Algebraic Constraint on page 8-12) 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 feedback path. 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 may arise as a consequence of the physical interconnectivity of the system you are modeling, or it may arise because you are specifically trying to model a differential/algebraic system (DAE).



When a model contains an algebraic loop, Simulink calls a loop solving routine 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.

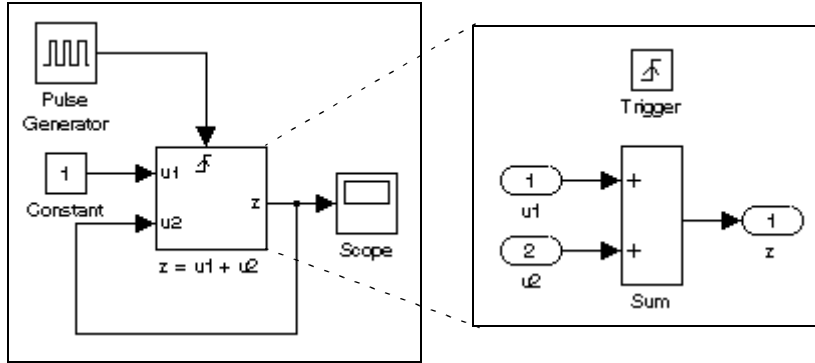
### **Non-Algebraic Direct-Feedthrough Loops**

There are exceptions to the general rule that all loops comprising direct-feedthrough blocks are algebraic. The exceptions are:

- Loops involving triggered subsystems
- A loop from the output to the reset port of an integrator

In the case of a triggered subsystem, a solver can safely assume that the subsystem's inputs are stable at the time of the trigger. This allows use of the output from a previous time step to compute the input at the current time step, thus eliminating the need for an algebraic loop solver.

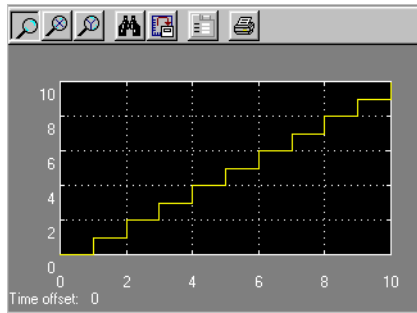
Consider, for example, the following system.



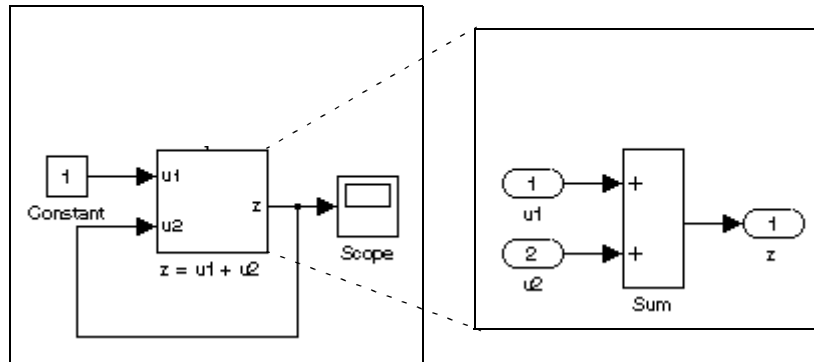
This system effectively solves the equation

$$z = 1 + u$$

where  $u$  is the value of  $z$  the last time the subsystem was triggered. The output of the system is a staircase function as illustrated by the display on the system's scope.



Now consider the effect of removing the trigger from the system shown in the previous example.



In this case, the input at the  $u2$  port of the adder subsystem is equal to the subsystem's output at the current time step for every time step. The mathematical representation of this system

$$z = z + 1$$

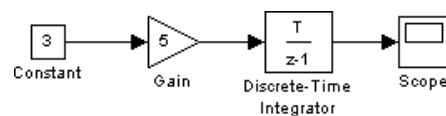
reveals that it has no mathematically valid solution.

## Invariant Constants

Blocks either have explicitly defined sample times or inherit their sample times from blocks that feed them or are fed by them.

Simulink assigns Constant blocks a sample time of infinity, also referred to as a *constant sample time*. Other blocks have constant sample time if they receive their input from a Constant block and do not inherit the sample time of another block. This means that the output of these blocks does not change during the simulation unless the parameters are explicitly modified by the model user.

For example, in this model, both the Constant and Gain blocks have constant sample time.



Because Simulink supports the ability to change block parameters during a simulation, all blocks, even blocks having constant sample time, must generate their output at the model's effective sample time.

Because of this feature, *all* blocks compute their output at each sample time hit, or, in the case of purely continuous systems, at every simulation step. For blocks having constant sample time whose parameters do not change during a simulation, evaluating these blocks during the simulation is inefficient and slows down the simulation.

You can set the `InvariantConstants` parameter to remove all blocks having constant sample times from the simulation "loop." The effect of this feature is twofold: first, parameters for these blocks cannot be changed during a simulation; and second, simulation speed is improved. The speed improvement depends on model complexity, the number of blocks with constant sample time, and the effective sampling rate of the simulation.

You can set the parameter for your model by entering this command:

```
set_param('model_name', 'InvariantConstants', 'on')
```

You can turn off the feature by issuing the command again, assigning the parameter the value of 'off'.

You can determine which blocks have constant sample time by selecting **Sample Time Colors** from the **Format** menu. Blocks having constant sample time are colored magenta.

## Discrete-Time Systems

Simulink has the ability to simulate discrete (sampled data) systems. Models can be *multirate*; that is, they can contain blocks that are sampled at different rates. Models can also be *hybrid*, containing a mixture of discrete and continuous blocks.

### Discrete Blocks

Each of the discrete blocks has a built-in sampler at its input, and a zero-order hold at its output. When the discrete blocks are mixed with continuous blocks, the output of the discrete blocks between sample times is held constant. The outputs of the discrete blocks are updated only at times that correspond to sample hits.

### Sample Time

The **Sample time** parameter sets the sample time at which a discrete block's states are updated. Normally, the sample time is set to a scalar variable; however, it is possible to specify an offset time (or skew) by specifying a two-element vector in this field.

For example, specifying the **Sample time** parameter as the vector `[Ts, offset]` sets the sample time to `Ts` and the offset value to `offset`. The discrete block is updated on integer multiples of the sample time and offset values only

$$t = n * Ts + offset$$

where `n` is an integer and `offset` can be positive or negative, but less than the sample time. The offset is useful if some discrete blocks must be updated sooner or later than others.

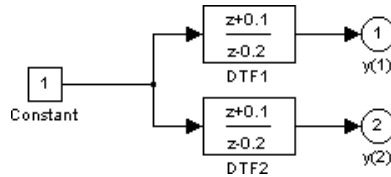
You cannot change the sample time of a block while a simulation is running. If you want to change a block's sample time, you must stop and restart the simulation for the change to take effect.

### Purely Discrete Systems

Purely discrete systems can be simulated using any of the solvers; there is no difference in the solutions. To generate output points only at the sample hits, choose one of the discrete solvers.

## Multirate Systems

Multirate systems contain blocks that are sampled at different rates. These systems can be modeled with discrete blocks or both discrete and continuous blocks. For example, consider this simple multirate discrete model.

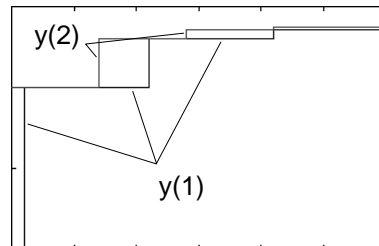


For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to  $[1 \ 0.1]$ , which gives it an offset of  $0.1$ . The DTF2 Discrete Transfer Fcn block's **Sample time** is set to  $0.7$ , with no offset.

Starting the simulation and plotting the outputs using the stairs function

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

produces this plot:



For the DTF1 block, which has an offset of  $0.1$ , there is no output until  $t = 0.1$ . Because the initial conditions of the transfer functions are zero, the output of DTF1,  $y(1)$ , is zero before this time.

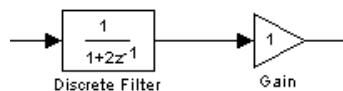
## Sample Time Colors

Simulink identifies different sample rates in a model using the sample time color feature, which shows sample rates by applying the color scheme shown in this table.

**Table 9-2: Sample Time Colors**

Color	Use
Black	Continuous blocks
Magenta	Constant blocks
Yellow	Hybrid (subsystems grouping blocks, or Mux or Demux blocks grouping signals with varying sample times)
Red	Fastest discrete sample time
Green	Second fastest discrete sample time
Blue	Third fastest discrete sample time
Light Blue	Fourth fastest discrete sample time
Dark Green	Fifth fastest discrete sample time
Cyan	Triggered sample time
Gray	Fixed in minor step

To understand how this feature works, it is important to be familiar with Simulink's Sample Time Propagation Engine (STPE). The figure below illustrates a Discrete Filter block with a sample time of  $T_s$  driving a Gain block. Because the Gain block's output is simply 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 that of the filter's sample rate. This is the fundamental mechanism behind the STPE.



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

Simulink 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**.

When using sample time colors, the color assigned to each block depends on its sample time with respect to other sample times in the model.

Simulink sets sample times for individual blocks according to these rules:

- Continuous blocks (e.g., Integrator, Derivative, Transfer Fcn, etc.) are, by definition, continuous.
- Constant blocks (for example, Constant) are, by definition, constant.
- Discrete blocks (e.g., Zero-Order Hold, Unit Delay, Discrete Transfer Fcn, etc.) have sample times that are explicitly specified by the user on the block dialog boxes.
- All other blocks have implicitly defined sample times that are based on the sample times of their inputs. For instance, a Gain block that follows an Integrator is treated as a continuous block, whereas a Gain block that follows a Zero-Order Hold is treated as a discrete block having the same sample time as the Zero-Order Hold block.

For blocks whose inputs have different sample times, if all sample times are integer multiples of the fastest sample time, the block is assigned the sample time of the fastest input. If a variable-step solver is being used, the block is assigned the continuous sample time. If a fixed-step solver is being used and the greatest common divisor of the sample times (the fundamental sample time) can be computed, it is used. Otherwise continuous is used.

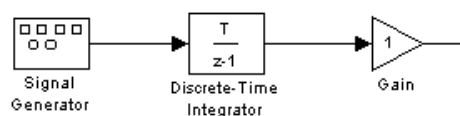
It is important to note that Mux and Demux blocks are simply grouping operators – signals passing through them retain their timing information. For this reason, the lines emanating from a Demux block may have different colors if they are driven by sources having different sample times. In this case, 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



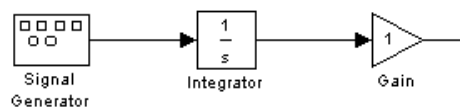
them. If all of the blocks within a subsystem run at a single rate, then the Subsystem block is colored according to that rate.

Under some circumstances, Simulink also backpropagates sample times to source blocks if it can do so without affecting the output of a simulation. For instance, in the model below, Simulink recognizes that the Signal Generator block is driving a Discrete-Time Integrator block so it assigns the Signal Generator block and the Gain block the same sample time as the Discrete-Time Integrator block.



You can verify this by enabling **Sample Time Colors** and noting that all blocks are colored red. Because the Discrete-Time Integrator block only looks at its input at its sample times, this change does not affect the outcome of the simulation but does result in a performance improvement.

Replacing the Discrete-Time Integrator block with a continuous Integrator block, as shown below, and recoloring the model by choosing **Update Diagram** from the **Edit** menu cause the Signal Generator and Gain blocks to change to continuous blocks, as indicated by their being colored black.



## Mixed Continuous and Discrete Systems

Mixed continuous and discrete systems are composed of both sampled and continuous blocks. Such systems can be simulated using any of the integration methods, although certain methods are more efficient and accurate than others. For most mixed continuous and discrete systems, the Runge-Kutta variable step methods, ode23 and ode45, are superior to the other methods in terms of efficiency and accuracy. Due to discontinuities associated with the sample and hold of the discrete blocks, the ode15s and ode113 methods are not recommended for mixed continuous and discrete systems.



# Model Construction Commands

---

<b>Introduction</b> . . . . .	10-2
How to Specify Parameters for the Commands . . . . .	10-3
How to Specify a Path for a Simulink Object . . . . .	10-3
<b>add_block</b> . . . . .	10-4
<b>add_line</b> . . . . .	10-5
<b>bdclose</b> . . . . .	10-6
<b>bdroot</b> . . . . .	10-7
<b>close_system</b> . . . . .	10-8
<b>delete_block</b> . . . . .	10-10
<b>delete_line</b> . . . . .	10-11
<b>find_system</b> . . . . .	10-12
<b>gcb</b> . . . . .	10-14
<b>gcbh</b> . . . . .	10-15
<b>gcs</b> . . . . .	10-16
<b>get_param</b> . . . . .	10-17
<b>new_system</b> . . . . .	10-19
<b>open_system</b> . . . . .	10-20
<b>replace_block</b> . . . . .	10-21
<b>save_system</b> . . . . .	10-23
<b>set_param</b> . . . . .	10-24
<b>simulink</b> . . . . .	10-26

## Introduction

This table indicates the tasks performed by the commands described in this chapter. The reference section of this chapter lists the commands in alphabetical order.

<b>Task</b>	<b>Command</b>
Create a new Simulink system.	<code>new_system</code>
Open an existing system.	<code>open_system</code>
Close a system window.	<code>close_system</code> , <code>bdclose</code>
Save a system.	<code>save_system</code>
Find a system, block, line, or annotation.	<code>find_system</code>
Add a new block to a system.	<code>add_block</code>
Delete a block from a system.	<code>delete_block</code>
Replace a block in a system.	<code>replace_block</code>
Add a line to a system.	<code>add_line</code>
Delete a line from a system.	<code>delete_line</code>
Get a parameter value.	<code>get_param</code>
Set parameter values.	<code>set_param</code>
Get the pathname of the current block.	<code>gcb</code>
Get the pathname of the current system.	<code>gcs</code>
Get the handle of the current block.	<code>gcbh</code>
Get the name of the root-level system.	<code>bdroot</code>
Open the Simulink block library.	<code>simulink</code>

## How to Specify Parameters for the Commands

The commands described in this chapter require that you specify arguments that describe a system, block, or block parameter. Appendix A provides comprehensive tables of model and block parameters.

## How to Specify a Path for a Simulink Object

Many of the commands described in this chapter require that you identify a Simulink system or block. Identify systems and blocks by specifying their paths:

- To identify a system, specify its name, which is the name of the file that contains the system description, without the mdl extension.  
`system`
- To identify a subsystem, specify the system and the hierarchy of subsystems in which the subsystem resides.  
`system/subsystem1/.../subsystem`
- To identify a block, specify the path of the system that contains the block and specify the block name.  
`system/subsystem1/.../subsystem/block`

If the block name includes a newline or carriage return, specify the block name as a string vector and use `sprintf('\n')` as the newline character. For example, these lines assign the newline character to `cr`, then get the value for the Signal Generator block's Amplitude parameter.

```
cr = sprintf('\n');
get_param(['untitled/Signal',cr,'Generator'],'Amplitude')
ans =
    1
```

If the block name includes a slash character (`/`), you repeat the slash when you specify the block name. For example, to get the value of the Location parameter for the block named Signal/Noise in the `mymodel` system.

```
get_param('mymodel/Signal//Noise','Location')
```

# add\_block

---

**Purpose** Add a block to a Simulink system.

**Syntax**  
`add_block('src', 'dest')`  
`add_block('src', 'dest', 'parameter1', value1, ...)`

**Description** `add_block('src', 'dest')` copies the block with the full pathname 'src' to a new block with the full path name 'dest'. The block parameters of the new block are identical to those of the original. The name 'built-in' can be used as a source system name for all Simulink built-in blocks (blocks available in Simulink block libraries that are not masked blocks).

`add_block('src', 'dest_obj', 'parameter1', value1, ...)` creates a copy as above, in which the named parameters have the specified values. Any additional arguments must occur in parameter-value pairs.

**Examples** This command copies the Scope block from the Sinks subsystem of the simulink system to a block named Scope1 in the timing subsystem of the engine system.

```
add_block('simulink/Sinks/Scope', 'engine/timing/Scope1')
```

This command creates a new subsystem named controller in the F14 system.

```
add_block('built-in/SubSystem', 'F14/controller')
```

This command copies the built-in Gain block to a block named Volume in the mymodel system and assigns the Gain parameter a value of 4.

```
add_block('built-in/Gain', 'mymodel/Volume', 'Gain', '4')
```

**See Also** `delete_block`, `set_param`

<b>Purpose</b>	Add a line to a Simulink system.
<b>Syntax</b>	<pre>h = add_line('sys', 'oport', 'iport') h = add_line('sys', points)</pre>
<b>Description</b>	<p>The <code>add_line</code> command adds a line to the specified system and returns a handle to the new line. The line can be defined in two ways:</p> <ul style="list-style-type: none"><li>• By naming the block ports that are to be connected by the line</li><li>• By specifying the location of the points that define the line segments</li></ul> <p><code>add_line('sys', 'oport', 'iport')</code> adds a straight line to a system from the specified block output port <code>'oport'</code> to the specified block input port <code>'iport'</code>. <code>'oport'</code> and <code>'iport'</code> are strings consisting of a block name and a port identifier in the form <code>'block/port'</code>. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as <code>'Gain/1'</code> or <code>'Sum/2'</code>. Enable, Trigger, and State ports are identified by name, such as <code>'subsystem_name/Enable'</code>, <code>'subsystem_name/Trigger'</code>, or <code>'Integrator/State'</code>.</p> <p><code>add_line(system, points)</code> adds a segmented line to a system. Each row of the <code>points</code> array specifies the <math>x</math> and <math>y</math> coordinates of a point on a line segment. The origin is the top left corner of the window. The signal flows from the point defined in the first row to the point defined in the last row. If the start of the new line is close to the output of an existing block or line, a connection is made. Likewise, if the end of the line is close to an existing input, a connection is made.</p>
<b>Examples</b>	<p>This command adds a line to the <code>mymodel</code> system connecting the output of the Sine Wave block to the first input of the Mux block.</p> <pre>add_line('mymodel', 'Sine Wave/1', 'Mux/1')</pre> <p>This command adds a line to the <code>mymodel</code> system extending from (20,55) to (40,10) to (60,60).</p> <pre>add_line('mymodel',[20 55; 40 10; 60 60])</pre>
<b>See Also</b>	<code>delete_line</code>

# bdclose

---

**Purpose** Close any or all Simulink system windows unconditionally.

**Syntax**

```
bdclose
bdclose('sys')
bdclose('all')
```

**Description** bdclose with no arguments closes the current system window unconditionally and without confirmation. Any changes made to the system since it was last saved are lost.

bdclose('sys') closes the specified system window.

bdclose('all') closes all system windows.

**Examples** This command closes the vdp system.

```
bdclose('vdp')
```

**See Also** close\_system, new\_system, open\_system, save\_system



**Purpose** Return the name of the top-level Simulink system.

**Syntax** `bdroot`  
`bdroot('obj')`

**Description** `bdroot` with no arguments returns the top-level system name.  
`bdroot('obj')` where 'obj' is a system or block pathname, returns the name of the top-level system containing the specified object name.

**Examples** This command returns the name of the top-level system that contains the current block.

```
bdroot(gcb)
```

**See Also** `find_system`, `gcb`

# close\_system

---

**Purpose** Close a Simulink system window or a block dialog box.

**Syntax**

```
close_system
close_system('sys')
close_system('sys', saveflag)
close_system('sys', 'newname')
close_system('blk')
```

**Description** `close_system` with no arguments closes the current system or subsystem window. If the current system is the top-level system and it has been modified, then `close_system` asks if the changed system should be saved to a file before removing the system from memory. The current system is defined in the description of the `gcs` command (see “`gcs`” on page 10-16).

`close_system('sys')` closes the specified system or subsystem window.

`close_system('sys', saveflag)` closes the specified top-level system window and removes it from memory:

- If `saveflag` is 0, the system is not saved.
- If `saveflag` is 1, the system is saved with its current name.

`close_system('sys', 'newname')` saves the specified top-level system to a file with the specified new name, then closes the system.

`close_system('blk')` where `'blk'` is a full block pathname, closes the dialog box associated with the specified block or calls the block's `CloseFcn` callback parameter if one is defined. Any additional arguments are ignored.

**Examples** This command closes the current system.

```
close_system
```

This command closes the vdp system.

```
close_system('vdp')
```

This command saves the engine system with its current name, then closes it.

```
close_system('engine', 1)
```

This command closes the mymdl12 system with the name testsys, then closes it.

```
close_system('mymdl12', 'testsys')
```

This command closes the dialog box of the Unit Delay block in the Combustion subsystem of the engine system.

```
close_system('engine/Combustion/Unit Delay')
```

### See Also

bdclose, gcs, new\_system, open\_system, save\_system

# delete\_block

---

**Purpose** Delete a block from a Simulink system.

**Syntax** `delete_block('blk')`

**Description** `delete_block('blk')` where 'blk' is a full block pathname, deletes the specified block from a system.

**Example** This command removes the Out1 block from the vdp system:

```
delete_block('vdp/Out1')
```

**See Also** `add_block`

**Purpose** Delete a line from a Simulink system.

**Syntax** `delete_line('sys', 'oport', 'iport')`

**Description** `delete_line('sys', 'oport', 'iport')` deletes the line extending from the specified block output port 'oport' to the specified block input port 'iport'. 'oport' and 'iport' are strings consisting of a block name and a port identifier in the form 'block/port'. Most block ports are identified by numbering the ports from top to bottom or from left to right, such as 'Gain/1' or 'Sum/2'. Enable, Trigger, and State ports are identified by name, such as 'subsystem\_name/Enable', 'subsystem\_name/Trigger', or 'Integrator/State'.

`delete_line('sys', [x y])` deletes one of the lines in the system that contains the specified point (x,y), if any such line exists.

**Example** This command removes the line from the mymodel system connecting the Sum block to the second input of the Mux block.

```
delete_line('mymodel', 'Sum/1', 'Mux/2')
```

**See Also** `add_line`

# find\_system

---

**Purpose** Find systems, blocks, lines, and annotations.

**Syntax** `find_system(sys, 'constraint', cv, 'p1', v1, 'p2', v2, ...)`

**Description** `find_system(sys, constraint, cv, 'p1', v1, 'p2', v2, ...)` searches the system(s) or subsystems specified by `sys`, using the constraint specified by `constraint`, and returns handles or paths to the objects having the specified parameter values `v1`, `v2`, etc. `sys` can be a pathname (or cell array of pathnames), a handle (or vector of handles), or omitted. If `sys` is a pathname or cell array of pathnames, `find_system` returns a cell array of pathnames of the objects it finds. If `sys` is a handle or a vector of handles, `find_system` returns a vector of handles to the objects that it finds. If `sys` is omitted, `find_system` searches all open systems.

Case is ignored for parameter names. Value strings are case sensitive. Any parameters that correspond to dialog box entries have string values. See Appendix A for a list of model and block parameters.

You can specify any of the following search constraints.

**Table 10-1: Search Constraints**

Name	Value Type	Description
'SearchDepth'	scalar	Restricts the search depth to the specified level (0 for open systems only, 1 for blocks and subsystems of the top-level system, 2 for the top-level system and its children, etc.) Default is all levels.
'LookUnderMasks'	'on'   'off'	If 'on', search extends into masked systems. Default is 'off'.
'FollowLinks'	'on'   'off'	If 'on', search follows links into library blocks. Default is 'off'.
'FindAll'	'on'   'off'	If 'on', search extends to lines and annotations within systems. Default is 'off'.

If 'constraint' is omitted, find\_system uses the default constraint values.

## Examples

This command returns a cell array containing the names of all open systems and blocks.

```
find_system
```

This command returns the names of all open block diagrams.

```
open_bd = find_system('Type', 'block_diagram')
```

This command returns the names of all Goto blocks that are children of the Unlocked subsystem in the clutch system.

```
find_system('clutch/Unlocked', 'SearchDepth', 1, 'BlockType', 'Goto')
```

These commands return the names of all Gain blocks in the vdp system having a Gain parameter value of 1.

```
gb = find_system('vdp', 'BlockType', 'Gain')
find_system.gb, 'Gain', '1')
```

The above commands are equivalent to this command.

```
find_system('vdp', 'BlockType', 'Gain', 'Gain', '1')
```

These commands obtain the handles of all lines and annotations in the vdp system.

```
sys = get_param('vdp', 'Handle');
l = find_system(sys, 'FindAll', 'on', 'type', 'line');
a = find_system(sys, 'FindAll', 'on', 'type', 'annotation');
```

## See Also

get\_param, set\_param

# gcb

---

**Purpose** Get the pathname of the current block.

**Syntax** `gcb`  
`gcb('sys')`

**Description** `gcb` returns the full block path name of the current block in the current system.

`gcb('sys')` returns the full block path name of the current block in the specified system.

The current block is one of these:

- During editing, the current block is the block most recently clicked on.
- During simulation of a system that contains S-Function blocks, the current block is the S-Function block currently executing its corresponding MATLAB function.
- During callbacks, the current block is the block whose callback routine is being executed.
- During evaluation of the `MaskInitialization` string, the current block is the block whose mask is being evaluated.

**Examples** This command returns the path of the most recently selected block.

```
gcb
ans =
    clutch/Locked/Inertia
```

This command gets the value of the Gain parameter of the current block.

```
get_param(gcb,'Gain')
ans =
    1/(Iv+Ie)
```

**See Also** `gcbh`, `gcs`



**Purpose** Get the handle of the current block.

**Syntax** gcbh

**Description** gcbh returns the handle of the current block in the current system.

You can use this command to identify or address blocks that have no parent system. The command should be most useful to blockset authors.

**Examples** This command returns the handle of the most recently selected block.

```
gcbh
```

```
ans =
```

```
281.0001
```

**See Also** gcb

# gcs

---

**Purpose** Get the pathname of the current system.

**Syntax** gcs

**Description** gcs returns the full path name of the current system.

The current system is:

- During editing, the current system is the system or subsystem most recently clicked in.
- During simulation of a system that contains S-Function blocks, the current system is the system or subsystem containing the S-Function block that is currently being evaluated.
- During callbacks, the current system is the system containing any block whose callback routine is being executed.
- During evaluation of the MaskInitialization string, the current system is the system containing the block whose mask is being evaluated.

**Examples** This example returns the path of the system that contains the most recently selected block.

```
gcs
ans =
    clutch/Locked
```

**See Also** gcb

**Purpose** Get system and block parameter values.

**Syntax**

```
get_param('obj', 'parameter')
get_param( { objects }, 'parameter')
get_param(handle, 'parameter')
get_param('obj', 'ObjectParameters')
get_param('obj', 'DialogParameters')
```

**Description**

`get_param('obj', 'parameter')`, where 'obj' is a system or block path name, returns the value of the specified parameter. Case is ignored for parameter names.

`get_param( { objects }, 'parameter')` accepts a cell array of full path specifiers, enabling you to get the values of a parameter common to all objects specified in the cell array.

`get_param(handle, 'parameter')` returns the specified parameter of the object whose handle is `handle`.

`get_param('obj', 'ObjectParameters')` returns a structure that describes `obj`'s parameters. Each field of the returned structure corresponds to a particular parameter and has the parameter's name. For example, the `Name` field corresponds to the object's `Name` parameter. Each parameter field itself contains three fields, `Name`, `Type`, and `Attributes`, that specify the parameter's name (for example, "Gain"), data type (for example, `string`), and attributes (for example, `read-only`), respectively.

`get_param('obj', 'DialogParameters')` returns a cell array containing the names of the dialog parameters of the specified block.

Appendix A contains lists of model and block parameters.

**Examples**

This command returns the value of the `Gain` parameter for the `Inertia` block in the `Requisite Friction` subsystem of the `clutch` system.

```
get_param('clutch/Requisite Friction/Inertia', 'Gain')
ans =
    1/(Iv+Ie)
```

## get\_param

---

These commands display the block types of all blocks in the mx+b system (the current system), described in “A Sample Masked Subsystem” on page 6–3:

```
blks = find_system(gcs, 'Type', 'block');
listblks = get_param(blks, 'BlockType')

listblks =

    'SubSystem'
    'Inport'
    'Constant'
    'Gain'
    'Sum'
    'Outport'
```

This command returns the name of the currently selected block.

```
get_param(gcb, 'Name')
```

The following commands gets the attributes of the currently selected block’s Name parameter.

```
p = get_param(gcb, 'ObjectParameters');
a = p.Name.Attributes

ans =

    'read-write'    'always-save'
```

The following command gets the dialog parameters of a Sine Wave block.

```
p = get_param('untitled/Sine Wave', 'DialogParameters')
p =

    'Amplitude'
    'Frequency'
    'Phase'
    'SampleTime'
```

### See Also

`find_system`, `set_param`

**Purpose** Create an empty Simulink system.

**Syntax** `new_system('sys')`

**Description** `new_system('sys')` creates a new empty system with the specified name. If 'sys' specifies a path, the new system will be a subsystem of the system specified in the path. `new_system` does not open the system window.

For a list of the default parameter values for the new system, see Appendix A.

**Example** This command creates a new system named 'mysys'.

```
new_system('mysys')
```

This command creates a new subsystem named 'mysys' in the vdp system.

```
new_system('vdp/mysys')
```

**See Also** `close_system`, `open_system`, `save_system`

# open\_system

---

**Purpose** Open a Simulink system window or a block dialog box.

**Syntax**

```
open_system('sys')
open_system('blk')
open_system('blk', 'force')
```

**Description**

`open_system('sys')` opens the specified system or subsystem window.

`open_system('blk')`, where 'blk' is a full block pathname, opens the dialog box associated with the specified block. If the block's `OpenFcn` callback parameter is defined, the routine is evaluated.

`open_system('blk', 'force')`, where 'blk' is a full pathname or a masked system, looks under the mask of the specified system. This command is equivalent to using the **Look Under Mask** menu item.

**Example** This command opens the controller system in its default screen location.

```
open_system('controller')
```

This command opens the block dialog box for the Gain block in the controller system.

```
open_system('controller/Gain')
```

**See Also** `close_system`, `new_system`, `save_system`

**Purpose** Replace blocks in a Simulink model.

**Syntax** `replace_block('sys', 'blk1', 'blk2', 'noprompt')`  
`replace_block('sys', 'Parameter', 'value', 'blk', ...)`

**Description** `replace_block('sys', 'blk1', 'blk2')` replaces all blocks in 'sys' having the block or mask type 'blk1' with 'blk2'. If 'blk2' is a Simulink built-in block, only the block name is necessary. If 'blk' is in another system, its full block pathname is required. If 'noprompt' is omitted, Simulink displays a dialog box that asks you to select matching blocks before making the replacement. Specifying the 'noprompt' argument suppresses the dialog box from being displayed. If a return variable is specified, the paths of the replaced blocks are stored in that variable.

`replace_block('sys', 'Parameter', 'value', ..., 'blk')` replaces all blocks in 'sys' having the specified values for the specified parameters with 'blk'. You can specify any number of parameter/value pairs.

---

**Note** Because it may be difficult to undo the changes this command makes, it is a good idea to save your system first.

---

**Example** This command replaces all Gain blocks in the f14 system with Integrator blocks and stores the paths of the replaced blocks in RepNames. Simulink lists the matching blocks in a dialog box before making the replacement.

```
RepNames = replace_block('f14', 'Gain', 'Integrator')
```

This command replaces all blocks in the Unlocked subsystem in the clutch system having a Gain of 'bv' with the Integrator block. Simulink displays a dialog box listing the matching blocks before making the replacement.

```
replace_block('clutch/Unlocked', 'Gain', 'bv', 'Integrator')
```

This command replaces the Gain blocks in the f14 system with Integrator blocks but does not display the dialog box.

```
replace_block('f14', 'Gain', 'Integrator', 'noprompt')
```

# replace\_block

---

## See Also

find\_system, set\_param



**Purpose** Save a Simulink system.

**Syntax**

```
save_system
save_system('sys')
save_system('sys', 'newname')
```

**Description**

`save_system` saves the current top-level system to a file with its current name.

`save_system('sys')` saves the specified top-level system to a file with its current name. The system must be open.

`save_system('sys', 'newname')` saves the specified top-level system to a file with the specified new name. The system must be open.

**Example** This command saves the current system.

```
save_system
```

This command saves the vdp system:

```
save_system('vdp')
```

This command saves the vdp system to a file with the name 'myvdp'.

```
save_system('vdp', 'myvdp')
```

**See Also** `close_system`, `new_system`, `open_system`

# set\_param

---

**Purpose** Set Simulink system and block parameters.

**Syntax** `set_param('obj', 'parameter1', value1, 'parameter2', value2, ...)`

**Description** `set_param('obj', 'parameter1', value1, 'parameter2', value2, ...)`, where 'obj' is a system or block path, sets the specified parameters to the specified values. Case is ignored for parameter names. Value strings are case sensitive. Any parameters that correspond to dialog box entries have string values. Model and block parameters are listed in Appendix A.

You can change block parameter values in the workspace during a simulation and update the block diagram with these changes. To do this, make the changes in the command window, then make the model window the active window, then choose **Update Diagram** from the **Edit** menu.

---

**Note** Most block parameter values must be specified as strings. Two exceptions are the Position and UserData parameters, common to all blocks.

---

**Examples** This command sets the Solver and StopTime parameters of the vdp system.

```
set_param('vdp', 'Solver', 'ode15s', 'StopTime', '3000')
```

This command sets the Gain parameter of block Mu in the vdp system to 1000 (stiff).

```
set_param('vdp/Mu', 'Gain', '1000')
```

This command sets the position of the Fcn block in the vdp system.

```
set_param('vdp/Fcn', 'Position', [50 100 110 120])
```

This command sets the Zeros and Poles parameters for the Zero-Pole block in the mymodel system.

```
set_param('mymodel/Zero-Pole', 'Zeros', '[2 4]', 'Poles', '[1 2 3]')
```

This command sets the Gain parameter for a block in a masked subsystem. The variable k is associated with the Gain parameter.

```
set_param('mymodel/Subsystem', 'k', '10')
```

This command sets the OpenFcn callback parameter of the block named Compute in system mymodel. The function 'my\_open\_fcn' executes when the user double-clicks on the Compute block. For more information, see “Using Callback Routines” on page 3–53.

```
set_param('mymodel/Compute', 'OpenFcn', 'my_open_fcn')
```

### See Also

get\_param, find\_system

# simulink

---

**Purpose** Open the Simulink block library.

**Syntax** `simulink`

**Description** The `simulink` command opens the Simulink block library window and creates and displays a new (empty) model window, except in these cases:

- If a model window is open, Simulink does not create a new model window.
- If the Simulink block library window is already open, issuing this command makes the Simulink window the active window.

# Simulink Debugger

---

<b>Introduction</b> . . . . .	11-2
<b>Using the Debugger</b> . . . . .	11-3
<b>Running a Simulation Incrementally</b> . . . . .	11-6
<b>Setting Breakpoints</b> . . . . .	11-9
<b>Displaying Information About the Simulation</b> . . . . .	11-13
<b>Displaying Information About the Model</b> . . . . .	11-17
<b>Debugger Command Reference</b> . . . . .	11-22

## Introduction

The Simulink debugger is a tool for locating and diagnosing bugs in a Simulink model. It enables you to pinpoint problems by running simulations step-by-step and displaying intermediate block states and input and outputs. The following sections describe how to use the debugger to diagnose problems in Simulink models.

# Using the Debugger

## Starting the Debugger

Use the `sldebug` command or the `debug` option of the `sim` command to start a model under debugger control. (See `sim` on page 4-30 for information on specifying `sim` options.)

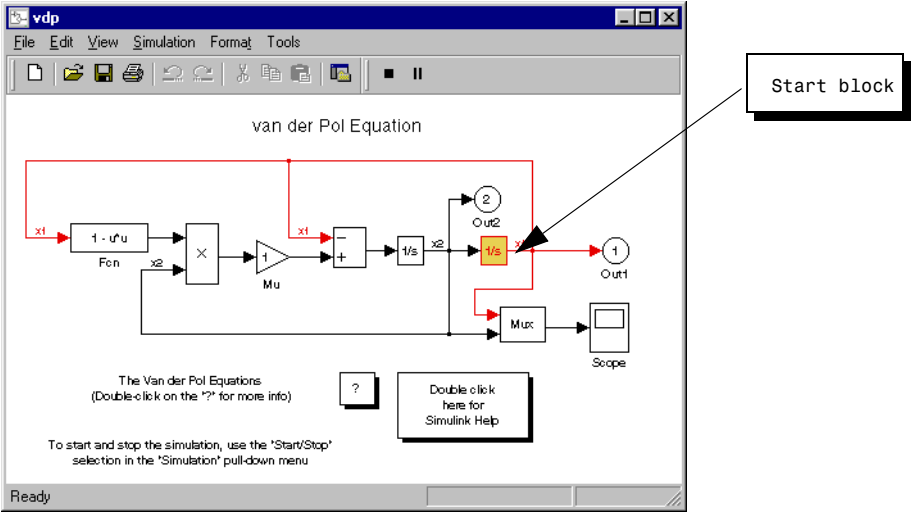
For example, either the command

```
sim('vdp',[0,10],simset('debug','on'))
```

or the command

```
sldebug 'vdp'
```

loads the Simulink demo model, `vdp`, into memory and pauses at the first block in the first time step. The debugger highlights the model's initial block and associated output signal lines in the model diagram. The next figure shows the `vdp` block diagram as it appears on debug mode start-up.



The debugger also prints the simulation start time and a debug command prompt in the MATLAB command window. The command prompt displays the block index (see “About Block Indexes” on page 11-4) and name of the first block

to be executed. For example, the command in the preceding example results in the following output in the MATLAB command window.

```
[Tm=0                               ] **Start** of system 'vdp' outputs  
(sldebug @0:0 'vdp/Integrator1'): step
```

At this point, you can get help, run the simulation step-by-step, examine data, or perform other debugging tasks by entering debugger and other MATLAB commands at the debug prompt. The following sections explain how to use the debugger commands.

## Getting Help

You can get a brief description of the debugger commands by typing help at the debug prompt. For a detailed description of each command, refer to the debugger command reference at the end of this chapter. The following sections show how to use these commands to debug a model.

## Entering Commands

The debugger accepts abbreviations for debugger commands. You can also repeat some commands by entering an empty command (i.e., by pressing the **Return** key) at the MATLAB command line. See “Debugger Command Reference” on page 11-22 for a list of command abbreviations and repeatable commands.

## About Block Indexes

Many Simulink debugger commands and messages use block indexes to refer to blocks. A block index has the form `s:b` where `s` is an integer identifying a system in the model being debugged and `b` is an integer identifying a block within that system. For example, the block index `0:1` refers to block 1 in the model's 0 system. The `slist` command shows the block index for each block in the model being debugged (see `slist` on page 11-40).

## Accessing the MATLAB Workspace

You can type 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`. Then, the following command

```
(sldebug ...) plot(tout, yout)
```



creates a plot. Suppose you would like to access a variable whose name is the same as the complete or incomplete name of an `slddebug` command, for example, `s`, which is a partial completion for the `step` command. Typing an `s` at the `slddebug` prompt steps the model. However,

```
(slddebug...) eval('s')
```

displays the value of the variable `s`.

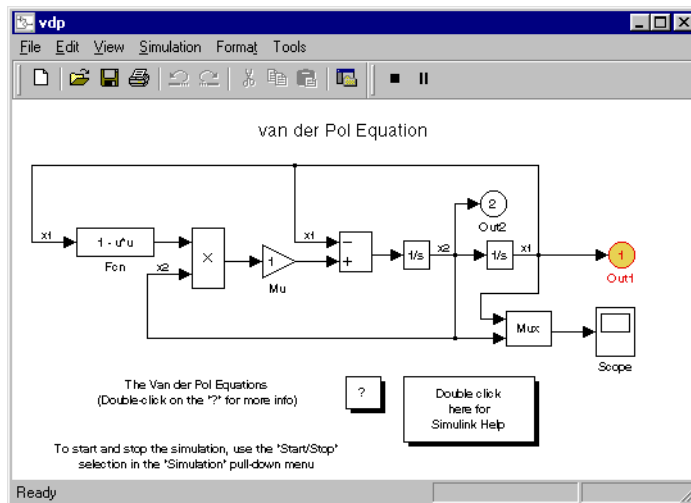
## Running a Simulation Incrementally

The Simulink debugger lets you run a simulation step by step. You can step from block to block, time point to time point, or from breakpoint to breakpoint (see “Setting Breakpoints” on page 11-9). You select the amount to advance the simulation by entering the appropriate debugger command.

Command	Advances a Simulation...
step	One block
next	One time step
continue	To next breakpoint
run	To end of simulation, ignoring breakpoints

### Stepping by Blocks

To advance a simulation one block, enter step at the debugger prompt. The debugger executes the current block, stops, and highlights the next block in the model’s block execution order (see “Displaying a Model’s Block Execution Order” on page 11-17). For example, the following figure shows the vdp block diagram after execution of the model’s first block.



If the next block to be executed occurs in a subsystem block, the debugger opens the subsystem's block diagram and highlights the next block.

After executing a block, the debugger prints the block's inputs (U) and outputs (Y) and redisplay the debug command prompt in the MATLAB command window. The debugger prompt shows the next block to be evaluated.

```
(sdebug @0:0 'vdp/Integrator1'): step
U1 = [0]
Y1 = [2]
(sdebug @0:1 'vdp/Out1'):
```

### Crossing a Time Step Boundary

When you step through the last block in the model's sorted list, the debugger advances the simulation to the next time step and halts the simulation at the beginning of the first block to be executed in the next time step. To signal that you have crossed a time step boundary, the debugger prints the current time in the MATLAB command window. For example, stepping through the last block of the first time step of the vdp model results in the following output in the MATLAB command window.

```
(sdebug @0:8 'vdp/Sum'): step
U1 = [2]
U2 = [0]
Y1 = [-2]
[Tm=0.0001004754572603832 ] **Start** of system 'vdp' outputs
```

### Stepping by Minor Time Steps

You can step by blocks within minor time steps, as well as within major steps. To step by blocks within minor time steps, enter `minor` at the debugger command prompt.

### Stepping by Time Steps

The next command executes the remaining blocks in the current time step. In effect, it enables you to advance the simulation to the next time step with a single command. This is convenient when you know that nothing of interest happens in the remainder of the current time step. After advancing the simulation to the next time step, the debugger breaks at the first block in the model's sorted list. For example, entering `next` after starting the vdp model in

debug mode causes the following message to appear in the MATLAB command window.

```
[Tm=0.0001004754572603832 ] **Start** of system 'vdp' outputs
```

## Stepping by Breakpoints

The `continue` command advances the simulation from the current breakpoint to the next breakpoint (see “Setting Breakpoints” on page 11-9) or to the end of the simulation, whichever comes first.

## Running a Simulation Nonstop

The `run` command lets you run a program from the current point in the simulation to the end, skipping any intervening breakpoints. At the end of the simulation, the debugger returns you to the MATLAB command line. To continue debugging a model, you must restart the debugger (see “Starting the Debugger” on page 11-3).

## Setting Breakpoints

The Simulink debugger allows you to define stopping points in a simulation called breakpoints. You can then run a simulation from breakpoint to breakpoint, using the debugger's `continue` command. The debugger lets you define two types of breakpoints: unconditional and conditional. An unconditional breakpoint occurs whenever a simulation reaches a block or time step that you specified previously. A conditional breakpoint occurs when a condition that you specified in advance arises in the simulation.

Breakpoints come in handy 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.

You set a particular kind of breakpoint by entering the appropriate breakpoint command.

<b>Command</b>	<b>Causes Simulation to Stop...</b>
<code>break &lt;gcb   s:b&gt;</code>	At the beginning of a block
<code>bafter &lt;gcb   s:b&gt;</code>	At the end of a block
<code>tbreak [t]</code>	At a simulation time step
<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.

### Breaking at Blocks

The debugger lets you specify a breakpoint at the beginning or end of a block.

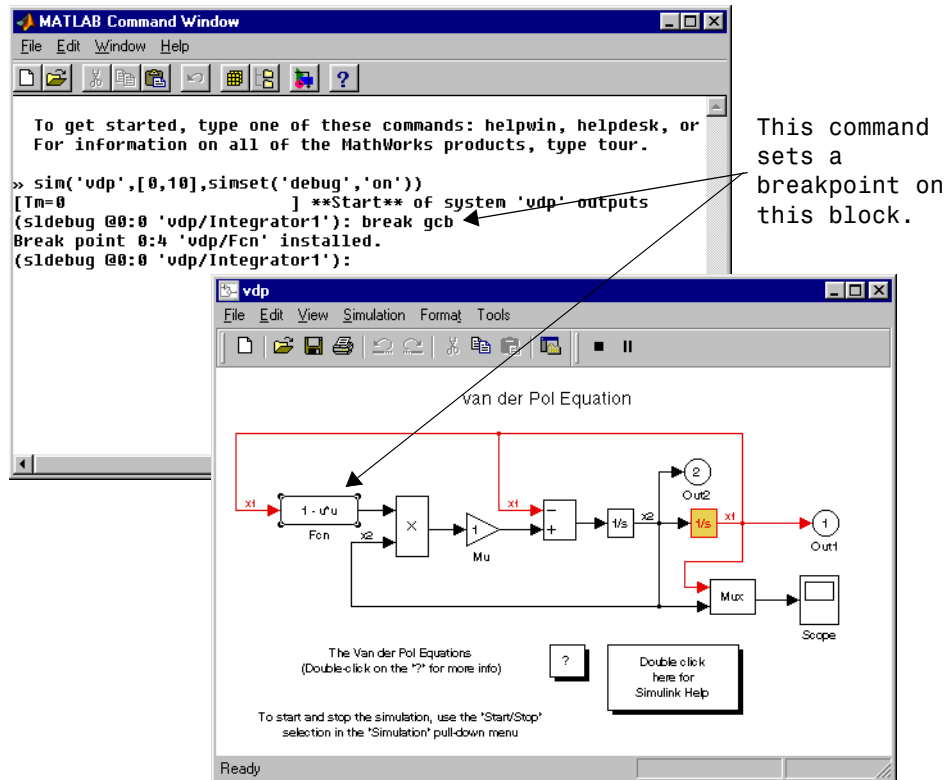
### Breaking at a Block's Beginning

The break command lets you set a breakpoint at the beginning of a block. Setting a breakpoint at the beginning of a block causes the debugger to stop the simulation when it reaches the block on each time step.

You can specify the block on which to set the breakpoint via a block index or graphically. To specify the block graphically, select the block in the model's block diagram and enter

```
break gcb
```

as shown in the following figure.



To specify the block via its index, enter

```
break s:b
```

where `s:b` is the block's index (see "About Block Indexes" on page 11-4).

---

**Note** You cannot set a breakpoint on a virtual block. A virtual block is a block whose function is purely graphical: it indicates a grouping or relationship among a model's computational blocks. The debugger warns you if you attempt to set a breakpoint on a virtual block. You can obtain a listing of a model's nonvirtual blocks, using the `slist` command (see "Displaying a Model's Nonvirtual Blocks" on page 11-18).

---

### Breaking at a Block's End

The `bafter` command sets a breakpoint at the end of a nonvirtual block. As with `break`, you can specify the block graphically or via its block index.

### Clearing Breakpoints from Blocks

The `clear` command clears a breakpoint from the beginning or end of a block. You can specify the block by entering its block index or by selecting the block in the model diagram and entering `gcb` as the argument of the `clear` command.

### Breaking at Time Steps

You can use the `tbreak` command to set a breakpoint at a particular time step. The `tbreak` command takes a time value as its only argument. It causes the debugger to stop the simulation at the beginning of the first time step that follows the specified time. For example, starting `vdv` in debug mode and entering the commands

```
tbreak 9
continue
```

causes the debugger to halt the simulation at the beginning of time step 9.0785 as indicated by the output of the `continue` command.

```
[Tm=9.07847133212036      ] **Start** of system 'vdv' outputs
```

### Breaking on Nonfinite Values

The `nanbreak` command stops a simulation when the simulation computes a value that is infinite or outside the range of values that can be represented by

the machine running the simulation. The `nanbreak` command is useful for pinpointing computational errors in a Simulink model.

## Breaking on Step-Size Limiting Steps

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

The `zcbreak` command causes the simulation to halt when Simulink detects a non-sampled zero crossing in a model that includes blocks where zero-crossings can arise. After halting, Simulink prints the location in the model, the time, and the type (rising or falling) of the zero-crossing. For example, setting a zero-crossing break at the start of execution of the `zeroxing` demo model

```
sldebug zeroxing
[Tm=0                ] **Start** of system 'zeroxing' outputs
(sldebug @0:0 'zeroxing/Sine Wave'): zcbreak
Break at zero crossing events is enabled.
```

and continuing the simulation

```
(sldebug @0:0 'zeroxing/Sine Wave'): continue
```

results in a rising zero-crossing break at

```
[Tm=0.34350110879329    ] Breaking at block 0:2

[Tm=0.34350110879329    ] Rising zero crossing on 3rd zcsignal
in block 0:2 'zeroxing/Saturation'
```

If a model does not include blocks capable of producing nonsampled zero-crossings, the command prints a message advising you of this fact.



## Displaying Information About the Simulation

The Simulink debugger provides a set of commands that allow you to display block states, block inputs and outputs, and other information while running a model.

### Displaying Block I/O

The debugger provides three commands for displaying block I/O. Each displays the I/O of a specified block. The main difference among them is when they display the I/O.

Command	Displays a Block's I/O...
probe	Immediately
disp	At every breakpoint
trace	Whenever the block is executed

### probe Command

The probe command prints the current inputs and outputs of a block that you specify. The command prints the block's I/O in the MATLAB command window.

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	Displays I/O of selected block.
probe s:b	Prints the I/O of the block specified by system number s and block number b.

The probe command comes in handy 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 block by block. Each time you step the model,

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. Similarly, suppose you are using the next command to step through a model by time steps. The next command does not display block I/O. However, if you need to examine a block's I/O after entering a next command, you can do so, using the probe command.

### **disp Command**

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's 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`.

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. So, you do not need to use the `disp` command if you are interested in watching only the I/O of the current block.

### **trace Command**

The `trace` command causes the debugger to display a specified block's I/O whenever Simulink evaluates the block. It lets you obtain a complete record of a block's I/O without having to stop the simulation. As with the other block I/O display commands, you can specify the block either by entering its block index or by selecting it in the model diagram. You can remove a block from the debugger's list of trace points, using the `untrace` command.

## **Displaying Algebraic Loop Information**

The `atrace` command causes the debugger to display information about a model's algebraic loops (see "Algebraic Loops" on page 9-7) each time they are

solved. The command takes a single argument that specifies the amount of information to display.

Command	Displays for Each Algebraic Loop ...
atrace 0	No information
atrace 1	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
atrace 2	Same as level 1
atrace 3	Level 2 plus the Jacobian matrix used to solve 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 command window. For example, the following sequence of commands shows the states of the Simulink bouncing ball demo (`bounce`) after its first and second time-steps.

```
sldebug bounce
[Tm=0                ] **Start** of system 'bounce' outputs
(sldebug @0:0 'bounce/Position'): states
Continuous state vector (value,index,name):
    10                0 (0:0 'bounce/Position')
    15                1 (0:5 'bounce/Velocit...')
(sldebug @0:0 'bounce/Position'): next
[Tm=0.01            ] **Start** of system 'bounce' outputs
(sldebug @0:0 'bounce/Position'): states
Continuous state vector (value,index,name):
    10.1495095        0 (0:0 'bounce/Position')
    14.9019           1 (0:5 'bounce/Velocit...')
```

## Displaying Integration Information

The `ishow` command toggles display of integration information. When enabled, this option causes the debugger to print a message each time that it takes a

time step or encounters a state that limits the size of a time step. In the first case, the debugger prints the size of the time step, for example,

```
[Tm=9.996264188473381      ] Step of 0.01 was taken by integrator
```

In the second case, the debugger displays the state that currently determines the size of time steps, for example,

```
[Ts=9.676264188473388      ] Integration limited by 1st state of  
block 0:0 'bounce/Position'
```

## Displaying Information About the Model

In addition to providing information about a simulation, the debugger can provide you with information about the model that underlies the simulation.

### Displaying a Model's Block Execution Order

Simulink determines the order in which to execute blocks at the beginning of a simulation run, during model initialization. During simulation, Simulink maintains a list of blocks sorted by execution order. This list is called the sorted list. You can display the sorted list at any time by typing `slist` at the debugger command prompt. The `slist` command lists the model's blocks in execution order. The list includes the block index for each command:

```
---- 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)
```

### Displaying a Block

To determine which block in a model's diagram corresponds to a particular index, type `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 system's window.

## Displaying a Model's Nonvirtual Systems

The `systems` command prints a list of the nonvirtual systems in the model being debugged. For example, the Simulink 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 in nature, that is, subsystems that the model diagram represents as Subsystem blocks but which Simulink solves as part of a parent system. In Simulink models, the root system and triggered or enabled subsystems are true systems. All other subsystems are virtual (that is, graphical) and hence do not appear in the listing produced by 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:

```
sldbg vdp
[Tm=0                ] **Start** of system 'vdp' outputs
(sldbg @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' (Output)
 0:2  'vdp/Integrator2' (Integrator)
 0:3  'vdp/Out2' (Output)
 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 in nature, that is, blocks that indicate relationships or groupings among computational blocks.

---

## Displaying Blocks with Potential Zero-Crossings

The `zclist` prints a list of blocks in which nonsampled zero-crossings can occur during a simulation. For example, `zclist` prints the following list for the clutch sample model:

```
(sdebug @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, type `ashow s#n`, where `s` is the index of the system (see “Displaying a Model’s Block Execution Order” on page 11-17) 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, type `ashow s:b`, where `s:b` is the block’s index. To clear algebraic-loop highlighting from the model diagram, enter `ashow clear`.



## Displaying Debug Settings

The `status` command displays the settings of various debug options, such as conditional breakpoints. 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.
```

## Debugger Command Reference

The following table lists the debugger commands. The table's Repeat column specifies whether pressing the **Return** key at the command line repeats the command. Detailed descriptions of the commands follow the table.

<b>Command</b>	<b>Short Form</b>	<b>Repeat</b>	<b>Description</b>
ashow	as	No	Show an algebraic loop.
atrace	at	No	Set algebraic loop trace level.
bafter	ba	No	Insert a breakpoint after execution of a block.
break	b	No	Insert a breakpoint before execution of a block.
bshow	bs	No	Show a specified block.
clear	cl	No	Clear a breakpoint from a block.
continue	c	Yes	Continue the simulation.
disp	d	Yes	Display a block's I/O when the simulation stops.
help	? or h	No	Display help for debugger commands.
ishow	i	No	Enable or disable display of integration information.
minor	m	No	Enable or disable minor step mode.
nanbreak	na	No	Set or clear break on nonfinite value.
next	n	Yes	Go to start of the next time step.
probe	p	No	Display a block's I/O.
quit	q	No	Abort simulation.
run	r	No	Run the simulation to completion.

<b>Command</b>	<b>Short Form</b>	<b>Repeat</b>	<b>Description</b>
slist	sli	No	List a model's nonvirtual blocks.
states	state	No	Display current state values.
status	stat	No	Display debugging options in effect.
step	s	Yes	Step to next block.
stop	sto	No	Stop the simulation.
systems	sys	No	List a model's nonvirtual systems.
tbreak	tb	No	Set or clear a time breakpoint.
trace	tr	Yes	Display a block's I/O each time it executes.
undisp	und	Yes	Remove a block from the debugger's list of display points.
untrace	unt	Yes	Remove a block from the debugger's list of trace point.
xbreak	x	No	Break when the debugger encounters a step-size-limiting state.
zcbreak	zcb	No	Break at nonsampled zero-crossing events.
zclist	zcl	No	List blocks containing nonsampled zero crossings.

**Purpose** Show an algebraic loop.

**Syntax** `ashow <gcb | s:b | s#n | clear>`

**Arguments**

<code>s:b</code>	The block whose system index is <code>s</code> and block index is <code>b</code> .
<code>gcb</code>	Current block.
<code>s#n</code>	The algebraic loop numbered <code>n</code> in system <code>s</code> .
<code>clear</code>	Switch that clears loop coloring.

**Description** `ashow gcb` or `ashow s:b` highlights the algebraic loop that contains the specified block. `ashow s#n` highlights the `n`th algebraic loop in system `s`. `ashow clear` removes algebraic loop highlights from the model diagram.

**See Also** `atrace`, `slist`

**Purpose** Set algebraic loop trace level.

**Syntax** `atrace level`

**Arguments** `level` Trace level (0 = none, 4 = everything).

**Description** The `atrace` command sets the algebraic loop trace level for a simulation.

<b>Command</b>	<b>Displays for Each Algebraic Loop ...</b>
<code>atrace 0</code>	No information
<code>atrace 1</code>	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
<code>atrace 2</code>	Same as level 1
<code>atrace 3</code>	Level 2 plus Jacobian matrix used to solve loop
<code>atrace 4</code>	Level 3 plus intermediate solutions of the loop variable

**See Also** `systems`, `states`

# bafter

---

<b>Purpose</b>	Insert a break point after a block is executed.
<b>Syntax</b>	<code>bafter gcb</code> <code>bafter s:b</code>
<b>Arguments</b>	<code>s:b</code> The block whose system index is <code>s</code> and block index is <code>b</code> . <code>gcb</code> Current block.
<b>Description</b>	The <code>bafter</code> command inserts a breakpoint after execution of the specified block.
<b>See Also</b>	<code>break</code> , <code>xbreak</code> , <code>tbreak</code> , <code>nanbreak</code> , <code>zcbreak</code> , <code>slist</code>

<b>Purpose</b>	Insert a break point before a block is executed.				
<b>Syntax</b>	<pre>break gcb break s:b</pre>				
<b>Arguments</b>	<table><tr><td>s:b</td><td>The block whose system index is s and block index is b.</td></tr><tr><td>gcb</td><td>Current block.</td></tr></table>	s:b	The block whose system index is s and block index is b.	gcb	Current block.
s:b	The block whose system index is s and block index is b.				
gcb	Current block.				
<b>Description</b>	The break command inserts a breakpoint before execution of the specified block.				
<b>See Also</b>	bafter, tbreak, xbreak, nanbreak, zcbreak, slist				

# bshow

---

<b>Purpose</b>	Show a specified block.
<b>Syntax</b>	bshow s:b
<b>Arguments</b>	s:b            The block whose system index is s and block index is b.
<b>Description</b>	This command opens the model window containing the specified block and selects the block.
<b>See Also</b>	slist



<b>Purpose</b>	Clear a breakpoint from a block.
<b>Syntax</b>	<code>clear gcb</code> <code>clear s:b</code>
<b>Arguments</b>	<code>s:b</code> The block whose system index is s and block index is b. <code>gcb</code> Current block.
<b>Description</b>	The <code>clear</code> command clears a breakpoint from the specified block.
<b>See Also</b>	<code>bafter</code> , <code>slist</code>

# continue

---

**Purpose** Continue the simulation.

**Syntax** `continue`

**Description** The `continue` command continues the simulation from the current breakpoint. The simulation continues until it reaches another breakpoint or its final time step.

**See Also** `run`, `stop`, `quit`

**Purpose**            Display a block's I/O when the simulation stops.

**Syntax**            `disp gcb`  
                      `disp s:b`  
                      `disp`

**Arguments**        `s:b`            The block whose system index is `s` and block index is `b`.  
                      `gcb`            Current block.

**Description**        The `disp` command registers a block as a display point. The debugger displays the inputs and outputs of all display points in the MATLAB command window whenever the simulation halts. Invoking `disp` without arguments shows a list of display points. Use `undisp` to unregister a block.

**See Also**            `undisp`, `slist`, `probe`, `trace`

# help

---

**Purpose** Display help for debugger commands.

**Syntax** help

**Description** The help command displays a list of debugger commands in the command window. The list includes the syntax and a brief description of each command.

<b>Purpose</b>	Enable or disable display of integration information.
<b>Syntax</b>	ishow
<b>Description</b>	The <code>ishow</code> command toggles display of integration information during a simulation.
<b>See Also</b>	<code>atrace</code>

# minor

---

**Purpose** Enable or disable minor step mode.

**Syntax** `minor`

**Description** The `minor` command causes the debugger to enter or exit minor step mode. In minor step mode, the `step` command advances the simulation by blocks within a minor step. In minor step mode, after executing the last block in the model's sorted block list, the `step` command advances the simulation to the next minor time step, if any minor time steps remain in the current major time step; otherwise, the `step` command advances the simulation to the first minor time step in the next major time step.

**See Also** `step`

<b>Purpose</b>	Set or clear nonfinite value break mode.
<b>Syntax</b>	nanbreak
<b>Description</b>	The nanbreak command causes the debugger to break whenever the simulation encounters a nonfinite (NaN or Inf) value. If nonfinite break mode is set, nanbreak clears it.
<b>See Also</b>	break, bafter, xbreak, tbreak, zcbreak

# next

---

**Purpose** Go to start of the next time step.

**Syntax** next

**Description** The next command evaluates all of the blocks remaining to be evaluated in the current time step, stopping at the start of the next time step. After executing the next command, the debugger highlights the first block to be evaluated on the next time step and displays the time of the next step.

**See Also** step



**Purpose** Display a block's I/O.

**Syntax** `probe [<s:b | gcb>]`

**Arguments**

<code>s:b</code>	The block whose system index is <code>s</code> and block index is <code>b</code> .
<code>gcb</code>	Current block.

**Description** `probe` causes the debugger to enter or exit probe mode. In probe mode, the debugger displays the I/O of any block you select. To exit probe mode, type any command. `probe gcb` displays the I/O of the currently selected block. `probe s:b` displays the I/O of the block whose index is `s:b`.

**See Also** `disp`, `trace`

# quit

---

**Purpose** Abort simulation.

**Syntax** quit

**Description** The quit command terminates the current simulation.

**See Also** stop

<b>Purpose</b>	Run the simulation to completion.
<b>Syntax</b>	run
<b>Description</b>	The <code>run</code> command runs the simulation from the current breakpoint to its final time step. It ignores breakpoints and display points.
<b>See Also</b>	<code>continue</code> , <code>stop</code> , <code>quit</code>

# slist

---

**Purpose** List a model's nonvirtual blocks.

**Syntax** `slist`

**Description** The `slist` command lists the nonvirtual blocks in the model being debugged. The list shows the block index and name of each listed block.

**See Also** `systems`

**Purpose**            Display current state values.

**Syntax**            `states`

**Description**        The `states` command displays a list of the current states of the model. The display lists the value, index, and name of each state.

**See Also**            `ishow`

# systems

---

<b>Purpose</b>	List a model's nonvirtual systems.
<b>Syntax</b>	systems
<b>Description</b>	The systems command lists a model's nonvirtual systems in the MATLAB command window.
<b>See Also</b>	slist

**Purpose**            Display debugging options in effect.

**Syntax**            status

**Description**      The status command displays a list of the debugging options in effect.

# step

---

**Purpose** Step to next block.

**Syntax** step

**Description** The step command evaluates the next block to be evaluated in the current time step. After executing the step command, the debugger highlights the next block to be evaluated and its output signal lines. It also displays the name of the next block as part of the debugger command-line prompt.

**See Also** next



<b>Purpose</b>	Stop the simulation.
<b>Syntax</b>	<code>stop</code>
<b>Description</b>	The <code>stop</code> command stops the simulation.
<b>See Also</b>	<code>continue</code> , <code>run</code> , <code>quit</code>

# tbreak

---

**Purpose** Set or clear a time breakpoint.

**Syntax** tbreak t  
tbreak

**Description** The tbreak command sets a breakpoint at the specified time step. If a breakpoint already exists at the specified time, tbreak clears the breakpoint. If you do not specify a time, tbreak toggles a breakpoint at the current time step.

**See Also** break, bafter, xbreak, nanbreak, zcbreak

<b>Purpose</b>	Display a block's I/O each time the block executes.
<b>Syntax</b>	<code>trace gcb</code> <code>trace s:b</code>
<b>Arguments</b>	<code>s:b</code> The block whose system index is s and block index is b. <code>gcb</code> Current block.
<b>Description</b>	The trace command registers a block as a trace point. The debugger displays the I/O of each registered block each time the block executes.
<b>See Also</b>	<code>disp</code> , <code>probe</code> , <code>untrace</code> , <code>slist</code>

# undisp

---

<b>Purpose</b>	Remove a block from the debugger's list of display points.
<b>Syntax</b>	<code>undisp gcb</code> <code>undisp s:b</code>
<b>Arguments</b>	<code>s:b</code> The block whose system index is s and block index is b. <code>gcb</code> Current block.
<b>Description</b>	The <code>undisp</code> command removes the specified block from the debugger's list of display points.
<b>See Also</b>	<code>disp</code> , <code>slist</code>

<b>Purpose</b>	Remove a block from the debugger's list of trace points.				
<b>Syntax</b>	<pre>untrace gcb untrace s:b</pre>				
<b>Arguments</b>	<table><tr><td>s:b</td><td>The block whose system index is s and block index is b.</td></tr><tr><td>gcb</td><td>Current block.</td></tr></table>	s:b	The block whose system index is s and block index is b.	gcb	Current block.
s:b	The block whose system index is s and block index is b.				
gcb	Current block.				
<b>Description</b>	The <code>untrace</code> command removes the specified block from the debugger's list of trace points.				
<b>See Also</b>	<code>trace</code> , <code>slist</code>				

# xbreak

---

<b>Purpose</b>	Break when the debugger encounters a step-size-limiting state.
<b>Syntax</b>	<code>xbreak</code>
<b>Description</b>	The <code>xbreak</code> command pauses execution of the model when the debugger encounters a state that limits the size of the steps that the solver takes. If <code>xbreak</code> mode is already on, <code>xbreak</code> turns the mode off.
<b>See Also</b>	<code>break</code> , <code>bafter</code> , <code>zcbreak</code> , <code>tbreak</code> , <code>nanbreak</code>

<b>Purpose</b>	Toggle breaking at nonsampled zero-crossing events.
<b>Syntax</b>	zcbreak
<b>Description</b>	The zcbreak command causes the debugger to break when a nonsampled zero-crossing event occurs. If zero-crossing break mode is already on, zcbreak turns the mode off.
<b>See Also</b>	break, bafter, xbreak, tbreak, nanbreak, zclist

# zclist

---

**Purpose** List blocks containing nonsampled zero crossings.

**Syntax** `zclist`

**Description** The `zclist` command prints a list of blocks in which nonsampled zero crossings can occur. The command prints the list in the MATLAB command window.

**See Also** `zcbreak`



# Model and Block Parameters

---

<b>Introduction</b> . . . . .	A-2
<b>Model Parameters</b> . . . . .	A-3
<b>Common Block Parameters</b> . . . . .	A-7
<b>Block-Specific Parameters</b> . . . . .	A-10
<b>Mask Parameters</b> . . . . .	A-24

## Introduction

This appendix lists model, block, and mask parameters. The tables that list the parameters provide enough information to enable you to modify models from the command line, using the `set_param` command, described in Chapter 10.

## Model Parameters

This table lists and describes parameters that describe a model. The parameters appear in the order they are defined in the model file, described in Appendix B. The table also includes model callback parameters, described in “Using Callback Routines” on page 3-53. The **Description** column indicates where you can set the value on the **Simulation Parameters** dialog box. Model parameters that are simulation parameters are described in more detail in Chapter 4. Examples showing how to change parameters follow the table.

Parameter values must be specified as quoted strings. The string contents depend on the parameter and can be numeric (scalar, vector, or matrix), a variable name, a filename, or a particular value. The **Values** column shows the type of value required, the possible values (separated with a vertical line), and the default value, enclosed in braces.

**Table A-1: Model Parameters**

Parameter	Description	Values
Name	Model name	text
Version	Simulink version used to modify the model (read-only)	(release)
SimParamPage	<b>Simulation Parameters</b> dialog box page to display (page last displayed)	{Solver}   WorkspaceI/O   Diagnostics
SampleTimeColors	<b>Sample Time Colors</b> menu option	on   {off}
InvariantConstants	Invariant constant setting	on   {off}
WideVectorLines	<b>Wide Vector Lines</b> menu option	on   {off}
ShowLineWidths	<b>Show Line Widths</b> menu option	on   {off}
PaperOrientation	Printing paper orientation	portrait   {landscape}
PaperPosition	Position of diagram on paper	[left, bottom, width, height]
PaperPositionMode	Paper position mode	auto   {manual}
PaperSize	Size of PaperType in PaperUnits	[width height] (read only)

**Table A-1: Model Parameters (Continued)**

Parameter	Description	Values
PaperType	Printing paper type	{usletter}   uslegal   a0   a1   a2   a3   a4   a5   b0   b1   b2   b3   b4   b5   arch-A   arch-B   arch-C   arch-D   arch-E   A   B   C   D   E   tabloid
PaperUnits	Printing paper size units	normalized   {inches}   centimeters   points
StartTime	Simulation start time	scalar {0.0}
StopTime	Simulation stop time	scalar {10.0}
Solver	Solver	{ode45}   ode23   ode113   ode15s   ode23s   ode5   ode4   ode3   ode2   ode1   FixedStepDiscrete   VariableStepDiscrete
RelTol	Relative error tolerance	scalar {1e-3}
AbsTol	Absolute error tolerance	scalar {1e-6}
Refine	Refine factor	scalar {1}
MaxStep	Maximum step size	scalar {auto}
InitialStep	Initial step size	scalar {auto}
FixedStep	Fixed step size	scalar {auto}
MaxOrder	Maximum order for ode15s	1   2   3   4   {5}
OutputOption	Output option	AdditionalOutputTimes   {RefineOutputTimes}   SpecifiedOutputTimes
OutputTimes	Values for chosen OutputOption	vector {[]}
LoadExternalInput	Load input from workspace	on   {off}
ExternalInput	Time and input variable names	scalar or vector [t, u]

**Table A-1: Model Parameters (Continued)**

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
SaveTime	Save simulation time	{on}   off
TimeSaveName	Simulation time name	variable {tout}
SaveState	Save states	on   {off}
StateSaveName	State output name	variable {xout}
SaveOutput	Save simulation output	{on}   off
OutputSaveName	Simulation output name	variable {yout}
LoadInitialState	Load initial state	on   {off}
InitialState	Initial state name or values	variable or vector {xInitial}
SaveFinalState	Save final state	on   {off}
FinalStateName	Final state name	variable {xFinal}
LimitMaxRows	Limit output	on   {off}
MaxRows	Maximum number of output rows to save	scalar {1000}
Decimation	Decimation factor	scalar {1}
AlgebraicLoopMsg	Algebraic loop diagnostic	none   {warning}   error
MinStepSizeMsg	Minimum step size diagnostic	{warning}   error
UnconnectedInputMsg	Unconnected input ports diagnostic	none   {warning}   error
UnconnectedOutputMsg	Unconnected output ports diagnostic	none   {warning}   error
UnconnectedLineMsg	Unconnected lines diagnostic	none   {warning}   error
ConsistencyChecking	Consistency checking	on   {off}
ZeroCross	Intrinsic zero crossing detection (see “Zero Crossings” on page 9-3)	{on}   off
CloseFcn	Close callback	command or variable
PreLoadFcn	Pre-load callback	command or variable
PostLoadFcn	Post-load callback	command or variable

**Table A-1: Model Parameters (Continued)**

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
SaveFcn	Save callback	command or variable
StartFcn	Start simulation callback	command or variable
StopFcn	Stop simulation callback	command or variable
BooleanDataType	Enable Boolean mode	on   {off}
BufferReuse	Enable reuse of block I/O buffers	{on}   off

These examples show how to set model parameters for the `mymodel` system.

This command sets the simulation start and stop times.

```
set_param('mymodel', 'StartTime', '5', 'StopTime', '100')
```

This command sets the solver to `ode15s` and changes the maximum order.

```
set_param('mymodel', 'Solver', 'ode15s', 'MaxOrder', '3')
```

This command associates a `SaveFcn` callback.

```
set_param('mymodel', 'SaveFcn', 'my_save_cb')
```

## Common Block Parameters

This table lists the parameters common to all Simulink blocks, including block callback parameters, which are described in “Using Callback Routines” on page 3-53. Examples of commands that change these parameters follow this table.

**Table A-2: Common Block Parameters**

Parameter	Description	Values
Name	Block's name	string
Type	Simulink object type (read-only)	'block'
Parent	Name of the system that owns the block	string
BlockType	Block type	text
BlockDescription	Block description	text
Description	User-specifiable description	text
InputPorts	Array of input port locations	[h1,v1; h2,v2; ...]
OutputPorts	Array of output port locations	[h1,v1; h2,v2; ...]
CompiledPortWidths	Structure of port widths	scalar and vector
Orientation	Where block faces	{right}   left   down   up
ForegroundColor	Block name, icon, outline, output signals, and signal label	{black}   white   red   green   blue   cyan   magenta   yellow   gray   lightBlue   orange   darkGreen
BackgroundColor	Block icon background	black   {white}   red   green   blue   cyan   magenta   yellow   gray   lightBlue   orange   darkGreen
DropShadow	Display drop shadow	{off}   on
NamePlacement	Position of block name	{normal}   alternate
FontName	Font	{Helvetica}
FontSize	Font size	{10}

**Table A-2: Common Block Parameters (Continued)**

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
FontWeight	Font weight	(system-dependent) light   {normal}   demi   bold
FontAngle	Font angle	(system-dependent) {normal}   italic   oblique
Position	Position of block in model window	vector [left top right bottom] <i>not</i> enclosed in quotes
ShowName	Display block name	{on}   off
Tag	User-defined string	' '
UserData	Any MATLAB data type (not saved in the mdl file)	[ ]
Selected	Block selected state	on   {off}
CloseFcn	Close callback	MATLAB expression
CopyFcn	Copy callback	MATLAB expression
DeleteFcn	Delete callback	MATLAB expression
InitFcn	Initialization callback	MATLAB expression
LoadFcn	Load callback	MATLAB expression
ModelCloseFcn	Model close callback	MATLAB expression
NameChangeFcn	Block name change callback	MATLAB expression
OpenFcn	Open callback	MATLAB expression
ParentCloseFcn	Parent subsystem close callback	MATLAB expression
PreSaveFcn	Pre-save callback	MATLAB expression
PostSaveFcn	Post-save callback	MATLAB expression
StartFcn	Start simulation callback	MATLAB expression
StopFcn	Termination of simulation callback	MATLAB expression



**Table A-2: Common Block Parameters (Continued)**

Parameter	Description	Values
UndoDeleteFcn	Undo block delete callback	MATLAB expression
LinkStatus	Link status of block.	none   resolved   unresolved   implicit
AttributesFormatString	Specifies parameters to be displayed below block in a block diagram	string

These examples illustrate how to change these parameters.

This command changes the orientation of the Gain block in the `mymodel` system so it faces the opposite direction (right to left).

```
set_param('mymodel/Gain','Orientation','left')
```

This command associates an `OpenFcn` callback with the Gain block in the `mymodel` system.

```
set_param('mymodel/Gain','OpenFcn','my_open_cb')
```

This command sets the `Position` parameter of the Gain block in the `mymodel` system. The block is 75 pixels wide by 25 pixels high. The position vector is *not* enclosed in quotes.

```
set_param('mymodel/Gain','Position',[50 250 125 275])
```

## Block-Specific Parameters

These tables list block-specific parameters for all Simulink blocks. When setting block parameters with the `set_param` command, you identify the block by specifying its `BlockType` parameter. The `BlockType` appears in parentheses after the block name.

The table includes detailed information only for built-in blocks, not for masked blocks, although the table includes the `MaskType` parameter value for masked blocks. For more information, see “Mask Parameters” on page A-24.

The **Dialog Box Prompt** column indicates the text of the prompt for the parameter on the block’s dialog box. The **Values** column shows the type of value required (scalar, vector, variable), the possible values (separated with a vertical line), and the default value (enclosed in braces).

**Table A-3: Sources Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Band-Limited White Noise (Continuous White Noise)(masked)		
Chirp Signal (chirp) (masked)		
Clock (Clock) (no block-specific parameters)		
Constant (Constant)		
Value	Constant value	scalar or vector {1}
Digital Clock (DigitalClock)		
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Digital Pulse Generator		
From File (FromFile)		
FileName	Filename	filename {untitled.mat}
From Workspace (FromWorkspace)		
VariableName	Matrix table	matrix {[T,U]}

**Table A-3: Sources Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Pulse Generator (Pulse Generator) (masked)		
Ramp (Ramp) (masked)		
Random Number (RandomNumber)		
Seed	Initial seed	scalar or vector {0}
Repeating Sequence (Repeating table) (masked)		
Signal Generator (SignalGenerator)		
WaveForm	Wave form	{sine}   square   sawtooth   random
Amplitude	Amplitude	scalar or vector {1}
Frequency	Frequency	scalar or vector {1}
Units	Units	{Hertz}   rad/sec
Sine Wave (Sin)		
Amplitude	Amplitude	scalar or vector {1}
Frequency	Frequency	scalar or vector {1}
Phase	Phase	scalar or vector {0}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
Step (Step)		
Time	Step time	scalar or vector {1}
Before	Initial value	scalar or vector {0}
After	Final value	scalar or vector {1}
Uniform Random Number (Uniform RandomNumber)		
Minimum	Minimum	scalar or vector {-1}
Maximum	Maximum	scalar or vector {1}

**Table A-3: Sources Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Seed	Initial Seed	scalar or vector {0}
SampleTime	Sample Time	scalar or vector {0}

**Table A-4: Sinks Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Display (Display)		
Format	Format	{short}   long   short_e   long_e   bank
Decimation	Decimation	scalar {1}
Floating	Floating display	{off} on
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
Scope (Scope)		
Location	Position of Scope window on screen	vector {[left top right bottom]}
Open	(If Scope open when the model is opened. Cannot set from dialog box)	{off}   on
NumInputPorts	Number of Axes	positive integer > 0
TickLabels	Hide tick labels	{on}   off
ZoomMode	(Zoom button initially pressed)	{on}   xonly   yonly
AxesTitles	Title (on right click axes)	scalar {auto}
Grid	(for future use)	{on}   off
TimeRange	Time range	scalar {auto}

**Table A-4: Sinks Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
YMin	Y min	scalar {-5}
YMax	Y max	scalar {5}
SaveToWorkspace	Save data to workspace	{off}   on
SaveName	Variable name	variable {ScopeData}
DataFormat	Format	{matrix   structure}
LimitMaxRows	Limit rows to last	{on}   off
MaxRows	(no label)	scalar {5000}
Decimation	(Value if Decimation selected)	scalar {1}
SampleInput	(Toggles with Decimation)	{off}   on
SampleTime	(SampleInput value)	scalar (sample period) {0} or vector [period offset]
Stop Simulation (StopSimulation) (no block-specific parameters)		
To File (ToFile)		
Filename	Filename	filename {untitled.mat}
MatrixName	Variable name	variable {ans}
Decimation	Decimation	scalar {1}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
To Workspace (ToWorkspace)		
VariableName	Variable name	variable {simout}
Buffer	Maximum number of rows	scalar {inf}
Decimation	Decimation	scalar {1}

**Table A-4: Sinks Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
XY Graph (XY scope.) (masked)		

**Table A-5: Discrete Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Discrete Filter (DiscreteFilter)		
Numerator	Numerator	vector {[1]}
Denominator	Denominator	vector {[1 2]}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Discrete State-Space (DiscreteStateSpace)		
A	A	matrix {1}
B	B	matrix {1}
C	C	matrix {1}
D	D	matrix {1}
X0	Initial conditions	vector {0}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Discrete-Time Integrator (DiscreteIntegrator)		
IntegratorMethod	Integrator method	{ForwardEuler}   BackwardEuler   Trapezoidal
ExternalReset	External reset	{none}   rising   falling   either
InitialConditionSource	Initial condition source	{internal}   external
InitialCondition	Initial condition	scalar or vector {0}

**Table A-5: Discrete Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
LimitOutput	Limit output	{off}   on
UpperSaturationLimit	Upper saturation limit	scalar or vector {inf}
LowerSaturationLimit	Lower saturation limit	scalar or vector {-inf}
ShowSaturationPort	Show saturation port	{off}   on
ShowStatePort	Show state port	{off}   on
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Discrete Transfer Fcn (DiscreteTransferFcn)		
Numerator	Numerator	vector {[1]}
Denominator	Denominator	vector {[1 0.5]}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Discrete Zero-Pole (DiscreteZeroPole)		
Zeros	Zeros	vector {[1]}
Poles	Poles	vector [0 0.5]
Gain	Gain	scalar {1}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
First-Order Hold (First Order Hold) (masked)		
Unit Delay (UnitDelay)		
X0	Initial condition	scalar or vector {0}
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]
Zero-Order Hold (ZeroOrderHold)		
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]

**Table A-6: Continuous Library Block Parameters**

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Derivative (Derivative) (no block-specific parameters)		
Integrator (Integrator)		
ExternalReset	External reset	{none}   rising   falling   either
InitialConditionSource	Initial condition source	{internal}   external
InitialCondition	Initial condition	scalar or vector {0}
LimitOutput	Limit output	{off}   on
UpperSaturationLimit	Upper saturation limit	scalar or vector {inf}
LowerSaturationLimit	Lower saturation limit	scalar or vector {-inf}
ShowSaturationPort	Show saturation port	{off}   on
ShowStatePort	Show state port	{off}   on
AbsoluteTolerance	Absolute tolerance	scalar {auto}
Memory (Memory)		
X0	Initial condition	scalar or vector {0}
InheritSampleTime	Inherit sample time	{off}   on
State-Space (StateSpace)		
A	A	matrix {1}
B	B	matrix {1}
C	C	matrix {1}
D	D	matrix {1}
X0	Initial conditions	vector {0}
Transfer Fcn (TransferFcn)		
Numerator	Numerator	vector or matrix {[1]}



**Table A-6: Continuous Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Denominator	Denominator	vector {[1 1]}
Transport Delay (TransportDelay)		
DelayTime	Time delay	scalar or vector {1}
InitialInput	Initial input	scalar or vector {0}
BufferSize	Initial buffer size	scalar {1024}
Variable Transport Delay (VariableTransportDelay)		
MaximumDelay	Maximum delay	scalar or vector {10}
InitialInput	Initial input	scalar or vector {0}
MaximumPoints	Buffer size	scalar {1024}
Zero-Pole (ZeroPole)		
Zeros	Zeros	vector {[1]}
Poles	Poles	vector {[0 -1]}
Gain	Gain	vector {[1]}

**Table A-7: Math Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Abs (Abs) (no block-specific parameters)		
Algebraic Constraint (Algebraic Constraint) (masked)		
Combinatorial Logic (CombinatorialLogic)		
TruthTable	Truth table	matrix {[0 0;0 1;0 1;1 0; 0 1;1 0;1 0;1 1]}
Complex to Magnitude-Angle		
Complex to Real-Imag		

**Table A-7: Math Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Dot Product (Dot Product) (masked)		
Gain (Gain)		
Gain	Gain	scalar or vector {1}
Logical Operator (Logic)		
Operator	Operator	{AND}   OR   NAND   NOR   XOR   NOT
Inputs	Number of input ports	scalar {2}
Magnitude-Angle to Complex		
Math Function (Math)		
Operator	Function	{exp}   log   log10   square   sqrt   pow   reciprocal   hypot   rem   mod
Matrix Gain (Matrix Gain) (masked)		
MinMax (MinMax)		
Function	Function	{min}   max
Inputs	Number of input ports	scalar {1}
Product (Product)		
Inputs	Number of inputs	scalar {2}
Relational Operator (RelationalOperator)		
Operator	Operator	==   !=   <   {<=}   >=   >
Relational Operator (RelationalOperator)		
Operator	Operator	==   !=   <   {<=}   >=   >
Rounding Function (Rounding)		
Operator	Function	{floor}   ceil   round   fix
Sign (Signum) (no block-specific parameters)		
Slider Gain (SliderGain) (masked)		

**Table A-7: Math Library Block Parameters (Continued)**

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Sum (Sum)		
Inputs	List of signs	scalar or list of signs {++}
Trigonometric Function (Trigonometry)		
Operator	Function	{sin}   cos   tan   asin   acos   atan   atan2   sinh   cosh   tanh

**Table A-8: Functions and Tables Block Parameters**

Block (BlockType)/Parameter	Dialog Box Prompt	Values
Fcn (Fcn)		
Expr	Expression	expression {sin(u(1)*exp(2.3*(-u(2))))}
Look-up Table (Lookup)		
InputValues	Vector of input values	vector {[-5:5]}
OutputValues	Vector of output values	vector {tanh([-5:5])}
Look-Up Table (2-D) (Lookup Table (2-D))		
RowIndex	Row	vector
ColumnIndex	Column	vector
OutputValues	Table	2-D matrix
MATLAB Fcn (MATLABFcn)		
MATLABFcn	MATLAB function	MATLAB function {sin}
OutputWidth	Output width	scalar or vector {-1}
S-Function (S-Function)		
FunctionName	S-function name	name {system}
Parameters	S-function parameters	additional parameters if needed

**Table A-9: Nonlinear Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Backlash (Backlash)		
BacklashWidth	Deadband width	scalar or vector {1}
InitialOutput	Initial output	scalar or vector {0}
Coulomb & Viscous Friction (Coulombic and Viscous Friction) (masked)		
Dead Zone (DeadZone)		
LowerValue	Start of dead zone	scalar or vector {-0.5}
UpperValue	End of dead zone	scalar or vector {0.5}
Manual Switch		
Multiport Switch (MultiPortSwitch)		
Inputs	Number of inputs	scalar or vector {3}
Quantizer (Quantizer)		
QuantizationInterval	Quantization interval	scalar or vector {0.5}
Rate Limiter (RateLimiter)		
RisingSlewLimit	Rising slew rate	scalar or vector {1.}
FallingSlewLimit	Falling slew rate	scalar or vector {-1.}
Relay (Relay)		
OnSwitchValue	Switch on point	scalar or vector {eps}
OffSwitchValue	Switch off point	scalar or vector {eps}
OnOutputValue	Output when on	scalar or vector {1}
OffOutputValue	Output when off	scalar or vector {0}
Saturation (Saturate)		
UpperLimit	Upper limit	scalar or vector {0.5}
LowerLimit	Lower limit	scalar or vector {-0.5}
S-Function (S-Function)		

**Table A-9: Nonlinear Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
FunctionName	S-function name	name {system}
Parameters	S-function parameters	additional parameters if needed
Sign (Signum) (no block-specific parameters)		
Switch (Switch)		
Threshold	Threshold	scalar or vector {0}

**Table A-10: Signals & Systems Library Block Parameters**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Bus Selector		
Configurable Subsystem (SubSystem)		
Choice	Block choice	string
LibraryName	Library name	string
Data Store Memory (DataStoreMemory)		
DataStoreName	Data store name	tag {A}
InitialValue	Initial value	vector {0}
Data Store Read (DataStoreRead)		
DataStoreName	Data store name	tag {A}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
Data Store Write (DataStoreWrite)		
DataStoreName	Data store name	tag {A}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]
Data Type Conversion		

**Table A-10: Signals & Systems Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Demux (Demux)		
Outputs	Number of outputs	scalar or vector {3}
Enable (EnablePort)		
StatesWhenEnabling	States when enabling	{held}   reset
ShowOutputPort	Show output port	{off}   on
From (From)		
GotoTag	Goto tag	tag {A}
Goto (Goto)		
GotoTag	Tag	tag {A}
TagVisibility	Tag visibility	{local}   scoped   global
Goto Tag Visibility (GotoTagVisibility)		
GotoTag	Goto tag	tag {A}
Ground (Ground) (no block-specific parameters)		
Hit Crossing (HitCross)		
HitCrossingOffset	Hit crossing offset	scalar or vector {0}
HitCrossingDirection	Hit crossing direction	rising   falling   {either}
ShowOutputPort	Show output port	{on}   off
IC (InitialCondition)		
Value	Initial value	scalar or vector {1}
In (Inport)		
Port	Port number	scalar {1}
PortWidth	Port width	scalar {-1}
SampleTime	Sample time	scalar (sample period) {-1} or vector [period offset]

**Table A-10: Signals & Systems Library Block Parameters (Continued)**

<b>Block (BlockType)/Parameter</b>	<b>Dialog Box Prompt</b>	<b>Values</b>
Merge		
Model Info		
Mux (Mux)		
Inputs	Number of inputs	scalar or vector {3}
Out (Output)		
Port	Port number	scalar {1}
OutputWhenDisabled	Output when disabled	{held}   reset
InitialOutput	Initial output	scalar or vector {0}
Probe (Probe)		
ProbeWidth	Probe width	{on}   off
ProbeSampleTime	Probe sample time	{on}   off
ProbeComplexSignal	Probe complex signal	{on}   off
Subsystem (SubSystem)		
ShowPortLabels	Show/Hide Port Labels <b>Format</b> menu item	{on}   off
Terminator (Terminator) (no block-specific parameters)		
Trigger (TriggerPort)		
TriggerType	Trigger type	{rising}   falling   either   function-call
ShowOutputPort	Show output port	{off}   on
Width (Width) (no block-specific parameters)		

## Mask Parameters

This section lists parameters that describe masked blocks. This table lists masking parameters, which correspond to **Mask Editor** dialog box parameters.

**Table A-11: Mask Parameters**

Parameter	Dialog Box Parameter	Values
MaskType	Mask type	string
MaskDescription	Block description	string
MaskHelp	Block help	string
MaskPrompts	Prompt (see below)	cell array of strings
MaskPromptString	Prompt (see below)	delimited string
MaskStyles	Control type (see below)	cell array {Edit}   Checkbox   Popup
MaskStyleString	Control type (see below)	{Edit}   Checkbox   Popup
MaskVariables	Variable (see below)	string
MaskInitialization	Initialization commands	MATLAB command
MaskDisplay	Drawing commands	display commands
MaskIconFrame	Icon frame (Visible is on, Invisible is off)	{on}   off
MaskIconOpaque	Icon transparency (Opaque is on, Transparent is off)	{on}   off
MaskIconRotate	Icon rotation (Rotates is on, Fixed is off)	on   {off}
MaskIconUnits	Drawing coordinates	Pixel   {Autoscale}   Normalized
MaskValues	Block parameter values (see below)	cell array of strings
MaskValueString	Block parameter values (see below)	delimited string
MaskTunableValues	Tunable parameter attributes	cell array of strings
MaskTunableValueString	Tunable parameter attributes	delimited string



When you use the **Mask Editor** to create a dialog box parameter for a masked block, you provide this information:

- The prompt, which you enter in the **Prompt** field
- The variable that holds the parameter value, which you enter in the **Variable** field
- The type of field created, which you specify by selecting a **Control type**
- Whether the value entered in the field is to be evaluated or stored as a literal, which you specify by selecting an **Assignment type**

The mask parameters, listed in the table on the previous page, store the values specified for the dialog box parameters in these ways:

- The **Prompt** field values for all dialog box parameters are stored in the `MaskPromptString` parameter as a string, with individual values separated by a vertical bar (|), as shown in this example.

```
"Slope:|Intercept:"
```

- The **Variable** field values for all dialog box parameters are stored in the `MaskVariables` parameter as a string, with individual assignments separated by a semi-colon. A sequence number indicates which prompt is associated with a variable. A special character preceding the sequence number indicates the **Assignment type**: @ indicates **Evaluate**, & indicates **Literal**.

For example, "a=@1;b=&2;" indicates that the value entered in the first parameter field is assigned to variable a and is evaluated in MATLAB before assignment, and the value entered in the second field is assigned to variable b and is stored as a literal, which means that its value is the string entered in the dialog box.

- The **Control type** field values for all dialog box parameters are stored in the `MaskStyleString` parameter as a string, with individual values separated by a comma. The **Popup strings** values appear after the popup type, as shown in this example:

```
"edit,checkbox,popup(red|blue|green)"
```

- The parameter values are stored in the `MaskValueString` mask parameter as a string, with individual values separated by a vertical bar. The order of the values is the same as the order the parameters appear on the dialog box.

For example, these statements define values for the parameter field prompts and the values for those parameters.

```
MaskPromptString    "Slope: |Intercept: "  
MaskValueString    "2|5"
```

# Model File Format

---

<b>Model File Contents</b> . . . . .	B-2
Model Section . . . . .	B-3
BlockDefaults Section . . . . .	B-3
AnnotationDefaults Section . . . . .	B-3
System Section . . . . .	B-3
 <b>A Sample Model File</b> . . . . .	 B-4

## Model File Contents

A model file is a structured ASCII file that contains keywords and parameter-value pairs that describe the model. The file describes model components in hierarchical order.

The structure of the model file is as follows.

```
Model {
  <Model Parameter Name> <Model Parameter Value>
  ...
  BlockDefaults {
    <Block Parameter Name> <Block Parameter Value>
    ...
  }
  AnnotationDefaults {
    <Annotation Parameter Name> <Annotation Parameter Value>
    ...
  }
  System {
    <System Parameter Name> <System Parameter Value>
    ...
    Block {
      <Block Parameter Name> <Block Parameter Value>
      ...
    }
    Line {
      <Line Parameter Name> <Line Parameter Value>
      ...
      Branch {
        <Branch Parameter Name> <Branch Parameter Value>
        ...
      }
    }
    Annotation {
      <Annotation Parameter Name> <Annotation Parameter Value>
      ...
    }
  }
}
```

The model file consists of sections that describe different model components:

- The `Model` section defines model parameters.
- The `BlockDefaults` section contains default settings for blocks in the model.
- The `AnnotationDefaults` section contains default settings for annotations in the model.
- The `System` section contains parameters that describe each system (including the top-level system and each subsystem) in the model. Each `System` section contains block, line, and annotation descriptions.

All model and block parameters are described in Appendix A.

## **Model Section**

The `Model` section, located at the top of the model file, defines the values for model-level parameters. These parameters include the model name, the version of Simulink used to last modify the model, and simulation parameters.

## **BlockDefaults Section**

The `BlockDefaults` section appears after the simulation parameters and defines the default values for block parameters within this model. These values can be overridden by individual block parameters, defined in the `Block` sections.

## **AnnotationDefaults Section**

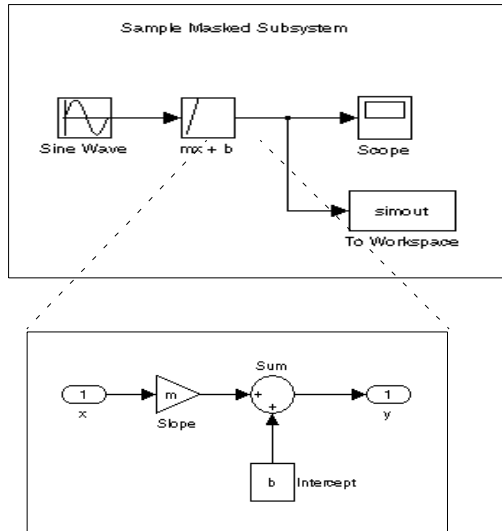
The `AnnotationDefaults` section appears after the `BlockDefaults` section. This section defines the default parameters for all annotations in the model. These parameter values cannot be modified using the `set_param` command.

## **System Section**

The top-level system and each subsystem in the model are described in a separate `System` section. Each `System` section defines system-level parameters and includes `Block`, `Line`, and `Annotation` sections for each block, line, and annotation in the system. Each `Line` that contains a branch point includes a `Branch` section that defines the branch line.

## A Sample Model File

This model file describes the sample system that illustrates masking, described in Chapter 6. The model and subsystem look like this.



The  $mx + b$  subsystem

The model file, `mask_example.mdl`, follows.

```
Model {
  Name      "slopeintercept"
  Version   3.00
  SimParamPage "Solver"
  SampleTimeColors off
  InvariantConstants off
  WideVectorLines off
  ShowLineWidths off
  ShowPortDataTypes off
  StartTime "0.0"
  StopTime  "10.0"
  SolverMode "Auto"
  Solver     "ode45"
  RelTol    "1e-3"
  AbsTol    "auto"
  Refine    "1"
  MaxStep   "auto"
  InitialStep "auto"
  FixedStep "auto"
  MaxOrder  5
  OutputOption "RefineOutputTimes"
  OutputTimes "[]"
  LoadExternalInput off
  ExternalInput "[t, u]"
  SaveTime on
  TimeSaveName "tout"
  SaveState off
  StateSaveName "xout"
  SaveOutput on
  OutputSaveName "yout"
  LoadInitialState off
  InitialState "xInitial"
  SaveFinalState off
  FinalStateName "xFinal"
  SaveFormat "Matrix"
  LimitMaxRows off
  MaxRows    "1000"
  Decimation "1"
```

```
AlgebraicLoopMsg "warning"
MinStepSizeMsg "warning"
UnconnectedInputMsg "warning"
UnconnectedOutputMsg "warning"
UnconnectedLineMsg "warning"
InheritedTsInSrcMsg "warning"
IntegerOverflowMsg "none"
UnnecessaryDatatypeConvMsg "none"
Int32ToFloatConvMsg "warning"
SignalLabelMismatchMsg "none"
ConsistencyChecking "off"
ZeroCross on
SimulationMode "normal"
BlockDataTips on
BlockParametersDataTip on
BlockAttributesDataTip off
BlockPortWidthsDataTip off
BlockDescriptionStringDataTipoff
BlockMaskParametersDataTip off
ToolBar on
StatusBar on
BrowserShowLibraryLinks off
BrowserLookUnderMasks off
OptimizeBlockIOStorage on
BufferReuse on
BooleanDataType off
RTWSystemTargetFile "grt.tlc"
RTWInlineParameters off
RTWRetainRTWFile off
RTWTemplateMakefile "grt_default_tmf"
RTWMakeCommand "make_rtw"
RTWGenerateCodeOnly off
ExtModeMexFile "ext_comm"
ExtModeBatchMode off
ExtModeTrigType "manual"
ExtModeTrigMode "oneshot"
ExtModeTrigPort "1"
ExtModeTrigElement "any"
ExtModeTrigDuration 1000
ExtModeTrigHoldOff 0
```



```
ExtModeTrigDelay 0
ExtModeTrigDirection "rising"
ExtModeTrigLevel 0
ExtModeArchiveMode "off"
ExtModeAutoIncOneShot off
ExtModeIncDirWhenArm off
ExtModeAddSuffixToVar off
ExtModeWriteAllDataToWs off
ExtModeArmWhenConnect off
Created "Tue Jul 28 15:18:08 1998"
Creator "JoeSchmo"
UpdateHistory "UpdateHistoryNever"
ModifiedByFormat "%<Auto>"
LastModifiedBy "MoeSchmo"
ModifiedDateFormat "%<Auto>"
LastModifiedDate "Fri Oct 30 15:17:59 1998"
ModelVersionFormat "1.%<AutoIncrement:4>"
ConfigurationManager "none"
BlockDefaults {
  Orientation "right"
  ForegroundColor "black"
  BackgroundColor "white"
  DropShadow off
  NamePlacement "normal"
  FontName "Helvetica"
  FontSize 10
  FontWeight "normal"
  FontAngle "normal"
  ShowName on
}
AnnotationDefaults {
  HorizontalAlignment "center"
  VerticalAlignment "middle"
  ForegroundColor "black"
  BackgroundColor "white"
  DropShadow off
  FontName "Helvetica"
  FontSize 10
  FontWeight "normal"
  FontAngle "normal"
}
```

```
}
LineDefaults {
  FontName    "Helvetica"
  FontSize    9
  FontWeight  "normal"
  FontAngle   "normal"
}
System {
  Name        "slopeintercept"
  Location    [629, 86, 984, 341]
  Open        on
  ModelBrowserVisibility off
  ModelBrowserWidth 200
  ScreenColor "automatic"
  PaperOrientation "landscape"
  PaperPositionMode "auto"
  PaperType    "usletter"
  PaperUnits   "inches"
  ZoomFactor   "100"
  AutoZoom     on
  ReportName   "simulink-default.rpt"
  Block {
    BlockType  Scope
    Name        "Scope"
    Ports       [1, 0, 0, 0, 0]
    Position    [205, 79, 235, 111]
    Floating    off
    Location    [188, 365, 512, 604]
    Open        off
    NumInputPorts "1"
    TickLabels   "OneTimeTick"
    ZoomMode     "on"
    List {
      ListTypeAxesTitles
      axes1      "%<SignalLabel>"
    }
    Grid        "on"
    TimeRange    "auto"
    YMin         "-5"
    YMax         "5"
  }
}
```

```

    SaveToWorkspace    off
    SaveName           "ScopeData"
    DataFormat         "StructureWithTime"
    LimitMaxRows       on
    MaxRows             "5000"
    Decimation          "1"
    SampleInput        off
    SampleTime         "0"
}
Block {
    BlockType          Sin
    Name               "Sine Wave"
    Position            [35, 80, 65, 110]
    Amplitude           "1"
    Frequency           "1"
    Phase              "0"
    SampleTime         "0"
}
Block {
    BlockType          ToWorkspace
    Name               "To Workspace"
    Position            [200, 150, 260, 180]
    VariableName       "simout"
    Buffer              "inf"
    Decimation          "1"
    SampleTime         "-1"
    SaveFormat         "Structure"
}
Block {
    BlockType          SubSystem
    Name               "mx + b"
    Ports              [1, 1, 0, 0, 0]
    Position            [105, 80, 135, 110]
    ShowPortLabels     on
    MaskType           "SampleMaskedBlock"
    MaskDescription    "Models the equation for a line, y =
mx + b.\nTh"
    MaskHelp           "e slope and intercept are mask block parameters."
    MaskHelp           "Enter the slope (m) and intercept (b) in the
bl"
}

```

"ock dialog parameter fields.\n\nThe block generates y for a given input x."

```
MaskPromptString      "Slope:|Intercept:"
MaskStyleString       "edit,edit"
MaskTunableValueString "on,on"
MaskCallbackString    "|"
MaskEnableString      "on,on"
MaskVisibilityString  "on,on"
MaskVariables         "m=@1;b=@2;"
MaskDisplay           "plot([0 1],[0 m] + (m<0))"
MaskIconFrame        on
MaskIconOpaque        on
MaskIconRotate        "none"
MaskIconUnits         "normalized"
MaskValueString       " 4|2"
System {
  Name      "mx + b"
  Location[175, 110, 504, 342]
  Open      off
  ModelBrowserVisibilityoff
  ModelBrowserWidth200
  ScreenColor"white"
  PaperOrientation"landscape"
  PaperPositionMode"auto"
  PaperType"usletter"
  PaperUnits"inches"
  ZoomFactor"100"
  AutoZoomon
  Block {
    BlockType Inport
    Name      "x"
    Position  [15, 68, 45, 82]
    Port      "1"
    PortWidth "-1"
    SampleTime "-1"
    DataType  "auto"
    SignalType "auto"
    Interpolate on
  }
  Block {
```

```
    BlockType Constant
    Name        "Intercept"
    Position    [145, 130, 175, 160]
    Orientation "up"
    Value       "b"
}
Block {
  BlockType Gain
  Name        "Slope"
  Position    [80, 60, 110, 90]
  Gain        "m"
  SaturateOnIntegerOverflow on
}
Block {
  BlockType Sum
  Name        "Sum"
  Ports       [2, 1, 0, 0, 0]
  Position    [145, 60, 175, 90]
  NamePlacement "alternate"
  IconShape   "round"
  Inputs      "|++"
  SaturateOnIntegerOverflow on
}
Block {
  BlockType Outport
  Name        "y"
  Position    [225, 68, 255, 82]
  Port        "1"
  OutputWhenDisabled "held"
  InitialOutput "[]"
}
Line {
  SrcBlock    "Intercept"
  SrcPort     1
  DstBlock    "Sum"
  DstPort     2
}
Line {
  SrcBlock    "Sum"
  SrcPort     1
}
```

```
        DstBlock "y"
        DstPort 1
    }
    Line {
        SrcBlock "x"
        SrcPort 1
        DstBlock "Slope"
        DstPort 1
    }
    Line {
        SrcBlock "Slope"
        SrcPort 1
        DstBlock "Sum"
        DstPort 1
    }
    }
}
Line {
    SrcBlock "Sine Wave"
    SrcPort 1
    DstBlock "mx + b"
    DstPort 1
}
Line {
    SrcBlock "mx + b"
    SrcPort 1
    Points [25, 0]
    Branch {
DstBlock"Scope"
DstPort 1
    }
    Branch {
Points [0, 70]
DstBlock"To Workspace"
DstPort 1
    }
}
Annotation {
    Position [135, 27]
    Text "Sample Masked Subsystem"
```

```
}  
}  
}
```





## A

- Abs block 8-11
  - zero crossings 9-6
- absolute tolerance 4-13, 4-32, 8-105
  - simulation accuracy 4-28
- absolute value, generating 8-11
- accuracy of derivative 8-49
- accuracy of simulation 4-28
- Ada Extension to Real-Time Workshop 1-12
- Adams-Bashforth-Moulton PECE solver 4-11
- add\_block command 10-4
- add\_line command 10-5
- adding
  - block inputs 8-191
  - blocks 10-4
  - lines 10-5
- Algebraic Constraint block 8-12
- algebraic equations, specifying 8-12
- algebraic loops 9-7
  - detection 9-2
  - integrator block reset or IC port 8-60
  - simulation speed 4-28
- alignment of blocks 3-11
- analysis functions, perturbing model 8-100
- AnnotationDefaults section of mdl file B-3
- annotations
  - annotation block, see Model Info block 8-131
  - changing font 3-37
  - creating 3-37
  - definition 3-37
  - deleting 3-37
  - editing 3-37
  - manipulating with mouse and keyboard 3-49
  - moving 3-37
  - using to document models 3-57
- Apply button on Mask Editor 6-9
- ashow debug command 11-24

- Assignment mask parameter 6-10
- atrace debug command 11-25
- attributes format string 3-18
- AttributesFormatString block parameter 3-14, 3-17
- Autoscale icon drawing coordinates 6-25
- auto-scaling Scope axes 8-166

## B

- Backlash block 8-14
  - zero crossings 9-6
- backpropagating sample time 9-17
- Backspace key 3-14, 3-33, 3-37
- Backward Euler method 8-59
- Backward Rectangular method 8-59
- bad link 3-22
- bafter debug command 11-26
- Band-Limited White Noise block 8-18, 8-150, 8-212
  - simulation speed 4-28
- bdclose command 10-6
- bdroot command 10-7
- Block data tips 3-9
- block descriptions
  - creating 6-6
  - entering 6-26
- block diagrams, printing 3-62
- block dialog boxes
  - closing 10-8
  - opening 3-13, 10-20
- block icons
  - drawing coordinates 6-24
  - font 3-16
  - icon frame property 6-23
  - icon rotation property 6-24

- icon transparency property 6-24
- properties 6-23
- question marks in 6-21, 6-23
- transfer functions on 6-21
- block indexes 11-4
- block libraries
  - Blocksets and Toolboxes 8-3
  - Demos 8-3
  - Discrete 8-5
  - Extras 8-3
  - Linear 8-5, 8-6
  - Nonlinear 8-7, 8-8
  - Sinks 8-4
  - Sources 8-3
- block names
  - changing location 3-17
  - copied blocks 3-11
  - editing 3-16
  - flipping location 3-17
  - font 3-16
  - hiding and showing 3-17
  - location 3-16
  - newline character in 10-3
  - rules 3-16
  - sequence numbers 3-11, 3-12
  - slash character in 10-3
- block parameters A-7, A-10-A-11
  - changing during simulation 10-24
  - Continuous library A-16
  - copying 3-11, 3-12
  - Discrete library A-14
  - displaying beneath a block icon 3-17
  - evaluating 9-2
  - Functions and Tables library A-19
  - Math library A-17
  - modifying 4-2
  - Nonlinear library A-20
  - prompts 6-10
  - scalar expansion 3-18, 3-19
  - Signals and Systems library A-21
  - Sinks library A-12
  - Sources library A-10
- block priorities
  - assigning 3-19
- block type of masked block 6-26
- BlockDefaults section of mdl file B-3
- blocks 3-9-3-20
  - adding to model 10-4
  - alignment 3-11
  - callback parameters 3-55
  - callback routines 3-53
  - connecting 2-10, 3-27
  - connections, checking 9-2
  - copying 3-26
  - copying from block library 3-22
  - copying into models 3-10
  - copying to other applications 3-12
  - current 10-14
  - deleting 3-14, 10-10
  - disconnecting 3-18
  - discrete 9-13
  - drop shadows 3-20
  - duplicating 3-12
  - grouping to create subsystem 3-52
  - handle of current 10-15
  - library 3-21
  - moving between windows 3-12
  - moving in a model 2-9, 3-12
  - orientation 3-15
  - path 10-3
  - reference 3-21, 3-22
  - replacing 10-21
  - resizing 3-15
  - reversing signal flow through 3-59

- signal flow through 3-15
  - under mask 6-9
  - updating 9-2
  - updating from library 3-23
  - vectorization 3-18
- blocksets
- DSP Blockset 1-14
  - Fixed-Point Blockset 1-14
  - Nonlinear Control Design Blockset 1-16
  - Power System Blockset 1-16
- Blocksets and Toolboxes library 8-3
- bode function 5-11
- Bogacki-Shampine formula 4-11, 4-12
- Boolean expressions, modeling 8-25
- boolean type checking 4-26
- bounding box
- grouping blocks for subsystem 3-52
  - selecting objects 3-7
- branch lines 3-28, 3-59
- break debug command 11-27
- Break Library Link menu item 3-24
- breaking link to library block 3-23
- breakpoints
- clearing from blocks 11-11
  - setting 11-9
  - setting at beginning of a block 11-10
  - setting at end of block 11-11
  - setting at timesteps 11-11
  - setting on nonfinite values 11-11
  - setting on step-size limiting steps 11-12
  - setting on zero crossings 11-12
- Browser 3-66
- bshow debug command 11-28
- building models
- exercise 2-6
  - tips 3-57
- ## C
- callback parameters
- block 3-55
  - model 3-54
- callback routines 3-53
- canceling a command 3-7
- capping unconnected blocks 8-196
- changing
- annotations, font 3-37
  - block icons, font 3-16
  - block names, font 3-16
  - block names, location 3-17
  - block size 3-15
  - sample time during simulation 9-13
  - signal labels, font 3-33
- check box control type 6-13
- Chirp Signal block 8-22
- clear debug command 11-29
- Clear menu item 3-14
- Clock block 8-24
- example 5-3
- Close Browser menu item 3-67
- Close button on Mask Editor 6-9
- Close menu item 2-3
- Close Model menu item 3-67
- close\_system command 10-8
- CloseFcn block callback parameter 3-55
- CloseFcn model callback parameter 3-54
- closing
- block dialog boxes 10-8
  - model windows 10-6
  - system windows 10-8
- clutch demo 8-96
- colors for sample times 9-15
- Combinatorial Logic block 8-25
- combining input lines into vector line 8-136
- Communications Toolbox 1-5

- Complex to Magnitude-Angle block 8-28
- Complex to Real-Imag block 8-29
- conditionally executed subsystems 7-2
- Configurable Subsystem block 8-30
- configuration manager 3-73
- connecting blocks 2-10, 3-27
- connecting lines to input ports 2-11
- consistency checking 4-24
- Constant block 8-34
- constant sample time 9-11
- constant value, generating 8-34
- continue debug command 11-30
- Continue menu item 4-5
- Continuous block library
  - block parameters A-16
- control input 7-2
- control signal 7-2
- Control System Toolbox 1-6
  - linearization 5-5
- control type 6-12
  - check box 6-13
  - edit 6-12
  - pop-up 6-13
- Copy menu item 3-11, 3-12
- copy, definition 3-21
- CopyFcn block callback parameter 3-55
- copying
  - block parameters 3-11, 3-12
  - blocks 3-10
  - library block into a model 3-22
  - signal labels 3-33
- Coulomb and Viscous Friction block 8-35
- Create Mask menu item 6-9
- Create Subsystem menu item 3-52, 8-190
- Created model parameter 3-77
- creating
  - annotations 3-37

- block libraries 3-21
- first mask prompt 6-11
- masked block descriptions 6-6
- masked block icons 6-6
- models 3-3, 10-19
- signal labels 3-33
- subsystems 3-51-3-56
- Creator model parameter 3-77
- current block 10-14
  - handle 10-15
- current system 10-16
- Cut menu item 3-12, 3-14

## D

- Data Store Memory block 8-36
- Data Store Read block 8-38
- Data Store Write block 8-39
- Data Type Conversion block 8-41
- data types 3-38-3-45
  - displaying 3-43
  - propagation 3-43
  - specifying 3-43
- dbstop if error command 6-17
- dbstop if warning command 6-17
- Dead Zone block 8-43
  - zero crossings 9-6
- deadband 8-14
- debug commands
  - ashow 11-24
  - atrace 11-25
  - bafter 11-26
  - break 11-27
  - bshow 11-28
  - clear 11-29
  - continue 11-30
  - disp 11-31

- help 11-32
- ishow 11-33
- minor 11-34
- nanbreak 11-35
- next 11-36
- probe 11-37
- quit 11-38
- run 11-39
- slist 11-40
- states 11-41
- status 11-43
- step 11-44
- stop 11-45
- systems 11-42
- tbreak 11-46
- trace 11-47
- undisp 11-48
- untrace 11-49
- xbreak 11-50
- zcbreak 11-51
- zclist 11-52
- debugger
  - getting command help 11-4
  - starting 11-3
- debugging initialization commands 6-17
- decimation factor 4-32
  - saving simulation output 4-21
- decision tables, modeling 8-25
- default
  - solvers 4-10
- defining
  - mask type 6-6, 6-26
  - masked block descriptions 6-26
  - masked block help text 6-6
- delaying
  - and holding input signals 8-214
  - input by specified sample time 8-221
  - input by variable amount 8-216
- Delete key 3-14, 3-33, 3-37
- delete\_block command 10-10
- delete\_line command 10-11
- DeleteFcn block callback parameter 3-55
- deleting
  - annotations 3-37
  - blocks 3-14, 10-10
  - lines 10-11
  - mask prompts 6-12
  - signal labels 3-33
- demo model, running 2-2
- Demos library 8-3
- Demux block 8-45
- Derivative block 8-49
  - linearization 5-5
- derivatives
  - calculating 8-49, 9-3
  - limiting 8-152
- Description model parameter 3-78
- description of masked blocks 6-26
- Diagnostics page of Simulation Parameter dialog box 4-24
- diagonal line segments 3-28
- diagonal lines 3-27
- dialogs
  - creating for masked blocks 6-28-6-30
- Digital Clock block 8-51
- direct feedthrough 9-2
- disabled subsystem, output 7-4
- disabling zero crossing detection 4-25, 9-5
- disconnecting blocks 3-18
- discontinuities
  - detecting 5-11
  - zero crossings 9-3
- Discrete block library 8-5
  - block parameters A-14

- discrete blocks 9-13
    - in enabled subsystem 7-5
    - in triggered systems 7-10
  - Discrete Filter block 8-52
  - Discrete Pulse Generator block 8-54
  - discrete solver 4-10, 4-11, 4-12
  - Discrete State-Space block 8-56
  - discrete state-space model 5-10
  - Discrete Transfer Fcn block 8-65, 8-214
  - Discrete Zero-Pole block 8-67
  - Discrete-Time Integrator block 8-58
    - sample time colors 9-17
  - discrete-time systems 9-13
    - linearization 5-10
  - disp command 6-18
  - disp debug command 11-31
  - Display Alphabetical List menu item 3-67
  - Display block 8-69
  - Display Hierarchical List menu item 3-67
  - displaying
    - line widths 3-31
    - output trajectories 5-2
    - output values 8-69
    - signals graphically 8-163
    - transfer functions on masked block icons 6-21
    - vector signals 8-164
    - X-Y plot of signals 8-219
  - dlinmod function 5-4, 5-9, 5-10
  - dlinmod2 function 5-9
  - documentation page of Mask Editor 6-9
  - Dormand-Prince
    - formula 4-12
    - pair 4-11
  - Dot Product block 8-72
  - dpoly command 6-22
  - drawing coordinates 6-24
    - Autoscale 6-25
    - normalized 6-7, 6-25
    - Pixel 6-25
  - droots command 6-23
  - drop shadows 3-20
  - DSP Blockset 1-14
  - duplicating blocks 3-12
- ## E
- edit control type 6-12
  - editing
    - annotations 3-37
    - block names 3-16
    - mask prompts 6-11
    - models 3-3
    - signal labels 3-33
  - eigenvalues of linearized matrix 5-10
  - either trigger type 7-9
  - Elementary Math block
    - algebraic loops 9-7
  - Enable block 8-74
    - creating enabled subsystems 7-3
    - outputting enable signal 7-5
    - states when enabling 7-4
  - enabled subsystems 7-2, 7-3, 8-74
    - setting states 7-4
  - ending Simulink session 3-79
  - equations, modeling 3-58
  - equilibrium point determination 5-7
  - error tolerance 4-13
    - simulation accuracy 4-28
    - simulation speed 4-27
  - Euler's method 4-12
  - eval command and masked block help 6-27
  - Evaluate Assignment type 6-10
  - examples
    - Clock block 5-3

- continuous system 3-59
- converting Celsius to Fahrenheit 3-58
- equilibrium point determination 5-7
- linearization 5-4
- masking 6-3
- multirate discrete model 9-14
- return variables 5-2
- Scope block 5-2
- To Workspace block 5-3
- Transfer Function block 3-60
- Exit MATLAB menu item 2-13, 3-79
- Expand All menu item 3-67
- Expand Library Links menu item 3-67
- expressions, applying to block inputs 8-76, 8-121
- external inputs 4-31
  - from workspace 8-100
- extracting linear models 5-4, 5-9
- Extras block library 8-3

## F

- falling trigger 7-9
- Fcn block 8-76
  - compared to Math Function block 8-119
  - compared to Rounding Function block 8-161
  - compared to Trigonometric Function block 8-210
  - simulation speed 4-27
- file
  - reading from 8-82
  - writing to 4-5, 8-197
- final states, saving 4-21
- Financial Toolbox 1-6
- find\_system command 10-12
- finding library block 3-24
- finding objects 10-12
- Finite Impulse Response filter 8-52

- finite-state machines, implementing 8-25
- First-Order Hold block 8-78
  - compared to Zero-Order Hold block 8-78, 8-88
- fixed icon rotation 6-24
- fixed step size 4-13, 4-33
- Fixed-Point Blockset 1-14
- fixed-step solvers 4-9, 4-12
- Flip Block menu item 3-15, 3-59
- Flip Name menu item 3-17
- flip-flops, implementing 8-25
- floating Display block 4-2, 8-69
- floating Scope block 4-2, 8-170
- fohdemo demo 8-78, 8-88
- font
  - annotations 3-37
  - block icons 3-16
  - block names 3-16
  - signal labels 3-33
- Font menu item 3-16, 3-33
- Forward Euler method 8-58
- Forward Rectangular method 8-58
- fprintf command 6-19
- Frequency-Domain System Identification Toolbox 1-6
- From block 8-80
- From File block 8-82
- From Workspace block 8-85
- Function-Call Generator block 8-88
- Functions and Tables block library
  - block parameters A-19
- fundamental sample time 4-10
- Fuzzy Logic Toolbox 1-6

## G

- Gain block 8-89
  - and algebraic loops 9-7

- gain, varying during simulation 8-183
- Gaussian number generator 8-150
- gcb command 10-14
- gcbh command 10-15
- gcs command 10-16
- get\_param command 10-17
  - checking simulation status 4-29
- global Goto tag visibility 8-80, 8-91
- Go To Library Link menu item 3-24
- Goto block 8-91
- Goto Tag Visibility block 8-94
- Ground block 8-95
- grouping blocks 3-51

## H

- handle of current block 10-15
- handles on selected object 3-7
- hardstop demo 8-96
- held output of enabled subsystem 7-4
- held states of enabled subsystem 7-5
- Help button on Mask Editor 6-9
- help debug command 11-32
- help text for masked blocks 6-6, 6-27
- Heun's method 4-12
- Hide Name menu item 3-17, 3-53, 8-140
- Hide Port Labels menu item 3-53
- hiding block names 3-17
- hierarchy of model 3-57, 9-2
- Higher-Order Spectral Analysis Toolbox 1-6
- Hit Crossing block 8-96
  - zero crossing detection 4-25
  - zero crossings 9-4, 9-6
- hybrid systems
  - integrating 9-17
  - linearization 5-10
  - simulating 9-13

## I

- IC block 8-98
- icon frame mask property 6-23
- icon page of Mask Editor 6-9
- icon rotation mask property 6-24
- icon transparency mask property 6-24
- icons
  - creating for masked blocks 6-6, 6-18
  - displaying graphics on 6-20
  - displaying images on 6-21
  - displaying text on 6-18
  - transfer functions on 6-21
- Image Processing Toolbox 1-6
- improved Euler formula 4-12
- inf values in mask plotting commands 6-21
- Infinite Impulse Response filter 8-52
- InitFcn block callback parameter 3-55
- InitFcn model callback parameter 3-54
- initial conditions
  - determining 4-22
  - setting 8-98
  - specifying 4-21
- initial states 4-33
- initial step size 4-12, 4-13, 4-33
  - simulation accuracy 4-28
- initialization commands 6-15
  - debugging 6-17
- initialization page of Mask Editor 6-9
- Inport block 8-99
  - in subsystem 3-51, 3-52, 8-190
  - linearization 5-4
  - linmod function 5-9
  - supplying input to model 4-17
- input ports, unconnected 8-95
- inputs
  - adding 8-191
  - applying expressions to 8-76



- applying MATLAB function to 8-76, 8-121
- choosing between 8-134
- combining into vector line 8-136
- constraining 8-12
- delaying and holding 8-214
- delaying by specified time 8-221
- delaying by variable amount 8-216
- external 4-31
- from outside system 8-99
- from previous time step 8-124
- from workspace 8-100
- generating step between two levels 8-187
- loading from base workspace 4-17
- logical operations on 8-108
- mixing vector and scalar 3-18
- multiplying 8-89
- outputting minimum or maximum 8-129
- passing through stair-step function 8-148
- piecewise linear mapping 8-110, 8-113
- plotting 8-219
- reading from file 8-82
- scalar expansion 3-18
- sign of 8-177
- vector or scalar 3-18
- width of 8-218

inserting mask prompts 6-11

integration

- block input 8-103
- discrete-time 8-58

Integrator block 8-103

- algebraic loops 9-7
- example 3-59
- sample time colors 9-17
- simulation speed 4-28
- zero crossings 9-6

invariant constants 9-11

invisible icon frame 6-23

ishow debug command 11-33

## J

Jacobian matrices 4-12

Jacobians 5-9

## K

keyboard actions, summary 3-48

keyboard command 6-17

## L

labeling signals 3-32

labeling subsystem ports 3-53

LastModificationDate model parameter 3-78

left-hand approximation 8-58

libinfo command 3-24

libraries 3-21-3-26

- creating 3-21

- modifying 3-22

- searching 3-25

library block

- definition 3-21

- finding 3-24

library blocks, getting information about 3-24

Library Browser 3-25

library, definition 3-21

limit rows to last check box 4-21

limiting

- derivative of signal 8-152

- integral 8-104

- signals 8-162

line segments 3-28

- creating 3-29

- diagonal 3-28

- moving 3-29
- line vertices, moving 3-31
- Line Widths menu item 3-31
- Linear block library 8-5, 8-6
- linear models, extracting 5-4, 5-9
- linearization 5-4, 5-9
  - discrete-time systems 5-10
- linearized matrix, eigenvalues 5-10
- lines 3-27-3-32
  - adding 10-5
  - branch 3-28, 3-59
  - carrying the same signal 2-11
  - connecting to input ports 2-11
  - deleting 10-11
  - diagonal 3-27
  - dividing into segments 3-29
  - manipulating with mouse and keyboard 3-48
  - signals carried on 4-2
  - widths, displaying 3-31
- link
  - breaking 3-23
  - definition 3-21
  - to library block 3-22
  - unresolved 3-22
- LinkStatus block parameter 3-23
- linmod function 5-4, 5-9, 8-100
  - Transport Delay block 8-206
- linmod2 function 5-11
- Literal Assignment type 6-10
- LMI Control Toolbox 1-7
- load initial check box 4-22
- LoadFcn block callback parameter 3-55
- loading from base workspace 4-17
- loading initial states 4-22
- local Goto tag visibility 8-80, 8-91
- location of block names 3-16, 3-17
- logic circuits, modeling 8-25

- Logical Operator block 8-108
- Look Into System menu item 3-67
- Look Under Mask Dialog menu item 3-67
- Look Under Mask menu item 6-9
- Look-Up Table (2-D) block 8-113
- Look-Up Table block 8-110
- loops, algebraic 9-7
- lorenzs demo 8-219

## M

- Magnitude-Angle to Complex block 8-116
- Manual Switch block 8-118
- manual, organization 1-3
- Mask Editor 6-9
- mask help text 6-6
- Mask Subsystem menu item 6-4, 6-9
- mask type 6-6, 6-26
- mask workspace 6-5, 6-15
- masked blocks
  - block descriptions 6-6
  - control types 6-12
  - description 6-26
  - dialogs
    - creating dynamic 6-28-6-30
    - setting parameters for 6-28
  - documentation 6-26
  - help text 6-27
  - icons
    - creating 6-6, 6-18
    - displaying a transfer function on 6-22
    - displaying graphics on 6-20
    - displaying images on 6-21
    - displaying text on 6-18
    - setting properties of 6-23
  - initialization commands 6-15
  - looking under 6-9

- parameters 6-3, A-24
    - assigning values to 6-10
    - default values 6-14
    - predefined 6-29
    - prompts for 6-10
    - tunable 6-14
    - undefined 6-23
  - ports
    - displaying labels of 6-20
  - question marks in icon 6-21, 6-23
  - type 6-26
  - unmasking 6-9
- Math block library
  - block parameters A-17
- Math Function block 8-119
- mathematical functions, performing 8-119, 8-161, 8-210
- MATLAB Fcn block 8-121
  - simulation speed 4-27
- MATLAB function, applying to block input 8-76, 8-121
- Matrix Gain block 8-123
- matrix, writing to 8-199
- maximum number of output rows 4-33
- maximum order of ode15s solver 4-14, 4-33
- maximum step size 4-12, 4-13, 4-33
- maximum step size parameter 4-13
- mdl file 3-61, B-2
- Memory block 8-124
  - simulation speed 4-27
- memory issues 3-57
- memory region, shared 8-36, 8-38, 8-39
- menus 3-3
- Merge block 8-126
- MEX-file models, simulating 4-3
- M-file models, simulating 4-3
- M-file S-functions
  - simulation speed 4-27
- M-files, running simulation from 4-3
- MinMax block 8-129
  - zero crossings 9-6
- minor debug command 11-34
- mixed continuous and discrete systems 9-17
- Model Browser 3-66
- model files 3-61, B-2
  - names 3-61
- Model Info block 8-131
- model parameters for version control 3-77
- Model Predictive Control Toolbox 1-7
- ModelCloseFcn block callback parameter 3-55
- modeling
  - equations 3-58
  - strategies 3-57
- models
  - building 2-6
  - callback parameters 3-54
  - callback routines 3-53
  - closing 10-6
  - creating 3-3, 10-19
  - creating change histories for 3-76
  - editing 3-3
  - name, getting 10-7
  - organizing and documenting 3-57
  - parameters A-3
  - printing 3-62
  - properties of 3-72
  - saving 2-13, 3-61
  - selecting entire 3-8
  - simulating 4-30
  - tips for building 3-57
  - tracking versions of 3-70
  - version control properties of 3-77
- ModelVersion model parameter 3-78
- ModelVersionFormat model parameter 3-78

- ModifiedBy model parameter 3-77
- ModifiedByFormat model parameter 3-77
- ModifiedComment model parameter 3-78
- ModifiedDate model parameter 3-78
- ModifiedDateFormat model parameter 3-78
- ModifiedHistory> model parameter 3-78
- modifying libraries 3-22
- Monte Carlo analysis 4-29
- mouse actions, summary 3-48
- MoveFcn block callback parameter 3-55
- moving
  - annotations 3-37
  - blocks and lines 3-12
  - blocks between windows 3-12
  - blocks in a model 2-9, 3-12
  - line segments 3-29
  - line vertices 3-31
  - mask prompts 6-12
  - signal labels 3-33
- Mu-Analysis and Synthesis Toolbox 1-7
- multiplying block inputs
  - by constant, variable, or expression 8-89
  - by matrix 8-123
  - during simulation 8-183
  - together 8-143
- Multiport Switch block 8-134
- multirate systems 9-13, 9-14
  - linearization 5-10
- Mux block 8-136
  - changing number of input ports 2-10
  
- N**
- NAG Foundation Toolbox 1-7
- NameChangeFcn block callback parameter 3-55
- names
  - blocks 3-16
  - copied blocks 3-11
  - model files 3-61
- Nan values in mask plotting commands 6-21
- nanbreak debug command 11-35
- Neural Network Toolbox 1-7
- New Library menu item 3-21
- New menu item 3-3
- new\_system command 3-21, 10-19
- newline in block name 10-3
- next debug command 11-36
- Nonlinear block library 8-7, 8-8
  - block parameters A-20
- Nonlinear Control Design Blockset 1-16
- nonlinear systems, spectral analysis of 8-22
- normalized icon drawing coordinates 6-7, 6-25
- normally distributed random numbers 8-150
- numerical differentiation formula 4-11
- numerical integration 9-3
  
- O**
- objects
  - finding 10-12
  - path 10-3
  - selecting more than one 3-7
  - selecting one 3-7
- ode1 solver 4-12
- ode113 solver 4-11
  - hybrid systems 9-17
  - Memory block 4-27, 8-124
- ode15s solver 4-10, 4-11, 4-27
  - hybrid systems 9-17
  - maximum order 4-14, 4-33
  - Memory block 4-27, 8-124
  - unstable simulation results 4-28
- ode2 solver 4-12
- ode23 solver 4-11

- hybrid systems 9-17
  - ode23s solver 4-11, 4-14, 4-28
  - ode3 solver 4-12
  - ode4 solver 4-12
  - ode45 solver 4-10, 4-11
    - hybrid systems 9-17
  - ode5 solver 4-12
  - offset to sample time 9-13
  - opaque icon 6-24
  - Open menu item 3-3
  - Open System menu item 3-67
  - open\_system command 10-20
  - OpenFcn block callback parameter 3-56, 3-68
  - OpenFcn model callback parameter 3-69
  - opening
    - block dialog boxes 3-13, 10-20
    - Simulink block library 10-26
    - Subsystem block 3-52
    - system windows 10-20
  - operating point 5-9
  - Optimization Toolbox 1-7
  - options structure
    - getting values 4-36
    - setting values 4-32
  - ordering of states 4-22
  - organization of manual 1-3
  - orientation of blocks 3-15
  - Outport block 8-139
    - example 5-2
    - in subsystem 3-51, 3-52, 8-190
    - linearization 5-4
    - linmod function 5-9
  - output
    - additional 4-16
    - between trigger events 7-10
    - disabled subsystem 7-4
    - displaying values of 8-69
    - enable signal 7-5
    - maximum rows 4-33
    - options 4-15
    - outside system 8-139
    - refine factor 4-34
    - saving to workspace 4-20
    - selected elements of input vector 8-173
    - smoother 4-16
    - specifying for simulation 4-16
    - specifying points 4-34
    - switching between inputs 8-194
    - switching between values 8-158
    - trajectories, viewing 5-2
    - trigger signal 7-10
    - variables 4-34
    - vector or scalar 3-18
    - writing to file 4-5, 8-197
    - writing to workspace 4-5, 4-20, 8-199
    - zero within range 8-43
  - output ports
    - capping unconnected 8-196
    - Enable block 7-5
    - Trigger block 7-10
- P**
- PaperOrientation model parameter 3-64
  - PaperPosition model parameter 3-64
  - PaperPositionMode model parameter 3-64
  - PaperType model parameter 3-64
  - parameters
    - blocks A-7, A-10-A-11
    - getting values of 10-17
    - masked blocks A-24
    - model A-3
    - setting values of 10-24
  - Parameters menu item 2-12, 4-4, 4-8

- ParentCloseFcn block callback parameter 3-56
  - Partial Differential Equation Toolbox 1-8
  - Paste menu item 3-11, 3-12
  - path, specifying 10-3
  - Pause menu item 4-5
  - perturbation
    - factor 5-9
    - levels 5-12
  - phase-shifted wave 8-178
  - piecewise linear mapping 8-110, 8-113
  - Pixel icon drawing coordinates 6-25
  - plot command and masked block icon 6-20
  - plotting input signals 8-163, 8-219
  - pop-up control type 6-13
  - port labels 8-140, 8-190
    - displaying 6-20
  - ports
    - block orientation 3-15
    - labeling in subsystem 3-53
  - PostLoadFcn model callback parameter 3-54
  - PostSaveFcn block callback parameter 3-56
  - PostSaveFcn model callback parameter 3-54
  - PostScript file, printing to 3-64
  - Power System Blockset 1-16
  - PreLoadFcn model callback parameter 3-54
  - PreSaveFcn block callback parameter 3-56
  - PreSaveFcn model callback parameter 3-54
  - Print (Browser) menu item 3-67
  - print command 3-62
  - Print menu item 3-62
  - printing
    - block diagrams 3-62
    - to PostScript file 3-64
  - Priority block parameter 3-19
  - probe debug command 11-37
  - proceeding with suspended simulation 4-5
  - produce additional output option 4-16
  - produce specified output only option 4-16
  - Product block 8-143, 8-145
    - algebraic loops 9-7
  - programmable logic arrays, modeling 8-25
  - prompts
    - control types 6-12
    - creating 6-11
    - deleting 6-12
    - editing 6-11
    - inserting 6-11
    - masked block parameters 6-10
    - moving 6-12
  - propagation of signal labels 3-33
  - properties of Scope block 8-169
  - Pulse Generator block 8-146
  - purely discrete systems 9-13
- Q**
- QFT Control Design Toolbox 1-8
  - Quantizer block 8-148
    - modeling A/D converter 8-221
  - question marks in masked block icon 6-21, 6-23
  - quit debug command 11-38
  - Quit MATLAB menu item 2-13, 3-79
- R**
- randn function 8-150
  - random noise, generating 8-178
  - Random Number block 8-150
    - and Band-Limited White Noise block 8-18
    - simulation speed 4-28
  - random numbers, generating normally distributed 8-18
  - Rate Limiter block 8-152
  - reading data

- from data store 8-38
  - from file 8-82
  - from workspace 8-85
  - Real-Imag to Complex block 8-154
  - Real-Time Workshop 1-10
  - Real-Time Workshop Ada Extension 1-12
  - Redo menu item 3-5
  - reference block 3-22
    - definition 3-21
  - refine factor 4-16, 4-34
  - region of zero output 8-43
  - Relational Operator block 8-156
    - zero crossings 9-6
  - relative tolerance 4-13, 4-34
    - simulation accuracy 4-28
  - Relay block 8-158
    - zero crossings 9-6
  - Repeating Sequence block 8-160
  - replace\_block command 10-21
  - replacing blocks in model 10-21
  - reset
    - output of enabled subsystem 7-4
    - states of enabled subsystem 7-5
  - resetting state 8-105
  - resizing blocks 3-15
  - return variables, example 5-2
  - reversing direction of signal flow 3-59
  - Revert button on Mask Editor 6-9
  - right-hand approximation 8-59
  - rising trigger 7-8, 7-9
  - Robust Control Toolbox 1-8
  - Rosenbrock formula 4-11
  - Rotate Block menu item 3-15
  - rotates icon rotation 6-24
  - Rounding Function block 8-161
  - run debug command 11-39
  - Runge-Kutta (2,3) pair 4-11
  - Runge-Kutta (4,5) formula 4-11
  - Runge-Kutta fourth-order formula 4-12
  - running the simulation 2-12
- ## S
- sample model 2-6
  - sample time 9-13
    - backpropagating 9-17
    - changing during simulation 9-13
    - colors 9-15
    - constant 9-11
    - fundamental 4-10
    - offset 9-13
    - parameter 9-13
    - simulation speed 4-27
  - Sample Time Colors menu item 9-12, 9-16
  - sample-and-hold, applying to block input 8-124
  - sample-and-hold, implementing 8-221
  - sampled data systems 9-13
  - sampling interval, generating simulation time 8-51
  - Saturation block 8-162
    - zero crossings 9-4, 9-6
  - Save As menu item 3-61
  - Save menu item 2-13, 3-61
  - save options area 4-20
  - save to workspace area 4-20
  - save\_system command 3-24, 10-23
  - saving
    - axes settings on Scope 8-168
    - final states 4-21, 4-22
    - models 2-13, 3-61
    - output to workspace 4-20
    - systems 10-23
  - sawtooth wave, generating 8-178
  - scalar expansion 3-18

- Scope block 8-163
  - example 3-60, 5-2
  - properties 8-169
- scoped Goto tag visibility 8-80, 8-91
- Select All menu item 3-8
- selecting
  - model 3-8
  - more than one object 3-7
  - one object 3-7
- Selector block 8-173
- separating vector signal 8-45
- sequence numbers on block names 3-11, 3-12
- sequence of signals 8-54, 8-146, 8-160
- sequential circuits, implementing 8-27
- Set Font dialog box 3-16
- set\_param command 3-24, 10-24
  - running a simulation 4-29
- setting breakpoints 11-9
- setting parameter values 10-24
- S-Function block 8-175
- Shampine, L. F. 4-12
- shared data store 8-36, 8-38, 8-39
- Show Browser menu item 3-67
- Show Name menu item 3-17
- show output port
  - Enable block 7-5
  - Trigger block 7-10
- showing block names 3-17
- Sign block 8-177
  - zero crossings 9-6
- signal flow through blocks 3-15
- Signal Generator block 8-178
- signal labels
  - changing font 3-33
  - copying 3-33
  - creating 3-33
  - deleting 3-33
  - editing 3-33
  - moving 3-33
  - propagation 3-33
  - using to document models 3-57
- Signal Processing Toolbox 1-8
- signal properties
  - setting 3-34
- Signal Properties Dialog 3-35
- signals 3-27
  - delaying and holding 8-214
  - displaying vector 8-164
  - labeling 3-32
  - limiting 8-162
  - limiting derivative of 8-152
  - passed from Goto block 8-80
  - passing to From block 8-91
  - plotting 8-163, 8-219
  - pulses 8-54, 8-146
  - repeating 8-160
  - vector 3-18
- Signals and Systems block library
  - block parameters A-21
- sim command 4-29, 4-30
- simget command 4-36
- simset command 4-32
- simulating models 4-30
- simulation
  - accuracy 4-28
  - command line 4-29



- displaying information about
  - algebraic loops 11-13, 11-14, 11-20
  - block execution order 11-17
  - block I/O 11-13
  - debug settings 11-21
  - integration 11-15
  - nonvirtual blocks 11-18
  - nonvirtual systems 11-18
  - system states 11-15
  - zero crossings 11-20
- menu 4-4
- proceeding with suspended 4-5
- running 2-12
- running incrementally 11-6
- speed 4-27
- starting 4-4
- stepping by blocks 11-6
- stepping by breakpoints 11-8
- stepping by time steps 11-7
- stopping 2-13, 4-5, 8-189
- suspending 4-5
- Simulation Diagnostics Dialog Box 4-6
- simulation parameters 4-8
  - setting 4-4
  - specifying 2-12, 4-4
  - specifying using `simset` command 4-32
- Simulation Parameters dialog box 2-12, 4-4, 4-8-4-25, A-3
- simulation time
  - compared to clock time 4-9
  - generating at sampling interval 8-51
  - outputting 8-24
  - writing to workspace 4-20
- Simulink
  - ending session 3-79
  - icon 3-2
  - menus 3-3
    - Real-Time Workshop 1-10
    - starting 3-2
    - windows and screen resolution 3-5
- Simulink block library 3-2
  - opening 10-26
- `simulink` command 3-2, 10-26
- sine wave
  - generating 8-178, 8-180
  - generating with increasing frequency 8-22
- Sine Wave block 8-180
- Sinks block library 8-4
  - block parameters A-12
- size of block, changing 3-15
- sizes vector 4-22
- slash in block name 10-3
- `sldebug` command 11-3
- Slider Gain block 8-183
- `slist` debug command 11-40
- Solver page of Simulation Parameters dialog box 4-8
- solver properties, specifying 4-32
- solvers 4-9-4-12
  - changing during simulation 4-2
  - choosing 4-4
  - default 4-10
  - discrete 4-10, 4-11, 4-12
  - fixed-step 4-9, 4-12
  - `ode1` 4-12
  - `ode113` 4-11, 4-27
  - `ode15s` 4-10, 4-11, 4-14, 4-27, 4-28
  - `ode2` 4-12
  - `ode23` 4-11
  - `ode23s` 4-11, 4-14, 4-28
  - `ode3` 4-12
  - `ode4` 4-12
  - `ode45` 4-10, 4-11
  - `ode5` 4-12

- specifying using `simset` command 4-34
- variable-step 4-9, 4-11
- Sources block library 8-3
  - block parameters A-10
- spectral analysis of nonlinear systems 8-22
- speed of simulation 4-27
- Spline Toolbox 1-8
- square wave, generating 8-178
- `ss2tf` function 5-12
- `ss2zp` function 5-12
- `stairs` function 9-14
- stair-step function, passing signal through 8-148
- Start menu item 2-2, 2-12, 3-59, 4-4
- start time 4-9
- `StartFcn` block callback parameter 3-56
- `StartFcn` model callback parameter 3-54
- starting Simulink 3-2
- state derivatives, setting to zero 5-13
- state events 9-3
- state space in discrete system 8-56
- states
  - absolute tolerance for 8-105
  - between trigger events 7-10
  - determining 9-3
  - initial 4-22, 4-33
  - loading initial 4-22
  - ordering of 4-22
  - outputting 4-34
  - resetting 8-105
  - saving at end of simulation 4-33
  - saving final 4-21, 4-22
  - updating 9-13
  - when enabling 7-4
  - writing to workspace 4-20
- states debug command 11-41
- State-Space block 8-185
  - algebraic loops 9-7
- Statistics Toolbox 1-9
- Status bar 3-5
- status debug command 11-43
- Step block 8-187
  - zero crossings 9-6
- step debug command 11-44
- step size 4-12
  - accuracy of derivative 8-49
  - simulation speed 4-27
- stiff problems 4-12
- stiff systems and simulation time 4-27
- stop debug command 11-45
- Stop menu item 2-3, 2-13, 4-5
- Stop Simulation block 8-189
- stop time 4-9
- Stop Time parameter 2-13
- `StopFcn` block callback parameter 3-56
- `StopFcn` model callback parameter 3-54
- stopping simulation 8-189
- Subsystem block 8-190
  - adding to create subsystem 3-51
  - opening 3-52
  - zero crossings 9-7
- subsystems
  - and Inport blocks 8-99
  - creating 3-51-3-56
  - labeling ports 3-53
  - model hierarchy 3-57
  - path 10-3
  - underlying blocks 3-52
- Sum block 8-191
  - algebraic loops 9-7
- summary of mouse and keyboard actions 3-48
- suspending simulation 4-5
- Switch block 8-194
  - zero crossings 9-7

switching output between inputs 8-118, 8-194  
switching output between values 8-158  
Symbolic Math Toolbox 1-9  
System Identification Toolbox 1-9  
System section of mdl file B-3  
systems  
    current 10-16  
    path 10-3  
systems debug command 11-42

## T

tbreak debug command 11-46  
terminating MATLAB 2-13  
terminating Simulink 2-13  
terminating Simulink session 3-79  
Terminator block 8-196  
text command 6-18  
tf2ss utility 8-203  
time delay, simulating 8-206  
time interval and simulation speed 4-27  
tips for building models 3-57  
To File block 8-197  
To Workspace block 8-199  
    example 5-3  
toolboxes 1-5  
    Communications Toolbox 1-5  
    Control System Toolbox 1-6  
    Financial Toolbox 1-6  
    Frequency-Domain System Identification Tool-  
        box 1-6  
    Fuzzy Logic Toolbox 1-6  
    Higher-Order Spectral Analysis Toolbox 1-6  
    Image Processing Toolbox 1-6  
    LMI Control Toolbox 1-7  
    Model Predictive Control Toolbox 1-7  
    Mu-Analysis and Synthesis Toolbox 1-7

NAG Foundation Toolbox 1-7  
Neural Network Toolbox 1-7  
Optimization Toolbox 1-7  
Partial Differential Equation Toolbox 1-8  
QFT Control Design Toolbox 1-8  
Robust Control Toolbox 1-8  
Signal Processing Toolbox 1-8  
Spline Toolbox 1-8  
Statistics Toolbox 1-9  
Symbolic Math Toolbox 1-9  
System Identification Toolbox 1-9  
Wavelet Toolbox 1-9  
trace debug command 11-47  
tracing facilities 4-34  
Transfer Fcn block 8-203  
    algebraic loops 9-7  
    example 3-60  
    linearization 5-5  
transfer function form, converting to 5-12  
transfer functions  
    discrete 8-65  
    linear 8-203  
    masked block icons 6-21  
    poles and zeros 8-222  
    poles and zeros, discrete 8-67  
transparent icon 6-24  
Transport Delay block 8-206  
    linearization 5-5  
Trapezoidal method 8-59  
trigger  
    control signal, outputting 7-10  
    events 7-2, 7-8  
    falling 7-9  
    input 7-8  
    rising 7-8, 7-9  
    type parameter 7-9  
Trigger block 8-208

- creating triggered subsystem 7-9
- outputting trigger signal 7-10
- showing output port 7-10
- trigger type
  - either 7-9
- triggered and enabled subsystems 7-2, 7-11
- triggered subsystems 7-2, 7-8, 8-208
- Trigonometric Function block 8-210
- trim function 5-7, 5-13, 8-100
- truth tables, implementing 8-25
- tunable parameters 6-14

**U**

- unconnected input ports 8-95
- unconnected output ports, capping 8-196
- undisp debug command 11-48
- Undo menu item 3-7
- UndoDeleteFcn block callback parameter 3-56
- Uniform Random Number block 8-212
- uniformly distributed random numbers 8-212
- Unit Delay block 8-214
  - compared to Transport Delay block 8-206
- Unmask button on Mask Editor 6-9
- unresolved link 3-22
- unstable simulation results 4-28
- untrace debug command 11-49
- Update Diagram menu item 3-18, 3-23, 3-34, 9-16, 10-24
- updating linked blocks 3-23
- updating states 9-13
- URL specification in block help 6-27
- user
  - specifying current 3-70

**V**

- variable time delay 8-216
- Variable Transport Delay block 8-216
- variable-step solvers 4-9, 4-11
- vdp model
  - initial conditions 4-23
  - using Scope block 8-165
- vector length, checking 9-2
- vector signals
  - displaying 8-164
  - generating from inputs 8-136
  - separating 8-45
- vectorization of blocks 3-18
- version control model parameters 3-77
- vertices, moving 3-31
- viewing output trajectories 5-2
- virtual blocks 3-9
- viscous friction 8-35
- visibility of Goto tag 8-94
- visible icon frame 6-23

**W**

- Wavelet Toolbox 1-9
- web command and masked block help 6-27
- white noise, generating 8-18
- Wide Vector Lines menu item 3-18
- Width block 8-218
- workspace
  - destination 4-33
  - loading from 4-17
  - mask 6-5, 6-15
  - reading data from 8-85
  - saving to 4-20
  - source 4-34
  - writing output to 8-199
  - writing to 4-5

Workspace I/O page of Simulation Parameters dialog box 4-17

writing

data to data store 8-39

output to file 8-197

output to workspace 8-199

## **X**

xbreak debug command 11-50

XY Graph block 8-219

## **Z**

zcbreak debug command 11-51

zclist debug command 11-52

zero crossings 9-3-9-7

detecting 4-35, 8-96

disabling detection of 4-25

zero output in region, generating 8-43

zero-crossing slope method 7-3

Zero-Order Hold block 8-214, 8-221

compared to First-Order Hold block 8-78, 8-88

Zero-Pole block 8-222

algebraic loops 9-7

zero-pole form, converting to 5-12

Zooming block diagrams 3-6

zooming in on displayed data 8-166