SIMULINK®

Dynamic System Simulation for MATLAB®

Modeling

Simulation

Implementation

The
MATH
WORKS
Inc.

Using Simulink

Version 4

How to Contact The MathWorks:

☎	508-647-7000	Phone
	508-647-7001	Fax
✉	The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
🖥	http://www.mathworks.com	Web
	ftp.mathworks.com	Anonymous FTP server
	comp.soft-sys.matlab	Newsgroup
@	support@mathworks.com	Technical support
	suggest@mathworks.com	Product enhancement suggestions
	bugs@mathworks.com	Bug reports
	doc@mathworks.com	Documentation error reports
	subscribe@mathworks.com	Subscribing user registration
	service@mathworks.com	Order status, license renewals, passcodes
	info@mathworks.com	Sales, pricing, and general information

Using Simulink

Contents

Running a Simulation

5

Analyzing Simulation Results

6

Using Masks to Customize Blocks

7

Conditionally Executed Subsystems

8

Block Reference

9

Model Construction Commands

10

Simulink Debugger

11

Performance Tools

12

Model and Block Parameters

A

B   Model File Format

Getting Started

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.

For a technical introduction to Simulink, see Chapter 3, "How Simulink Works." This chapter introduces many key concepts that you will need to understand how to create and run Simulink models.

Chapter 4, "Creating a Model" 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 5, "Running a Simulation" 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 6, "Analyzing Simulation Results" discusses Simulink and MATLAB features useful for viewing and analyzing simulation results.

Chapter 7, "Using Masks to Customize Blocks" discusses methods for creating your own blocks and using masks to customize their appearance and use.

Chapter 8, "Conditionally Executed Subsystems" describes subsystems whose execution depends on triggering signals.

Chapter 9, "Block Reference" provides reference information for all Simulink blocks.

Chapter 10, "Model Construction Commands" 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, "Simulink Debugger" explains how to use the Simulink debugger to debug Simulink models. It also documents debugger commands.

Chapter 12, "Performance Tools" explains how to use the Simulink accelerator and other optional tools that improve the performance of Simulink models.

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

Appendix B, "Model File Format" 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" in the Release Notes delivered with your Simulink system.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Simulink.

For more information about any of these products, see either

- The online documentation for that product, if it is loaded or if you are reading the documentation from the CD
- The MathWorks Web site, at www.mathworks.com; see the "products" section

See our Web page www.mathworks.com for the latest update on new products and capabilities. Also see the connections site www.mathworks.com/products/connections/ for third-party products compatible with Simulink.

Note The toolboxes listed below all include functions that extend the MATLAB environment. The blocksets all include blocks that extend the Simulink environment.

Product	Description
μ-Analysis and Synthesis Toolbox	Tools for robust control design using optimal control and the structured singular value
CDMA Reference Blockset	Simulink block libraries for the design and simulation of the IS-95A wireless communications standard
Communications Blockset	Simulink block libraries for modeling the physical layer of communications systems
Communications Toolbox	MATLAB functions for modeling the physical layer of communications systems
Control System Toolbox	An interactive environment for classical and modern control system design, analysis, and modeling

Product	Description
Dials & Gauges Blockset	Graphical instrumentation for monitoring and controlling signals and parameters in Simulink models
DSP Blockset	Simulink block libraries for the design, simulation, and prototyping of digital signal processing systems
Fixed-Point Blockset	Simulink blocks that model, simulate, and automatically generate pure integer code for fixed-point applications
Frequency Domain System Identification Toolbox	Tools for frequency domain model identification and validation
Motorola DSP Developer's Kit	Provides an object-oriented interface to program MEX-files or S-functions that call the appropriate Motorola Suite56^{TM} DSP Simulator.
Nonlinear Control Design (NCD) Blockset	Simulink block libraries that provide a time-domain-based optimization approach to system design; automatically tunes parameters based on user-defined time-domain performance constraints
Power System Blockset	Simulink block libraries for the design, simulation, and prototyping of electrical power systems
Real-Time Windows Target	Tool that allows you to run Simulink models interactively and in real time on your PC
Real-Time Workshop®	Tools that generate customizable code from Simulink and Stateflow models for targeting real-time systems or speeding up simulations.

Product	Description
Real-Time Workshop Ada Coder	Tool that allows you to automatically generate Ada 95 code. It produces the code directly from Simulink models and automatically builds programs that can be run in real time in a variety of environments.
Real-Time Workshop Production Coder	Add on component for generating embeddable production quality code from Simulink models. Included are utilities and capabilities to verify generated code in a co-simulation and code generation interfacing options.
Requirements Management Interface	This interface helps you coordinate, track, and implement changes in design specifications (requirements) throughout the development cycle.
Robust Control Toolbox	Tools for advanced robust multivariable feedback control
Signal Processing Toolbox	Tools for algorithm development, signal and linear system analysis, and time-series data modeling
Simulink Performance Tools	Includes tools for comparing models and profiling and accelerating the performance of simulations.
Simulink Report Generator	Tool for documenting information in MATLAB, Simulink, and Stateflow in multiple output formats
Stateflow	Tool for graphical modeling and simulation of complex reactive systems
Stateflow Coder	Tool for generating highly readable, efficient C code from Stateflow diagrams

Product	Description
System Identification Toolbox	An interactive environment for building accurate, simplified models of complex systems from noisy time-series data
Developer's Kit for Texas Instruments DSP	Lets you generate, target, and execute Simulink models on the Texas Instruments (TI) C6701 Evaluation Module (C6701 EVM).
xPC Target	Tools for adding I/O blocks to Simulink block diagrams, generating code with Real-Time Workshop, and downloading the code to a second PC that runs the xPC Target real-time kernel. The xPC Target is ideal for rapid prototyping and hardware-in-the-loop testing of control and DSP systems.

2

Quick Start

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.
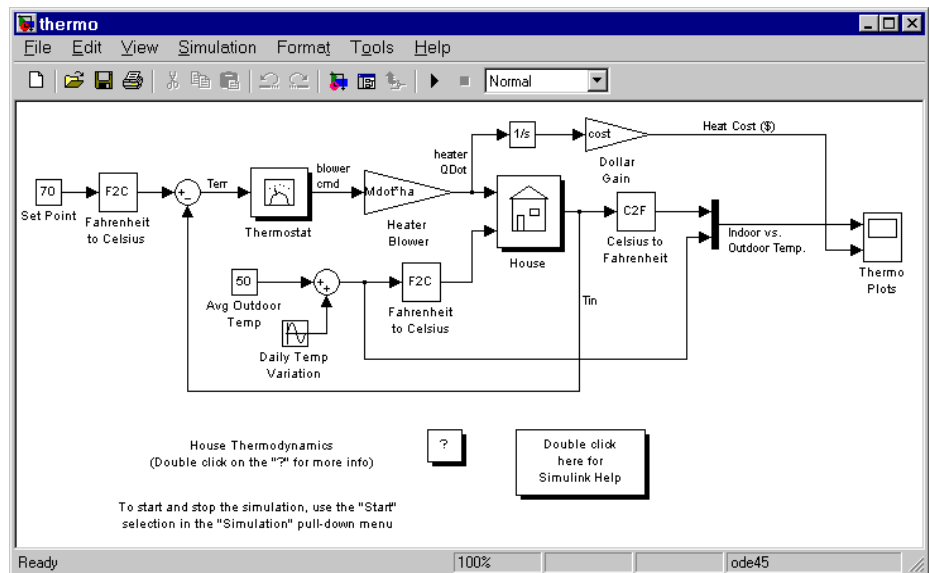
3 Double-click the Scope block labeled Thermo Plots.

The Scope block displays two plots labeled Indoor vs. Outdoor Temp and Heat Cost ($), respectively.

4 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.

5 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.

6 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.
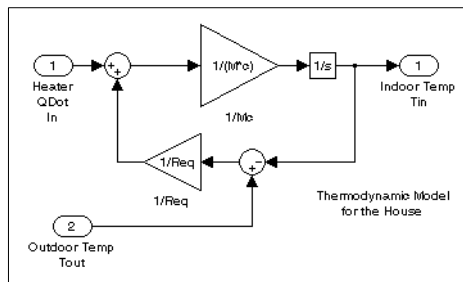
House subsystem

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.
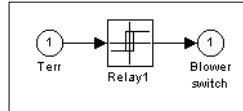
Thermostat subsystem

Both the outside and inside temperatures are converted from Fahrenheit to Celsius by identical subsystems.
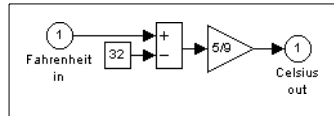
Fahrenheit to Celsius conversion (F2C)

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 Scope on page 9-206.

- 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. 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 "Running a Simulation Using Menu Commands" on page 5-4.
- You can encapsulate complex groups of related blocks in a single block, called a subsystem. See "Creating Subsystems" on page 4-65 for more information.
- You can create a customized icon and design a dialog box for a block by using the masking feature, described in detail in Chapter 7, "Using Masks to Customize Blocks." 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. See Scope on page 9-206 for more information.

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.
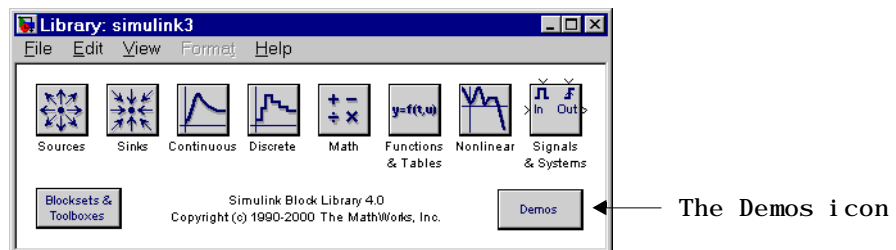
The Demos icon

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.
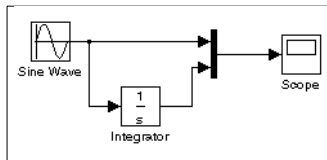
To create the model, first type simulink in the MATLAB command window. On Microsoft Windows, the Simulink Library Browser appears.
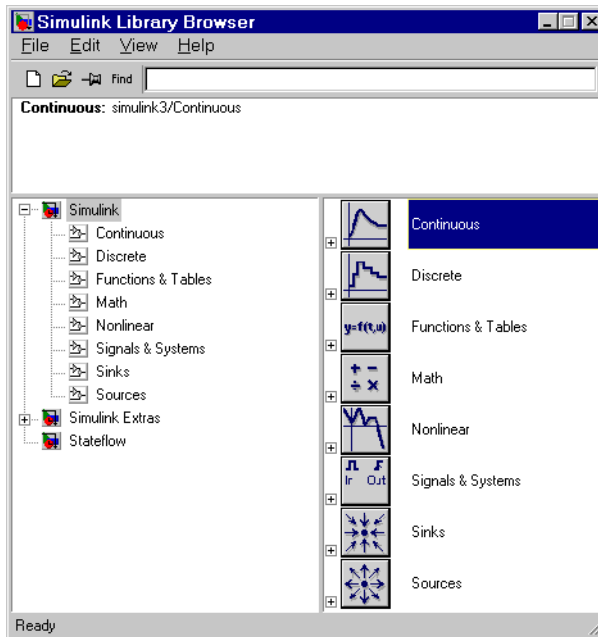
On UNIX, the Simulink library window appears.
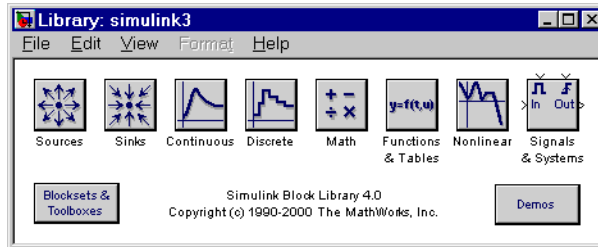
To create a new model on UNIX, select **Model** from the **New** submenu of the Simulink library window's **File** menu. To create a new model on Windows, select the **New Model** button on the Library Browser's toolbar.

New Model button
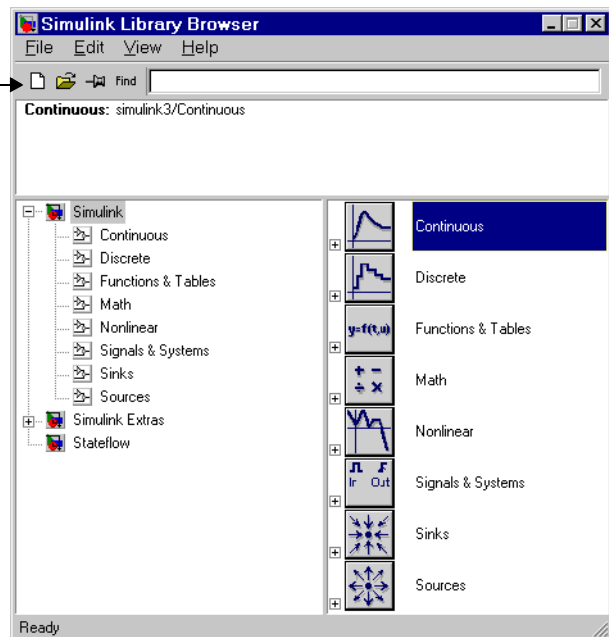
Simulink opens a new model window.
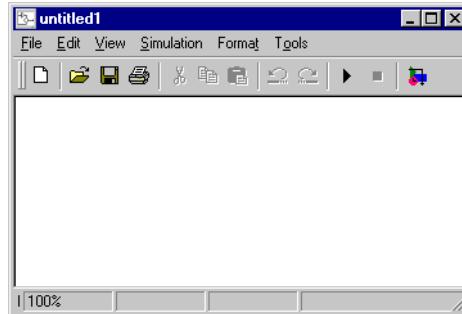
To create this model, you will need to copy blocks into the model from the following Simulink block libraries:

- 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 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.

Simulink library

Sources library
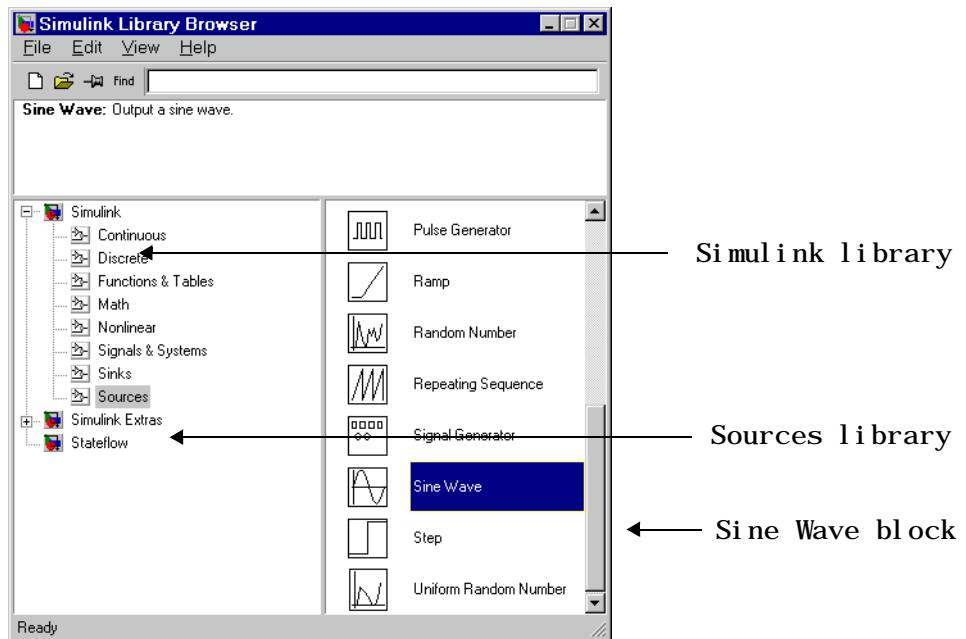
Sine Wave block

Now drag the Sine Wave block 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.
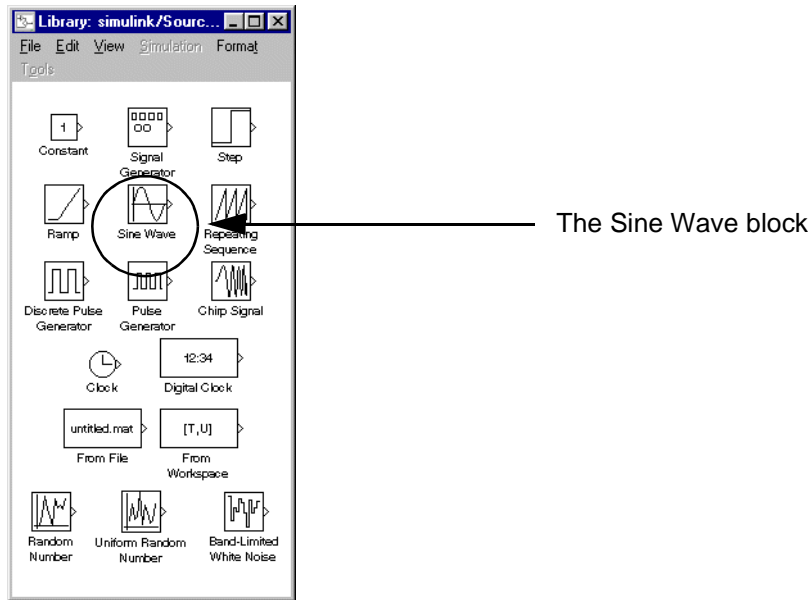
The Sine Wave block

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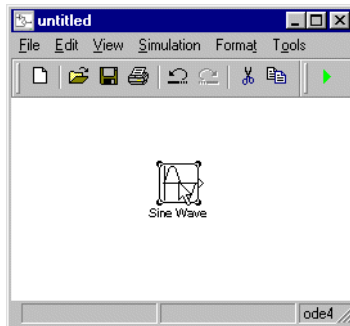
Copy the rest of the blocks in a similar manner from their respective libraries into the model window. You can move a block from one place in the model window to another by dragging the block. You can move a block a short distance by selecting the block, then pressing the arrow keys.

With all the blocks copied into the model window, the model should look something like this.
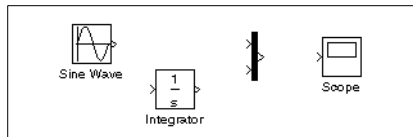
If you examine the block icons, you see an angle bracket on the right of the Sine Wave block and two on the left of the Mux block. The > symbol pointing out of a block is an *output port*; if the symbol points to a block, it is an *input port*. A signal travels out of an output port and into an input port of another block through a connecting line. When the blocks are connected, the port symbols disappear.
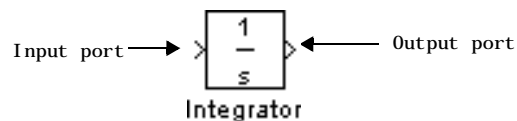
Now it's time to connect the blocks. Connect the Sine Wave block to the top input port of the Mux block. Position the pointer over the output port on the right side of the Sine Wave block. Notice that the cursor shape changes to cross hairs.
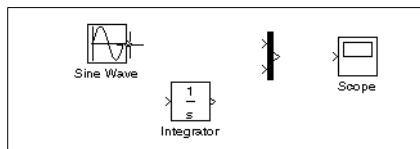
Hold down the mouse button and move the cursor to the top input port of the Mux block. Notice that the line is dashed while the mouse button is down and that the cursor shape changes to double-lined cross hairs as it approaches the Mux block.
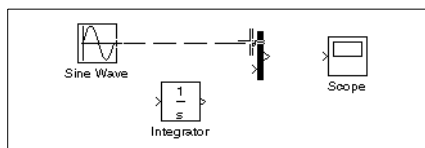
Now release the mouse button. The blocks are connected. You can also connect the line to the block by releasing the mouse button while the pointer is inside the icon. If you do, the line is connected to the input port closest to the cursor's position.
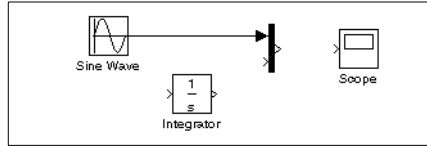
If you look again at the model at the beginning of this section (see "Building a Simple Model" on page 2-6), you'll notice that most of the lines connect output ports of blocks to input ports of other blocks. However, one line connects a *line* to the input port of another block. This line, called a *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.
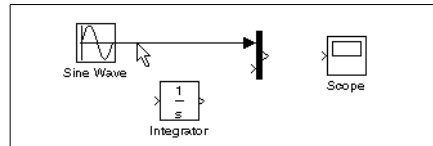
2 Press and hold down the **Ctrl** key (or click the right mouse button). Press the mouse button, then drag the pointer to the Integrator block's input port or over the Integrator block itself.
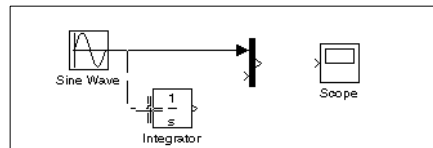
3 Release the mouse button. Simulink draws a line between the starting point and the Integrator block's input port.
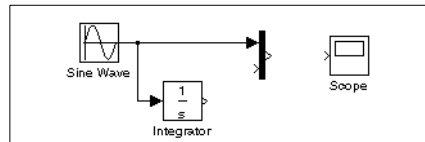
Finish making block connections. When you're done, your model should look something like this.
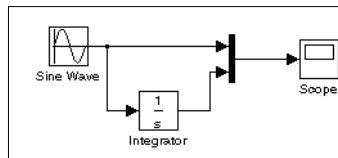
Now, open the Scope block to view the simulation output. Keeping the Scope window open, set up Simulink to run the simulation for 10 seconds. First, set the simulation parameters by choosing **Simulation Parameters** from the **Simulation** menu. On the dialog box that appears, notice that the **Stop time** is set to 10.0 (its default value).

Stop time parameter
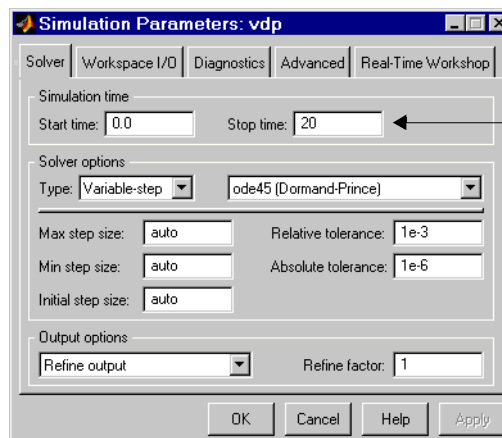
Close the **Simulation Parameters** dialog box by clicking on the **OK** button. Simulink applies the parameters and closes the dialog box.

2-13

Choose **Start** from the **Simulation** menu and watch the traces of the Scope block's input.
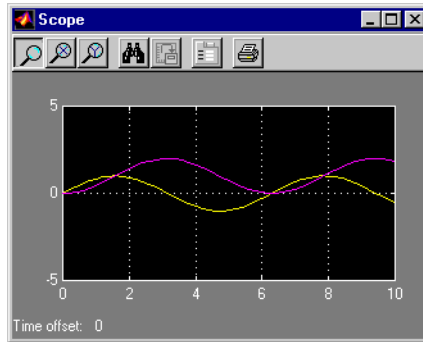
The simulation stops when it reaches the stop time specified in the **Simulation Parameters** dialog box or when you choose **Stop** from the **Simulation** menu.

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 4, "Creating a Model."

Setting Simulink Preferences

The MATLAB **Preferences** dialog box allows you to specify default settings for many Simulink options. To display the **Preferences** dialog box, select **Preferences** from the Simulink **File** menu.
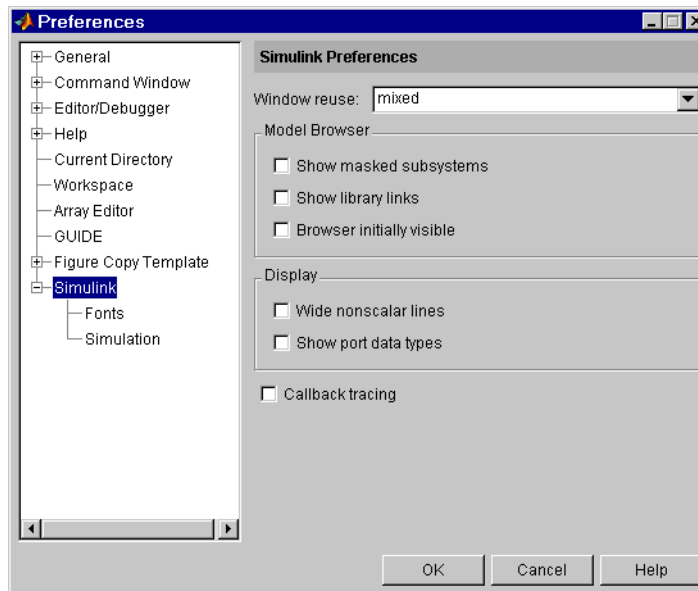
Simulink Preferences

The **Preferences** dialog box allows you to specify the following Simulink preferences.

Window reuse

Specifies whether Simulink uses existing windows or opens new windows to display a model's subsysems (see "Window Reuse" on page 4-67).

Model Browser

Specifies whether Simulink displays the browser when you open a model and whether the browser shows blocks imported from subsystems and the contents of masked subsystems (see "The Model Browser" on page 4-99).

Display

Specifies whether to use thick lines to display nonscalar connections between blocks and whether to display port data types on the block diagram (see "Setting Signal Display Options" on page 4-37).

Callback tracing

Specifies whether to display the model callbacks that Simulink invokes when simulating a model (see "Using Callback Routines" on page 4-70).

Simulink Fonts

Specifies fonts to be used for block and line labels and diagram annotations.

Solver

Specifies simulation solver options (see "The Solver Pane" on page 5-8).

Workspace

Specifies workspace options for simulating a model (see "The Workspace I/O Pane" on page 5-18).

Diagnostics

Specifies diagnostic options for simulating a model (see "The Diagnostics Pane" on page 5-26).

3

How Simulink Works

What Is Simulink

Simulink is a software package that enables you to model, simulate, and analyze dynamic systems, that is, systems whose outputs and states change with time. Simulink can be used to explore the behavior of a wide range of real-world systems, including electrical circuits, shock absorbers, braking systems, and many other electrical, mechanical, and thermodynamic systems.

Simulating a dynamic system is a two-step process with Simulink. First, you use Simulink's model editor to create a model of the system to be simulated. The model graphically depicts the time-dependent mathematical relationships among the system's inputs, states, and outputs (see "Modeling Dynamic Systems" on page 3-3). Then, you use Simulink to simulate the behavior of the system for a specified time span. Simulink uses information that you entered into the model to perform the simulation (see "Simulating Dynamic Systems" on page 3-9).

Modeling Dynamic Systems

Simulink provides a library browser that allows you to select blocks from libraries of standard blocks (see Chapter 9, "Block Reference") and a graphical editor that allows you to draw lines connecting the blocks (see Chapter 4, "Creating a Model"). You can model virtually any real-world dynamic system by selecting and interconnecting the appropriate Simulink blocks.

Block Diagrams

A Simulink block diagram is a pictorial model of a dynamic system. It consists of a set of symbols, called blocks, interconnected by lines. Each block represents an elementary dynamic system that produces an ouput either continuously (a continuous block) or at specific points in time (a discrete block). The lines represent connections of block inputs to block outputs. Every block in a block diagram is an instance of a specific type of block. The type of the block determines the relationship between a block's outputs and its inputs, states, and time. A block diagram can contain any number of instances of any type of block needed to model a system.

Note The MATLAB Based Books page on the MathWorks Web site includes texts that discuss the use of block diagrams in general, and Simulink in particular, to model dynamic systems.

Blocks

Blocks represent elementary dynamic systems that Simulink knows how to simulate. A block comprises one or more of the following: a set of inputs, a set of states, and a set of outputs.
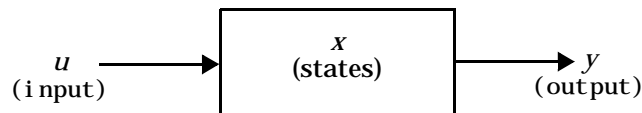
A block's output is a function of time and the block's inputs and states (if any). The specific function that relates a block's output to its inputs, states, and time depends on the type of block of which the block is an instance.

3-3

States

Blocks can have states. A *state* is a variable that determines a block's output and whose current value is a function of the previous values of the block's states and/or inputs. A block that has a state must store previous values of the state to compute its current state. States are thus said to be persistent. Blocks with states are said to have memory because such blocks must store the previous values of their states and/or inputs in order to compute the current values of the states.

The Simulink Integrator block is an example of a block that has a state. The Integrator block outputs the integral of the input signal from the start of the simulation to the current time. The integral at the current time step depends on the history of Integrator block's input. The integral therefore is a state of the Integrator block and is, in fact, its only state. Another example of a block with states is the Simulink Memory block. A Memory block stores the values of its inputs at the current simulation time and outputs them at a later time. The states of a Memory block are the previous values of its inputs.

The Simulink Gain block is an example of a stateless block. A Gain block outputs its input signal multiplied by a constant called the gain. The output of a Gain block is determined entirely by the current value of the input and the gain, which does not vary. A Gain block therefore has no states. Other examples of stateless blocks include the Sum and Product blocks. The output of these blocks is purely a function of the current values of their inputs (the sum in one case, the product in the other). Thus, these blocks have no states.

System Functions

Each Simulink block type is associated with a set of system functions that specify the time-dependent relationships among its inputs, states, and outputs. The system functions include:

- An output function, f_o, that relates the system's outputs to its inputs, states, and time

- An update function, f_u, that relates the future values of the system's discrete states to the current time, inputs, and states

- A derivative function, f_d, that relates the derivatives of the system's continuous states to time and the present values of the block's states and inputs

Symbolically, the system functions may be expressed as follows

$$y = f_o(t, x, u) \qquad \text{Output function}$$

$$x_{d_{k+1}} = f_u(t, x, u) \qquad \text{Update function}$$

$$x'_c = f_d(t, x, u) \qquad \text{Derivative function}$$

$$\text{where} \quad x = \begin{bmatrix} x_c \\ x_{d_k} \end{bmatrix}$$

where t is the current time, x is the block's states, u is the block's inputs, y is the block's outputs, x_d is the block's discrete derivatives, and x'_c is the derivatives of the block's continous states. During a simulation, Simulink invokes the system functions to compute the values of the system's states and outputs.

Block Parameters

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

A parameterized block effectively represents a family of similar blocks. For example, when creating a model, you can set the gain parameter of each instance of the Gain block separately so that each instance behaves differently. Because it allows each standard block to represent a family of blocks, block parameterization greatly increases the modeling power of Simulink's standard libraries.

Tunable Parameters

Many block parameters are tunable. A *tunable parameter* is a parameter whose value can change while Simulink is executing a model. For example, the gain parameter of the Gain block is tunable. You can alter the block's gain while a simulation is running. If a parameter is not tunable and the simulation is running, Simulink disables the dialog box control that sets the parameter. Simulink allows you to specify that all parameters are nontunable in your

model, except for those that you specify. This can speed up execution of large models and enable generation of faster code from your model. See "Model parameter configuration" on page 5–30 for more information.

Continuous Versus Discrete Blocks

Simulink's standard block set includes continuous blocks and discrete blocks. Continuous blocks respond continuously to continuously changing input. Discrete blocks, by contrast, respond to changes in input only at integral multiples of a fixed interval called the block's sample time. Discrete blocks hold their output constant between successive sample time hits. Each discrete block includes a sample time parameter that allows you to specify its sample rate. Examples of continuous blocks include the Constant block and the blocks in Simulink's Continuous block library. Examples of discrete blocks include the Discrete Pulse Generator and the blocks in the Discrete block library.

Many Simulink blocks, for example, the Gain block, can be either continuous or discrete, depending on whether they are driven by continuous or discrete blocks. A block that can be either discrete or continuous is said to have an implicit sample rate. The implicit sample time is continuous if any of the block's inputs are continuous. The implicit sample time is equal to the shortest input sample time if all the input sample times are integral multiples of the shortest time. Otherwise, the input sample time is equal to the *fundamental sample time* of the inputs, where the fundamental sample time of a set of sample times is defined as the greatest integer divisor of the set of sample times.

Simulink can optionally color code a block diagram to indicate the sample times of the blocks it contains, e.g., black (continous), magenta (constant), yellow (hybrid), red (fastest discrete), and so on. See "Mixed Continuous and Discrete Systems" on page 3-28 for more information.

Subsystems

Simulink allows you to model a complex system as a set of interconnected subsystems each of which is represented by a block diagram. You create a subsystem using Simulink's Subsystem block and the Simulink model editor. You can embed subsystems with subsystems to any depth to create hierarchical models. You can create conditionally executed subsystems that are executed only when a transition occurs on a triggering or enabling input (see Chapter 8, "Conditionally Executed Subsystems.").

Custom Blocks

Simulink allows you to create libraries of custom blocks that you can then use in your models. You can create a custom block either graphically or programmatically. To create a custom block graphically, you draw a block diagram representing the block's behavior, wrap this diagram in an instance of Simulink's Subsystem block, and provide the block with a parameter dialog, using Simulink's block mask facility. To create a block programmatically, you create an M-file or a MEX-file that contains the block's system functions (see *Writing S-Functions*). The resulting file is called an S-function. You then associate the S-function with instances of Simulink's S-function block in your model. You can add a parameter dialog to your S-function block by wrapping it in a Subsystem block and adding the parameter dialog to the Subsystem block.

Signals

Simulink uses the term *signal* to refer to the output values of blocks. Simulink allows you to specify a wide range of signal attributes, including signal name, data type (e.g., 8-bit, 16-bit, or 32-bit integer), numeric type (real or complex), and dimensionality (one-dimensional or two-dimensional array). Many blocks can accept or output signals of any data or numeric type and dimensionality. Others impose restrictions on the attributes of the signals they can handle.

Data Types

The term data type refers to the internal representation of data on a computer system. Simulink can handle parameters and signals of any built-in data type supported by MATLAB, such as `int8`, `double`, and `boolean` (see "Working with Data Types" on page 4-44). Further, Simulink defines two Simulink-specific data types:

- `Simulink.Parameter`
- `Simulink.Signal`

These Simulink-specific data types capture Simulink-specific information that is not captured by general-purpose numeric types, such as int32. Simulink allows you to create and use instances of Simulink data types, called *data objects*, as parameters and signals in Simulink models.

You can extend both Simulink data types to create data types that capture information specific to your models.

> **Note** The Simulink user interface and documentation also refers to the Simulink data types as classes to distinguish them from nonextensible data types, such as the built-in MATLAB types.

Solvers

A Simulink model specifies the time derivatives of its continuous states but not the values of the states themselves. Thus, when simulating a sytem, Simulink must compute continous states by numerically integrating their state derivatives. A variety of general-purpose numerical integration techniques exist, each having advantages in specific applications. Simulink provides implementations, called ordinary differential equation (ODE) solvers, of the most stable, efficient, and accurate of these numerical integration methods. You can specify the solver to use in the model or when running a simulation.

Simulating Dynamic Systems

Simulating a dynamic system refers to the process of computing a system's states and outputs over a span of time, using information provided by the system's model. Simulink simulates a system when you choose **Start** from the model editor's **Simulation** menu, with the system's model open.

Simulation of the system occurs in two phases: model initialization and model execution.

Model Initialization Phase

During the initialization phase, Simulink:

1 Evaluates the model's block parameter expressions to determine their values.

2 Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain (see "Atomic Versus Virtual Subsystems" on page 3-13).

3 Sorts the blocks into the order in which they need to be executed during the execution phase (see "Determining Block Update Order" on page 3-11).

4 Determines signal attributes, e.g., name, data type, numeric type, and dimesionality, not explicitly specified by the model and checks that each block can accept the signals connected to its inputs.

Simulink uses a process called attribute propagation to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives.

5 Determines the sample times of all blocks in the model whose sample times you did not explicitly specify.

6 Allocates and initializes memory used to store the current values of each block's states and outputs.

Model Execution Phase

The simulation now enters the model execution phase. In this phase, Simulink successively computes the states and outputs of the system at intervals from

the simulation start time to the finish time, using information provided by the model. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called the step size. The step size depends on the type of solver (see "Solvers" on page 3-13) used to compute the system's continuous states, the system's fundamental sample time (see "Modeling and Simulating Discrete Systems" on page 3-23), and whether the system's continuous states have discontinuities ("Zero Crossing Detection" on page 3-14).

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

Processing at Each Time Step

At each time step, Simulink

1 Updates the outputs of the models' blocks in sorted order (see "Determining Block Update Order" on page 3-11).

Simulink computes a block's outputs by invoking the block's output function. Simulink passes the current time and the block's inputs and states to the output function as it may require these arguments to compute the block's output. Simulink updates the output of a discrete block only if the current step is an integral multiple of the block's sample time.

2 Updates the states of the model's blocks in sorted order.

Simulink computes a block's discrete states by invoking its discrete state update function. Simulink computes a block's continuous states by numerically integrating the time derivatives of the continuous states. It computes the time derivatives of the states by invoking the block's continuous derivatives function.

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

Simulink uses a technique called zero crossing detection to detect
discontinuities in continuous states. See "Zero Crossing Detection" on
page 3-14 for more information.

4 Computes the time for the next time step.

Simulink repeats steps 1 through 4 until the simulation stop time is reached.

Determining Block Update Order

During a simulation, Simulink updates the states and outputs of a model's
blocks once per time step. The order in which the blocks are updated is
therefore critical to the validity of the results. In particular, if a block's outputs
are a function of its inputs at the current time step, the block must be updated
after the blocks that drive its inputs. Otherwise, the block's outputs will be
invalid. The order in which blocks are stored in a model file is not necessarily
the order in which they need to be updated during a simulation. Consequently,
Simulink sorts the blocks into the correct order during the model initialization
phase.

Direct Feedthrough Blocks

In order to create a valid update ordering, Simulink categorizes blocks
according to the relationship of outputs to inputs. Blocks whose current
outputs depend on their current inputs are called *direct feedthrough* blocks. All
other blocks are called nondirect-feedthrough blocks. Examples of
direct-feedthrough blocks include the Gain, Product, and Sum blocks.
Examples of nondirect-feedthrough blocks include the Integrator block (its
output is a function purely of its state), the Constant block (it does not have an
input), and the Memory block (its output is dependent on its input in the
previous time step).

Block Sorting Rules

Simulink uses the following basic update rules to sort the blocks:

- Each block must be updated before any of the direct-feedthrough blocks that it drives.

  This rule ensures that the inputs to direct-feedthrough blocks will be valid when they are updated.
- Nondirect-feedthrough blocks can be updated in any order as long as they are updated before any direct-feedthrough blocks that they drive.

  This rule can be met by putting all nondirect-feedthrough blocks at the head of the update list in any order. It thus allows Simulink to ignore nondirect-feedthrough blocks during the sorting process.

The result of applying these rules is an update list in which nondirect-feedthrough blocks appear at the head of the list in no particular order followed by direct-feedthrough blocks in the order required to supply valid inputs to the blocks they drive.

During the sorting process, Simulink checks for and flags the occurrence of algebraic loops, that is, signal loops in which an output of a direct-feedthrough block is connected directly or indirectly to one of the block's inputs. Such loops seemingly create a deadlock condition since Simulink needs the input of a direct-feedthrough block in order to compute its output. However, an algebraic loop can represent a set of simultaneous algebraic equations (hence the name) where the block's input and output are the unknowns. Further, these equations can have valid solutions at each time step. Accordingly, Simulink assumes that loops involving direct-feedthrough blocks do, in fact, represent a solvable set of algebraic equations and attempts to solve them each time the block is updated during a simulation. For more information, see "Algebraic Loops" on page 3-18.

Block Priorities

Simulink allows you to assign update priorities to blocks (see "Assigning Block Priorities" on page 4-18). Simulink updates higher priority blocks before lower priority blocks. Simulink honors the priorities only if they are consistent with its block sorting rules.

Atomic Versus Virtual Subsystems

Subsystems can be virtual or atomic. Simulink ignores virtual subsystem boundaries when determining block update order. By contrast, Simulink executes all blocks within an atomic subsystem before moving onto the next block. Conditionally executed subsystems are atomic. Unconditionally executed subsystems are virtual by default. You can, however, designate an unconditionally executed subsystem as atomic (see Subsystem). This is useful if you need to ensure that a subsystem is executed in its entirety before any other block is executed.

Solvers

Simulink computes the current value of a block's continuous states by numerically integrating the state's derivatives. The numerical integration task is performed by a Simulink component called a solver. Simulink allows you to choose the solver that it uses to simulate a model. The solvers that Simulink provides fall into two classes: fixed-step solvers and variable-step solvers.

Fixed-Step Solvers

Fixed-step solvers divide the simulation timespan up into an integral number of fixed-size intervals called time steps. Then, starting from initial estimates, at each time step, a fixed-step solver computes the value of each of the system's state variables at the next time step from the variable's current value and the current value of its derivatives. The accuracy of the estimation depends on the *step size*, that is, the time between successive time steps. Generally, a smaller step size produces a more accurate simulation but results in a longer execution time because more steps are required to compute a system's states.

Variable Step Solvers

A variable step solver dynamically varies the step size to meet a specified level of precision. Such a solver expands the step size when the state variables are changing slowly (as indicated by the magnitude of the state derivatives) and decreases the step size when the state variables are changing rapidly. A variable step solver can, depending on the application, produce more accurate results without sacrificing execution speed.

Major Versus Minor Steps

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

solver produces a result at each major time step. It use results at the minor time steps to improve the accuracy of the result at the major time step.

Zero Crossing Detection

When simulating a dynamic system, Simulink checks for discontinuities in the system's state variables at each time step, using a technique known as zero crossing detection. If Simulink detects a discontinuity within the current time step, it determines the precise time at which the discontinuity occurs and takes additional time steps before and after the discontinuity. This section explains why zero crossing detection is important and how it works.

Discontinuities in state variables often coincide with significant events in the evolution of a dynamic system. For example, the instant when a bouncing ball hits the floor coincides with a discontinuty in its position. Because discontinuties often indicate a significant change in a dynamic system, it is important to simulate points of discontinuity precisely. Otherwise, a simulation could lead to false conclusions about the behavior of the system under investigation. Consider, for example, a simulation of a bouncing ball. If the point at which the ball hits the floor occurs between simulation steps, the simulated ball appears to reverse position in midair. This might lead an investigator to false conclusions about the physics of the bouncing ball.

To avoid such misleading conclusions, it is important that simulation steps occur at points of discontinuity. A simulator that relies purely on solvers to determine simulation times cannot efficiently meet this requirement. Consider, for example, a fixed-step solver. A fixed-step solver computes the values of state variables at integral multiples of a fixed step size. However, there is no guarantee that a point of discontinuity will occur at an integral multiple of the step size. You could reduce the step size to increase the probability of hitting a discontinuity, but this would greatly increase the execution time.

A variable step solver appears to offer a solution. A variable step solver adjusts the step size dynamically, increasing the step size when a variable is changing slowly and decreasing the step size when the variable changes rapidly. Around a discontinuity, a variable changes extremely rapidly. Thus, in theory, a variable step solver should be able to hit a discontinuity precisely. The problem is that to locate a discontinuity accurately, a variable step solver must again take many small steps, greatly slowing down the simulation.

How Zero Crossing Detection Works

Simulink uses a technique known as zero crossing detection to address this problem. With this technique, a block can register a set of zero crossing variables with Simulink, each of which is a function of a state variable that can have a discontinuity. The zero crossing function passes through zero from a positive or negative value when the corresponding discontinuity occurs. At the end of each simulation step, Simulink asks each block that has registered zero crossing variables to update the variables. Simulink then checks whether any variable has changed sign since the last step. Such a change indicates that a discontinuity occurred in the current time step.

If any zero crossings are detected, Simulink interpolates between the previous and current values of each variable that changed sign to estimate the times of the zero crossings (e.g., discontinuities). Simulink then steps up to and over each zero crossing in turn. In this way, Simulink avoids simulating exactly at the discontinuity where the value of the state variable may be undefined.

zero crossing detection enables Simulink to simulate discontinuities accurately without resorting to excessively small step sizes. Many Simulink blocks support zero crossing detection. The result is fast and accurate simulation of all systems, including systems with discontinuities.

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 3-17 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.
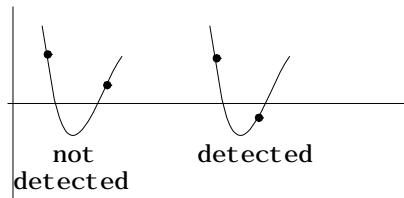
If you suspect this is happening, tighten the error tolerances to ensure that the solver takes small enough steps. For more information, see "Error Tolerances" on page 5–13.

Note Using the Refine option (see "Refine output" on page 5-16) will not help locate the missed zero crossings. You should alter the maximum step size or output times.

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** option on the **Advanced** pane of the **Simulation Parameters** dialog box (see "Zero-crossing detection" on page 5-32). 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

Block	Description of Zero Crossing
Abs	One: to detect when the input signal crosses zero in either the rising or falling direction.
Backlash	Two: one to detect when the upper threshold is engaged, and one to detect when the lower threshold is engaged.
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 **Disable zero crossing detection** option in the **Advanced** pane of the **Simulation Parameters** 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

Block	Description of Zero Crossing (Continued)
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.
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 9, "Block Reference."

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 3-20 for an example of an exception to this general rule.) An example of an algebraic loop is this simple scalar loop.
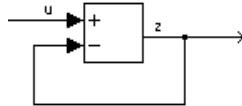
Mathematically, this loop implies that the output of the Sum block is an algebraic state z constrained to equal the first input u minus z (i.e. $z = u - z$). The solution of this simple loop is $z = u/2$, but most algebraic loops cannot be solved by inspection. It is easy to create vector algebraic loops with multiple algebraic state variables $z1$, $z2$, etc., as shown in this model.
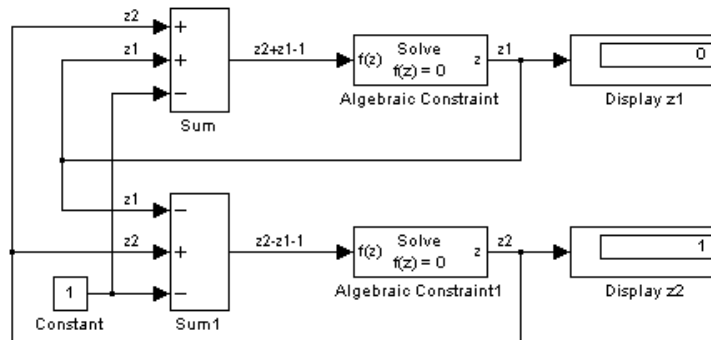
The Algebraic Constraint block is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal $F(z)$ to zero and outputs an algebraic state z. This block outputs the value necessary to produce a zero at the input. The output must 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

$z2 + z1 – 1 = 0$
$z2 – z1 – 1 = 0$

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

A triggered subsystem holds its outputs constant between trigger events (see "Triggered Subsystems" on page 8-8). Thus, a solver can safely use the output from the system's previous time step to compute its input at the current time step. This is, in fact, what a solver does when it encounters a loop involving a triggered subsystem, thus eliminating the need for an algebraic loop solver.

Note Because a solver uses a triggered subsystem's previous output to compute feedback inputs, the subsystem, and any block in its feedback path, can exhibit a one sample-time delay in its output. When simulating a system with triggered feedback loops, Simulink displays a warning to remind you that such delays can occur.

Consider, for example, the following system.
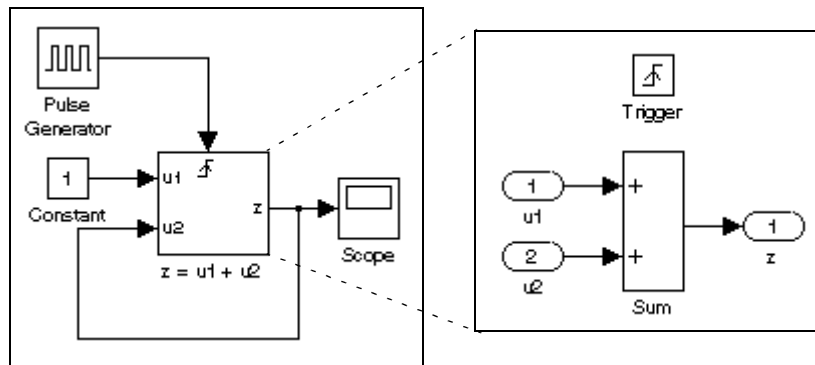
This system effectively solves the equation

    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.
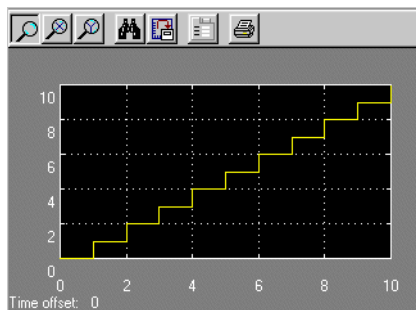
Now consider the effect of removing the trigger from the system shown in the previous example.
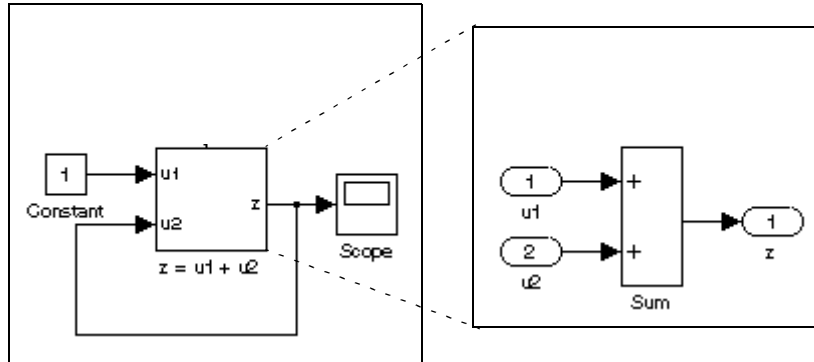
In this case, the input at the u2 port of the adder subsystem is equal to the subsystem's output at the current time step for every time step. The mathematical representation of this system

$$z = z + 1$$

reveals that it has no mathematically valid solution.

Modeling and Simulating Discrete 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.
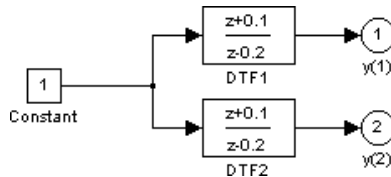
For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to [1 0.1], which gives it an offset of 0.1. The DTF2 Discrete Transfer Fcn block's **Sample time** is set to 0.7, with no offset.

Starting the simulation (see "Running a Simulation from the Command Line" on page 5-3) and plotting the outputs using the stairs function

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

produces this plot
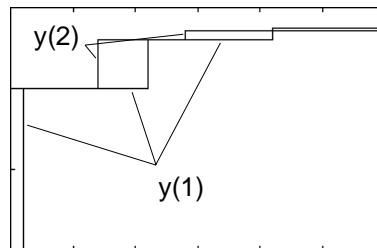
For the DTF1 block, which has an offset of 0.1, there is no output until t = 0.1. Because the initial conditions of the transfer functions are zero, the output of DTF1, y(1), is zero before this time.

Determining Step Size for Discrete Systems

Simulating a discrete system requires that the simulator take a simulation step at every *sample time hit*, that is, at integral multiples of the system's shortest sample time. Otherwise, the simulator may miss key transitions in the system's states. Simulink avoids this by choosing a simulation step size to

ensure that steps coincide with sample time hits. The step size that Simulink chooses depends on the system's fundamental sample time and the type of solver used to simulate the system.

The *fundamental sample time* of a discrete system is the greatest integral divisor of the system's actual sample times. For example, suppose that a system has sample times of 0.25 and 0.5 second. The fundamental sample time in this case is 0.25 second. Suppose, instead, the sample times are 0.5 and 0.75 second. In this case, the fundamental sample time is again 0.25 second.

You can direct Simulink to use either a fixed-step or a variable-step discrete solver to solve a discrete system. A fixed-step solver sets the simulation step size equal to the discrete system's fundamental sample time. A variable-step solver varies the step size to equal the distance between actual sample time hits. The following diagram illustrates the difference between a fixed-step and a variable-size solver.
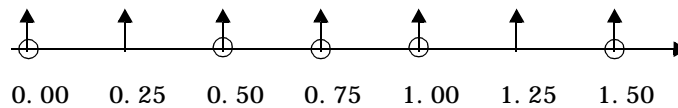
Fixed-Step Solver
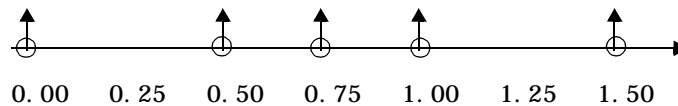
Variable-Step Solver

In the diagram, arrows indicate simulation steps and circles represent sample time hits. As the diagram illustrates, a variable-step solver requires fewer simulation steps to simulate a system, if the fundamental sample time is less than any of the actual sample times of the system being simulated. On the other hand, a fixed-step solver requires less memory to implement and is faster if one of the system's sample times is fundamental. This can be an advantage

in applications that entail generating code from a Simulink model (using the Real-Time Workshop).

Sample Time Propagation

The figure below illustrates a Discrete Filter block with a sample time of Ts driving a Gain block.
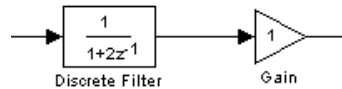
Because the Gain block's output is simply the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to that of the filter's sample rate. This is the fundamental mechanism behind sample time propagation in Simulink.

Simulink sets sample times for individual blocks according to these rules:

- Continuous blocks (e.g., Integrator, Derivative, Transfer Fcn, etc.) are, by definition, continuous.
- The Constant block is, 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.

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.
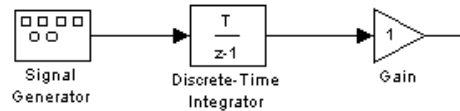
You can verify this by selecting **Sample time colors** from the Simulink **Format** menu and noting that all blocks are colored red. Because the Discrete-Time Integrator block only looks at its input at its sample times, this change does not affect the outcome of the simulation but does result in a 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.
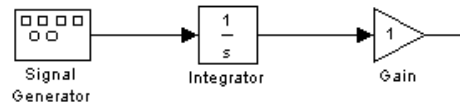
Invariant Constants

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.
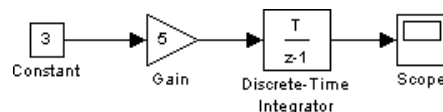
Because Simulink supports the ability to change block parameters during a simulation, all blocks, even blocks having constant sample time, must generate their output at the model's effective sample time.

Note 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.

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 Simulink's inline paramters option (see "Inline parameters" on page 5-30) 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. 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.

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.

4

Creating a Model

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 ![icon] on the MATLAB toolbar.
- Enter the si mul i nk command at the MATLAB prompt.

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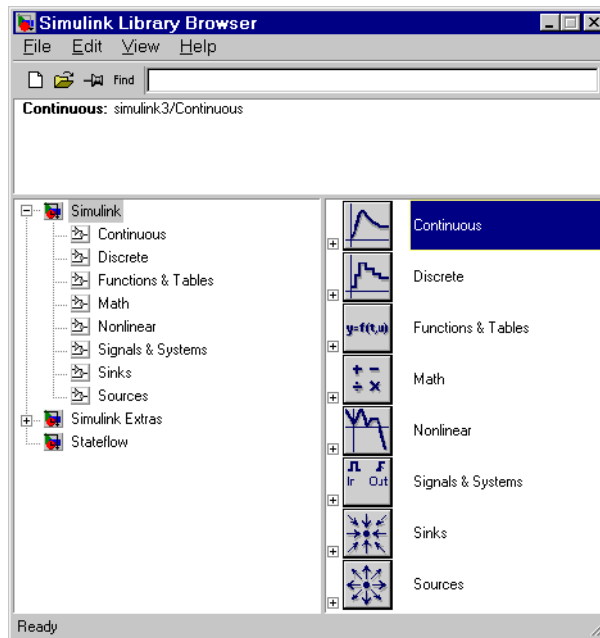
The Library Browser displays a tree-structured view of the Simulink block libraries installed on your system. You can build models by copying blocks from the Library Browser into a model window (this procedure is described later in this chapter).

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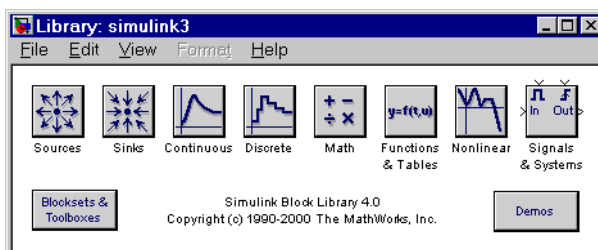
The Simulink library window displays icons representing the block libraries that come with Simulink. You can create models by copying blocks from the library into a model window.

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, "Quick Start" describes how to build a simple model. "Libraries" on page 4–77 describes how to build systems that model equations.

Editing an Existing Model

To edit an existing model diagram, either:

- Click the **Open** button on the Library Browser's toolbar (Windows only) or select **Open** from the Simulink library window's **File** menu and then choose or enter the model filename for the model 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

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.
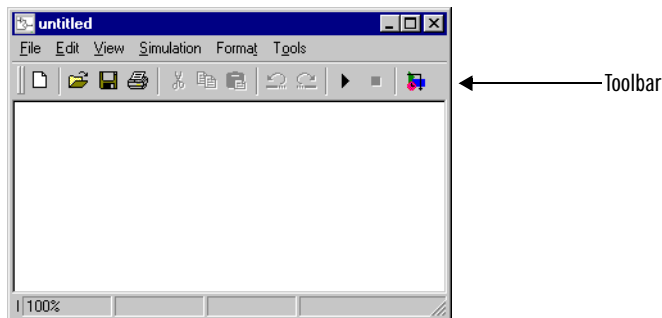
Toolbar

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 the 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 5, and analyzing simulation results is discussed in Chapter 6, "Analyzing Simulation Results.".

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
- Creating a subsystem

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.
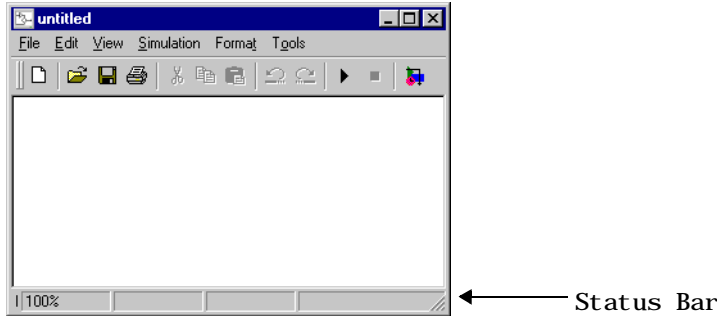
Status Bar

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** option 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.
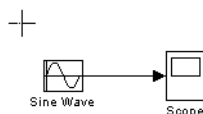
2 Drag the pointer to the opposite corner of the box. A dotted rectangle encloses the selected blocks and lines.
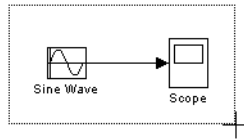
3 Release the mouse button. All blocks and lines at least partially enclosed by the bounding box are selected.
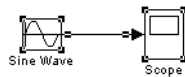
Selecting the Entire Model

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 4–65.

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 options** from the Simulink **View** menu.

Virtual Blocks

When creating models, you need to be aware that Simulink blocks fall into two basic categories: nonvirtual and virtual blocks. Nonvirtual blocks play an active role in the simulation of a system. If you add or remove a nonvirtual block, you change the model's behavior. Virtual blocks, by contrast, play no active role in the simulation; they help organize a model graphically. Some Simulink blocks are virtual in some circumstances and nonvirtual in others. Such blocks are called conditionally virtual blocks. The following table lists Simulink virtual and conditionally virtual blocks.

Table 4-1: Virtual and Conditionally Virtual Blocks

Block Name	Condition Under Which Block Will Be Virtual
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 4-1: Virtual and Conditionally Virtual Blocks (Continued)

Block Name	Condition Under Which Block Will Be Virtual
Ground	Always virtual.
Inport	Virtual *unless* the block resides in a conditionally executed subsystem *and* has a direct connection to an outport block.
Mux	Always virtual.
Outport	Virtual when the block resides within any subsystem block (conditional or not), and does *not* reside in the root (top-level) Simulink window.
Selector	Virtual except in matrix mode.
Subsystem	Virtual except if the block is conditionally executed and/or the block's **Treat as Atomic Unit** option is selected.
Terminator	Always virtual.
Test Point	Always virtual.
Trigger Port	Virtual when the outport 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 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 4-83 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 4–17.

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 4–7.

2 Drag the objects to their new location and release the mouse button.

Copying Blocks in a Model

You can copy blocks in a model as follows. While holding down the **Ctrl** key, select the block with the left mouse button, then drag it to a new location. You can also do this by dragging the block using the right mouse button. Duplicated blocks have the same parameter values as the original blocks. Sequence numbers are added to the new block names.

Block Parameters

All Simulink blocks have a common set of parameters, called block properties, that you can set (see "Common Block Parameters" on page A-7). See "Block Properties Dialog Box" on page 4-13 for information on setting block

properties. In addition, many blocks have one or more block-specific parameters that you can set (see "Block-Specific Parameters" on page A-10). By setting these parameters, you can customize the behavior of the block to meet the specific requirements of your model.

Setting Block-Specific Parameters

Every block that has block-specific parameters has a dialog box that you can use to view and set the parameters. You can display this dialog by selecting the block in the model window and choosing **BLOCK Parameters** from the model window's **Edit** menu or from the model window's context (right-click) menu, where **BLOCK** is the name of the block you selected, e.g., **Constant Parameters**. You can also display a block's parameter dialog box by double-clicking its icon in the model or library window.

Note This holds true for all blocks with parameter dialog boxes except for the Subsystem block. You must use the model window's **Edit** menu or context menu to display a Subsystem block's parameter dialog.

For information on the parameter dialog of a specific block, see the block's documentation in Chapter 9, "Block Reference."

You can set any block parameter, using the Simulink set_param command. See set_param on page 10-27 for details.

You can use any MATLAB constant, variable, or expression that evaluates to an acceptable result when specifying the value of a parameter in a block parameter dialog or a set_param command. You can also use variables or expressions that evaluate to Simulink data objects as parameters (see "Using Data Objects as Parameters" on page 4-53).

Block Properties Dialog Box

Use this dialog box to set a block's properties, i.e., parameters that it shares with all blocks. To display this dialog, select the block in the model window and then select the model window's **Edit** menu. The **Edit** menu includes an item **BLOCK Properties**, where **BLOCK** is the name of the block you selected, e.g., **Constant Properties**. Select this item to display the block's property dialog box.

The **Block Properties** dialog box lets you set some common block parameters.
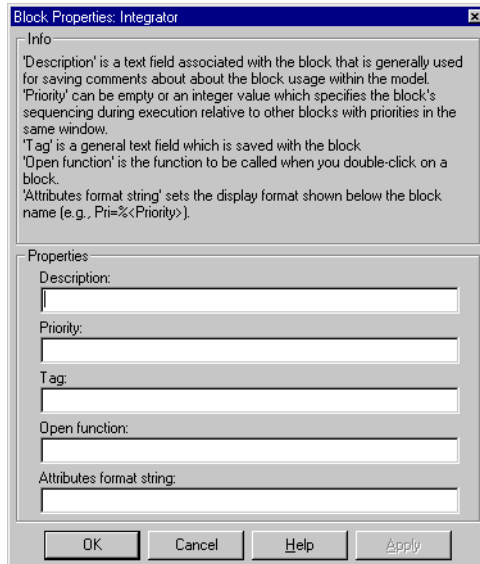
The dialog box contains the following fields:

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 4-18 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. Appendix A describes the parameters that a block can have. You can use the AttributesFormatString parameter to display the values of specified parameters beneath the block's icon.

The attributes format string can be any text string that has embedded parameter names. An embedded parameter name is a parameter name preceded by %< and followed by >, for example, %<priority>. Simulink displays the attributes format string beneath the block's icon, replacing each parameter name with the corresponding parameter value. You can use line-feed characters (\n) to display each parameter on a separate line. For example, specifying the attributes format string

```
pri=%<priority>\ngain=%<Gain>
```

for a Gain block displays

If a parameter's value is not a string or an integer, Simulink displays N/S (not supported) for the parameter's value. If the parameter name is invalid, Simulink displays "???".

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.
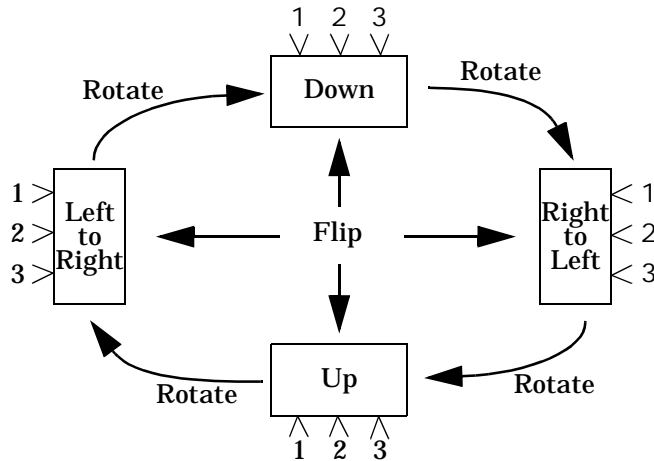
Resizing Blocks

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.
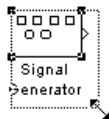
Manipulating Block Names

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 4–15.

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 4-13)

• By setting the value of the block's AttributesFormatString property to the format string, using set_param (see set_param on page 10-27)

Disconnecting Blocks

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

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 4-13).

Simulink honors the block priorities that you specify only if they are consistent with Simulink's block sorting algorithm (see "Determining Block Update Order" on page 3-11). If the specified priorities are inconsistent, Simulink ignores the specified priority and places the block in an appropriate location in the block execution order. If Simulink is unable to honor a block priority, it displays a Block Priority Violation diagnostic message (see "The Diagnostics Pane" on page 5-26).

Displaying Block Execution Order

To display the execution order of blocks during simulation, select **Execution order** from the Simulink **Format** menu. Selecting this option causes Simulink to display a number in the top right corner of each block in a block diagram.
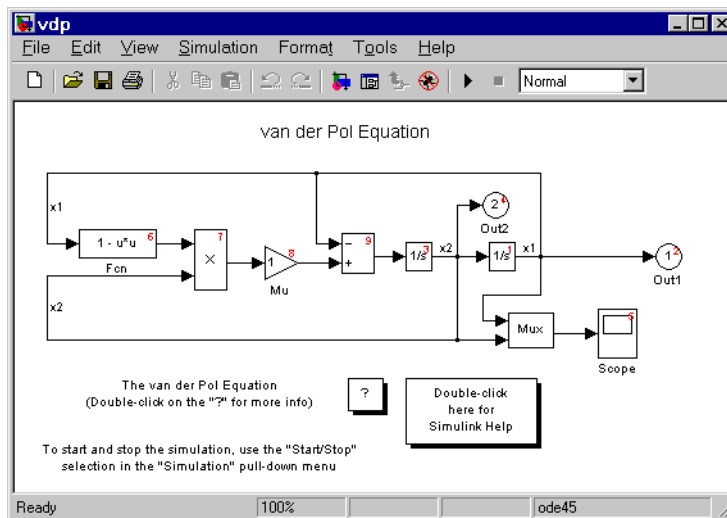
The number indicates the execution order of the block relative to other blocks in the diagram. For example, 1 indicates that the block is the first block executed on every time step, 2 indicates that the block is the second block executed on every time step, and so on.

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.

Sample Time Colors

Simulink can color-code the blocks and lines in your model to indicate the sample rates at which the blocks operate.

Table 4-2: 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
Orange	Sixth fastest discrete sample time
Cyan	Blocks in triggered subsystems

Table 4-2: Sample Time Colors (Continued)

Color	Use
Gray	Fixed in minor step

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.

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.

Connecting Blocks

You can connect an output port of one block to the input port of another block by drawing a line between the blocks. Lines represent pathways for signals generated by a model to travel among blocks. See "Working with Signals" on page 4–28 for information on signals. The rest of this section explains how to draw lines between blocks.

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.
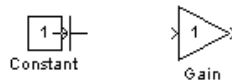
2 Press and hold down the mouse button.

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.
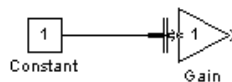
4 Release the mouse button. Simulink replaces the port symbols by a connecting line with an arrow showing the direction of the signal flow. You can create lines either from output to input, or from input to output. The arrow is drawn at the appropriate input port, and the signal is the same.
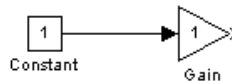
Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

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.
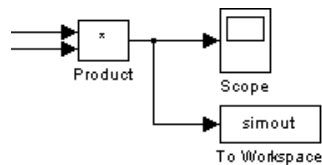
To add a branch line, follow these steps:

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.
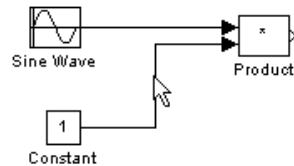
2 Press and hold down the left mouse button.
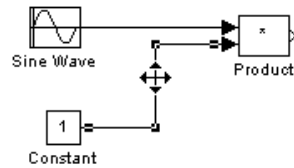
3 Drag the pointer to the desired location.
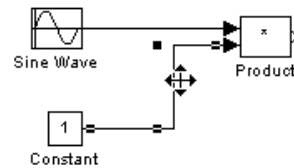
4 Release the mouse button.
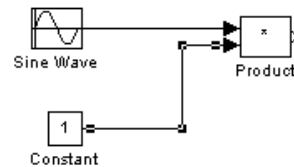
To move the segment connected to an input port, position the pointer over the port and drag the end of the segment to the new location. You cannot move the segment connected to an output port.

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.
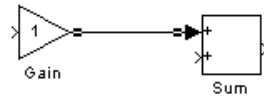
2 Position the pointer on the line where you want the vertex.
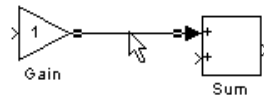
3 While holding down the **Shift** key, press and hold down the mouse button. The cursor shape changes to a circle that encloses the new vertex.
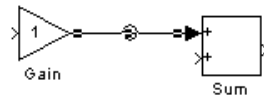
4 Drag the pointer to the desired location.
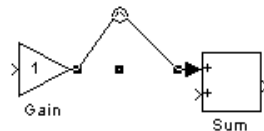
5 Release the mouse button and the **Shift** key.
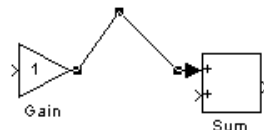
Moving a Line Vertex

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.
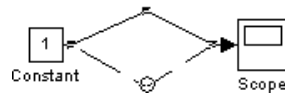
3 Release the mouse button.

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.
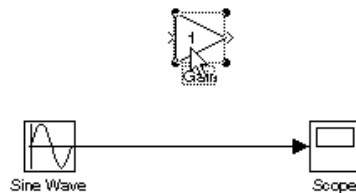
2 Drag the block over the line in which you want to insert the block.
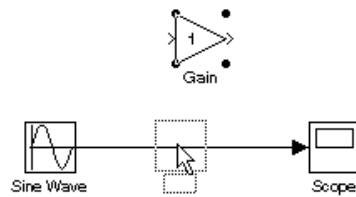
3 Release the mouse button to drop the block on the line. Simulink inserts the block where you dropped it.
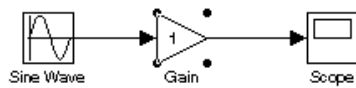
Working with Signals

This section provides an overview of Simulink signals and explains how to specify, display, and check the validity of signal connections.

About Signals

Signals are the streams of values that appear at the outputs of Simulink blocks when a model is simulated. It is useful to think of signals as traveling along the lines that connect the blocks in a model diagram. But note that the lines in a Simulink model represent logical, not physical, connections among blocks. Thus, the analogy between Simulink signals and electrical signals is not complete. Electrical signals, for example, take time to cross a wire. The output of a Simulink block, by contrast, appears instantaneously at the input of the block to which it is connected.

Signal Dimensions

Simulink blocks can output one- or two-dimensional signals. A one-dimensional (1-D) signal consists of a stream of one-dimensional arrays output at a frequency of one array (vector) per simulation time step. A two-dimensional (2-D) signal consists of a stream of two-dimensional arrays emitted at a frequency of one 2-D array (matrix per block sample time. The Simulink user interface and documentation generally refers to 1-D signals as *vectors* and 2-D signals as *matrices*. A one-element array is frequently referred to as a *scalar*. A *row vector* is a 2-D array that has one row. A *column vector* is a 2-D array that has one column.

Simulink blocks vary in the dimensionality of the signals they can accept or output during simulation. Some blocks can accept or output signals of any dimensions. Some can accept or output only scalar or vector signals. To determine the signal dimensionality of a particular block, see the block's description in Chapter 9, "Block Reference." See "Determining Output Signal Dimensions" on page 4-32 for information on what determines the dimensions of output signals for blocks that can output nonscalar signals.

Signal Data Types

Data type refers to the format used to represent signal values internally. The data type of Simulink signals is double by default. However, you can create signals of other data types. Simulink supports the same range of data types as MATLAB. See "Working with Data Types" on page 4-44 for more information.

Complex Signals

The values of Simulink signals can be complex numbers. A signal whose values are complex numbers is called a complex signal. See "Working with Complex Signals" on page 4-36 for information on creating and manipulating complex signals.

Virtual Signals

A *virtual signal* is a signal that represents another signal graphically. Virtual blocks, such as a Mux or Subsystem block (see "Virtual Blocks" on page 4-9), generate virtual signals. Like virtual blocks, virtual signals allow you to simplify your model graphically. For example, using a Mux block, you can reduce a large number of nonvirtual signals (i.e., signals originating from nonvirtual blocks) to a single virtual signal, thereby making your model easier to understand. You can think of a virtual signal as a tie-wrap that bundles together a number of signals.

Virtual signals are purely graphical entities. They have no mathematical or physical significance. Simulink ignores them when simulating a model.

Whenever you run or update a model, Simulink determines the nonvirtual signal(s) represented by the model's virtual signal(s), using a procedure known as *signal propagation*. When running the model, Simulink uses the corresponding nonvirtual signal(s), determined via signal propagation, to drive the blocks to which the virtual signals are connected. For example, in the following model,
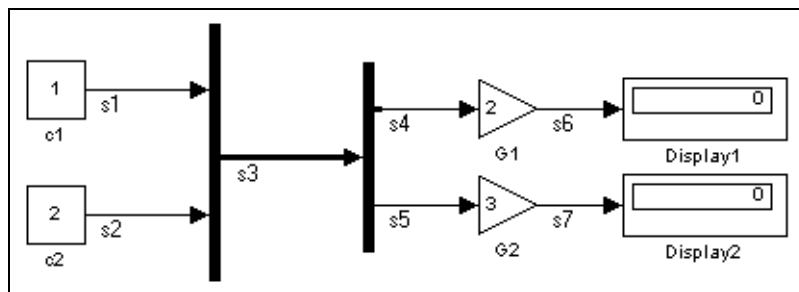
signal s4 appears to drive Gain block G1. However, s4 is a virtual signal. The actual signal driving Gain block G1 is signal s1. Simulink determines this automatically whenever you update or simulate the model.

Simulink's **Show Propagated Signals** option (see "Signal Properties Dialog Box" on page 4-39) displays the nonvirtual signals represented by virtual signals in the labels of the virtual signals.
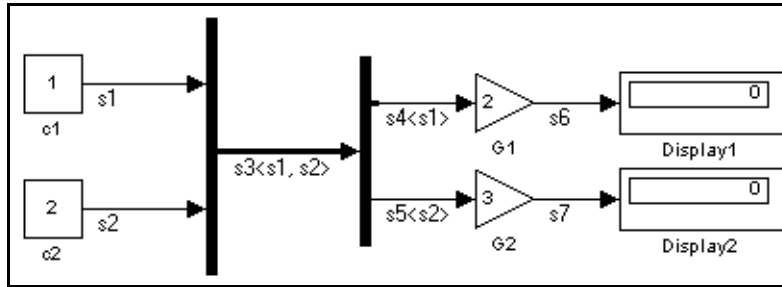
Note Virtual signals can represent virtual as well as nonvirtual signals. For example, you can use a Mux block to combine multiple virtual and nonvirtual signals into a single virtual signal. If during signal propagation, Simulink determines that a component of a virtual signal is itself virtual, Simulink determine its nonvirtual component(s), using signal propagation. This process continues until Simulink has determined all nonvirtual components of a virtual signal.

Signal Buses

You can use Mux and Demux blocks operating in bus selection mode (see "Demux" on page 9-53 for information on bus selection mode) to create signal buses.
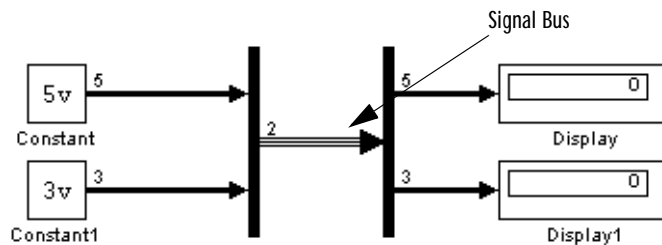
A signal bus is a virtual signal that represents a set of signals. It is analogous to a bundle of wires held together by tie wraps. Simulink uses a special line style to display signal buses. If you select **Signal Dimensions** from Simulink's **Format** menu, Simulink displays the number of signal components carried by the bus.

Signal Glossary

The following table summarizes the terminology used to describe signals in the Simulink user interface and documentation.

Term	Meaning
Complex signal	Signal whose values are complex numbers
Data type	Format used to represent signal values internally. See "Working with Data Types" on page 4–44 for more information.
Matrix	Two-dimensional signal array
Real signal	Signal whose values are real (as opposed to complex) numbers
Scalar	One-element array, i.e., a one-element, 1-D or 2-D array
Signal bus	Signal created by a Mux or Demux block.
Signal propagation	Process used by Simulink to determine attributes of signals and blocks, such as data types, labels, sample time, dimensionality, and so on, that are determined by connectivity
Size	Number of elements that a signal contains. The size of a matrix (2-D) signal is generally expressed as M-by-N where M is the number of columns and N is the number of rows making up the signal.
Vector	One-dimensional signal array

Term	Meaning
Width	Size of a vector signal
Virtual signal	Signal that represents another signal or set of signals.

Determining Output Signal Dimensions

If a block can emit nonscalar signals, the dimensions of the signals that the block outputs depends on the block's parameters, if the block is a source block; otherwise, the output dimensions depend on the dimensions of the block's input and parameters.

Determining the Output Dimensions of Source Blocks

A *source* block is a block that has no inputs. Examples of source blocks include the Constant block and the Sine Wave block. (See Table 9-1, Sources Library Blocks for a complete listing of Simulink source blocks.) The output dimensions of a source block are the same as that of its output value parameter(s) if the block's **Interpret Vector Parameters as 1-D** parameter is off (i.e., not checked in the block's parameter dialog box). If the **Interpret Vector Parameters as 1-D** parameter is on, the output dimensions equal the output value parameter dimensions except if the parameter dimensions are N-by-1 or 1-by-N. In the latter case, the block outputs a vector signal of width N.

As an example of how a source block's output value parameter(s) and **Interpret Vector Parameters as 1-D** parameter determine the dimensionality of its output, consider the Constant block. This block outputs a constant signal equal to its **Constant value** parameter. The following table illustrates how the dimensionality of the **Constant value** parameter and the setting of the **Interpret Vector Parameters as 1-D** parameter determine the dimensionality of the block's output.

Constant Value	Interpret vector parameters as 1-D	Output
2-D scalar	off	2-D scalar
2-D scalar	on	1-D scalar

Constant Value	Interpret vector parameters as 1-D	Output
1-by-N matrix	off	1-by-N matrix
1-by-N matrix	on	N-element vector
N-by-1 matrix	off	N-by-1 matrix
N-by-1 matrix	on	N-element vector
M-by-N matrix	off	M-by-N matrix
M-by-N matrix	on	M-by-N matrix

Simulink source blocks allow you to specify the dimensions of the signals that they output. You can therefore use them to introduce signals of various dimensions into your model.

Determining the Output Dimensions of Non-Source Blocks

If a block has inputs, the dimensions of its output are, after scalar expansion, the same as those of its inputs. (All inputs must have the same dimensions as discussed in the next section.)

Signal and Parameter Dimension Rules

When creating a Simulink model, you must observe the following rules regarding signal and parameter dimensions.

Input Signal Dimension Rule

In general, all inputs to a block must have the same dimensions.

However, a block may have a mix of scalar and nonscalar inputs as long as all the nonscalar inputs have the same dimensions. Simulink expands the scalar inputs to have the same dimensions as the nonscalar inputs (see "Scalar Expansion of Inputs" on page 4-35), thus preserving the general rule.

Block Parameter Dimension Rule

In general, a block's parameters must have the same dimensions as the corresponding inputs.

Two seeming exceptions exist to this general rule:

- A block may have scalar parameters corresponding to nonscalar inputs. In this case, Simulink expands the scalar parameter to have the same dimensions as the input (see "Scalar Expansion of Parameters" on page 4-35), thus preserving the general rule.
- If an input is a vector, the corresponding parameter may be either an Nx1 or a 1xN matrix. In this case, Simulink applies the N matrix elements to the corresponding elements of the input vector. This exception allows use of MATLAB row- or column vectors, which are actually 1xN or Nx1 matrices, respectively, to specify parameters that apply to vector inputs.

Vector or Matrix Input Conversion Rules

Simulink converts vectors to row or column matrices and row or column matrices to vectors under the following circumstances:

- If a vector signal is connected to an input that requires a matrix, Simulink converts the vector to a one-row or one-column matrix.
- If a one-column or one-row matrix is connected to an input that requires a vector, Simulink converts the matrix to a vector.
- If the inputs to a block consist of a mixture of vectors and matrices and the matrix inputs all have one column or one row, Simulink converts the vectors to matrices having one column or one row, respectively.

Note You can configure Simulink to display a warning or error message if a vector or matrix conversion occurs during a simulation. See "Configuration options" on page 5–27 for more information.

Scalar Expansion of Inputs and Parameters

Scalar expansion is the conversion of a scalar value into a nonscalar array of the same dimensions. Many Simulink blocks support scalar expansion of inputs and parameters. Block descriptions in Chapter 9, "Block Reference" indicate whether Simulink applies scalar expansion to a block's inputs and parameters.

Scalar Expansion of Inputs

Scalar expansion of inputs refers to the expansion of scalar inputs to match the dimensions of other nonscalar inputs or nonscalar parameters. When the input to a block is a mix of scalar and nonscalar signals, Simulink expands the scalar inputs into nonscalar signals having the same dimensions as the other nonscalar inputs. The elements of an expanded signal equal the value of the scalar from which the signal was expanded.

The following model illustrates scalar expansion of inputs. This model adds scalar and vector inputs. The input from block Constant1 is scalar expanded to match the size of the vector input from the Constant block. The input is expanded to the vector [3 3 3].
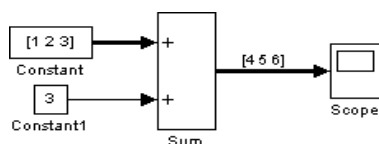
When a block's output is a function of a parameter and the parameter is nonscalar, Simulink expands a scalar input to match the dimensions of the parameter. For example, Simulink expands a scalar input to a Gain block to match the dimensions of a nonscalar gain parameter.

Scalar Expansion of Parameters

If a block has a nonscalar input and a corresponding parameter is a scalar, Simulink expands the scalar parameter to have the same number of elements as the input. Each element of the expanded parameter equals the value of the original scalar. Simulink then applies each element of the expanded parameter to the corresponding input element.

This example shows that a scalar parameter (the Gain) is expanded to a vector of identically valued elements to match the size of the block input, a three-element vector.
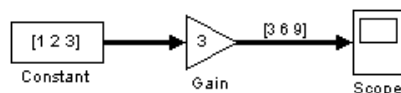
Working with Complex Signals

By default, the values of Simulink signals are real numbers. However, models can create and manipulates 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 9, "Block Reference."

Checking Signal Connections

Many Simulink blocks have limitations on the types of signals they can accept. Before simulating a model, Simulink checks all of blocks to ensure that they can accommodate the types of signals output by the ports to which they are connected. If any incompatibilities exist, Simulink reports an error and terminates the simulation. To detect such errors before running a simulation, choose **Update Diagram** from the Simulink **Edit** menu. Simulink reports any invalid connections found in the process of updating the diagram.

Setting Signal Display Options

Simulink offers the following options for displaying signal characteristics on a block diagram.

Signal Display Option	Description
Wide nonscalar lines	Draws lines that carry vector or matrix signal wider than lines that carry scalar signals.
Signal dimensions	Displays the dimensions of a signal next to the line that carries it.
Port data types	Displays the data type and signal type of a signal next to the output port that emits the signal.

You can set these options via either Simulink's **Format** menu or its model context (right-click) menu.

Signal Names

You can assign names to signals by

- editing the signal's label
- editing the **Name** field of the signal's property dialog (see "Signal Properties Dialog Box" on page 4-39)
- setting the name parameter of the port or line that represents the signal, e.g.,

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

Signal Labels

A signal's label displays the signal's name. A virtual signal's label optionally displays the signals it represents in angle brackets. You can edit a signal's label, thereby changing the signal's name.

To create a signal label (and thereby name the signal), double-click on the line that represents the signals. The text cursor appears. Type the name and click anywhere outside the label to exit label editing mode.

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

Labels can appear above or below horizontal lines or line segments, and left or right of vertical lines or line segments. Labels can appear at either end, at the center, or in any combination of these locations.

To move a signal label, drag the label to a new location on the line. When you release the mouse button, the label fixes its position near the line.

To copy a signal label, hold down the **Ctrl** key while dragging the label to another location on the line. When you release the mouse button, the label appears in both the original and the new locations.

To edit an existing signal label, select it:

- To replace the label, click 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.

Displaying Signals Represented by Virtual Signals

To display the signal(s) represented by a virtual signal, click the signal's label and enter an angle bracket (<) after the signal's name. (If the signal has no

name, simply enter the angle bracket.) Click anywhere outside the signal's label. Simulink exits label editing mode and displays the signals represented by the virtual signal in brackets in the label.

You can also display the signals represented by a virtual signal by selecting the **Show Propagated Signals** option on the signal's property dialog (see "Signal Properties Dialog Box" on page 4-39).

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 lets you view and edit signal properties.
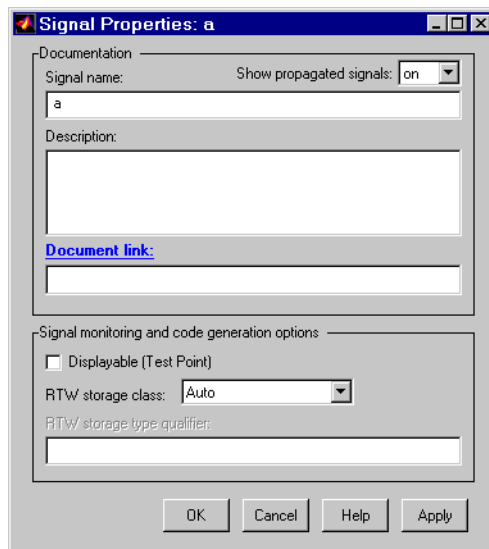
The dialog box includes the following controls.

Signal name
Name of signal.

Show propagated signals

Note This option appears only for signals that originate from a virtual block.

Show propagated signal names. You can select one of the following options:

Option	Description
off	Do not display signals represented by a virtual signal in the signal's label.
on	Display the virtual and nonvirtual signals represented by a virtual signal in the signal's label. For example, suppose that virtual signal s1 represents a nonvirtual signal s2 and a virtual signal s3. If this option is selected, s1's label is s1<s2, s3>.
all	Display all the nonvirtual signals that a virtual signal represents either directly or indirectly. For example, suppose that virtual signal s1 represents a nonvirtual signal s2 and a virtual signal s3 and virtual signal s3 represents nonvirtual signals s4 and s5. If this option is selected, s1's label is s1<s2, s4, s5>.

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 "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 set properties used by the Real-Time Workshop to generate code from the model. You can ignore them if you are not going 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.
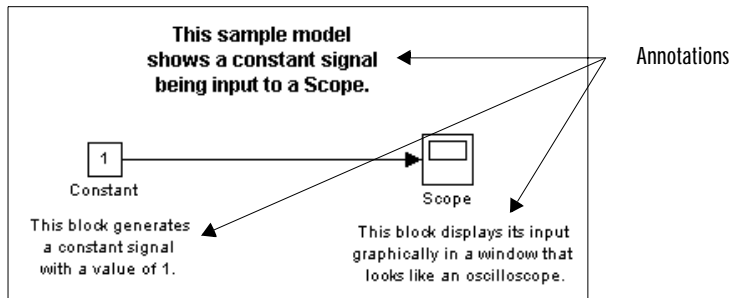
To create a model annotation, double-click on an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point. Start typing the annotation contents. Each line is centered within the rectangle that surrounds the annotation.

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.

To change the text alignment (e.g., left, center, or right) of the annotation, select the annotation and choose **Text Alignment** from the model window's

Format or context menu. Then choose one of the alignment options (e.g., **Center)** from the **Text Alignment** submenu.

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 (doubl e) 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 4-47). 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.

Name	Description
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. See Chapter 9, "Block Reference" for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type double.

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.

single(1.0)	Specifies a single-precision value of 1.0
int8(2)	Specifies an eight-bit integer of value 2
int32(3+2i)	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 4-48 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 4-48 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 array connected to a port of a nonvirtual block must be of the same data type.

- The signals connected to the input data ports of a nonvirtual block cannot differ in type.

- Control ports (for example, Enable and Trigger ports) accept any data type.

- Solver blocks accept only double signals.

- Connecting a 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 blocks that prefer bool ean input. This ensures compatibility with models created by earlier versions of Simulink that support only doubl e data type. You can enable strict boolean type checking by unchecking the **Boolean logic signals** option on the **Advanced** panel of the **Simulation Parameters** dialog box (see "The Advanced Pane" on page 5-29).

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 9-49).

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.
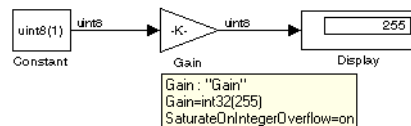
  This model uses a Gain block to amplify a constant input signal. Computing the output of the Gain block requires computing the product of the input signal and the gain. Such a computation requires that the two values be of the same data type. However, in this case, the data type of the signal, ui nt 8 (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 optionally issues a warning message and continues the simulation (see "Configuration options" on page 5-27).

Consider, for example, the following model.
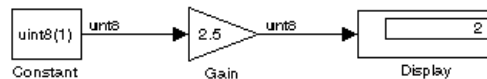
In this example, the signal type accommodates only integer values while the gain value has a fractional component. Simulating this model causes Simulink to truncate the gain to the nearest integral value (2) and issue a loss-of-precision warning. On the other hand, if the gain were 2.0, Simulink would 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 Data Objects

Simulink data objects allow you to specify information about the data used in a Simulink model (i.e., signals and parameters) and to store the information with the data itself in the model. Simulink uses properties of data objects to determine the tunability of parameters and the visibility of signals and to generate code. You can use data objects to specify information important to correct simulation of the model, such as minimum and maximum values for parameters. Further, you can store data objects with the model. Simulink thus allows you to create self-contained models.

Data Object Classes

A data object is an instance of another object called a data object *class*. A data object class defines the properties of its instances and methods for creating and manipulating the instances. Simulink comes with two built-in data classes, Simulink.Parameter and Simulink.Signal, that define parameter and signal data objects, respectively.

Data Object Properties

A property of a data object specifies an attribute of the data item that the object describes, such as the value or storage type of the data item. Every property has a name and a value. The value can be an array or a structure, depending on the property.

Data Object Packages

Simulink organizes classes into groups of classes called *packages*. Simulink comes with a single package named Simulink. The Simulink classes, Simulink.Parameter and Simulink.Signal, belong to the Simulink package. You can create additional packages and define classes that belong to those classes.

Qualified Names

When referring to a class on the MATLAB command line or in an M-file program, you must specify both the name of the class and the name of the class's package, using the following "dot" notation

    PackageName.ClassName

The PackageName.ClassName notation is called the qualified name of the class. For example, the qualified name of the Simulink parameter class is Simulink.Parameter.

Two packages can have identically named but distinct classes. For example, package A and B can both have a class named C. You can refer to these classes unambiguously on the MATLAB command line or in M-file program, using the qualified class name for each. Packages enable you to avoid naming conflicts when creating classes. For example, you can create your own Parameter and Signal classes without fear of conflicting with the similarly named Simulink classes.

Note Class and package names are case-sensitive. You cannot, for example, use A.B and a.b interchangeably to refer to the same class.

Creating Data Objects

You can use either the Simulink Data Explorer or MATLAB commands to create Simulink data objects. See "The Simulink Data Explorer" on page 4-60 for information on using the Data Explorer to create data objects.

Use the following syntax to create a data object at the MATLAB command line or in a program

```
h = package.class(arg1, arg2, ...argn);
```

where h is a MATLAB variable, package is the name of the package to which the class belongs, class is the name of the class, and arg1, arg2, ... argn, are optional arguments passed to the object constructor. (Constructors for the Simulink.Parameter and Simulink.Signal classes do not take arguments.) For example, to create an instance of Simulink.Parameter class, enter

```
hGain = Simulink.Parameter;
```

at the MATLAB command line.

This command creates an instance of Simulink.Parameter and stores its handle in gain.

Accessing Data Object Properties

You can use either the Simulink Data Explorer (see "The Simulink Data Explorer" on page 4-60) or MATLAB commands to get and set a data object's properties. See "Creating a Package" on page 4-56 for information on how to use the Data Explorer to display and set object properties.

To access a data object's properties at the MATLAB command line or in an M-file program, use the following notation.

```
hObject.property
```

where hObject is the handle to the object and property is the name of the property. For example, the following code

```
hGain = Simulink.Parameter;
hGain.Value = 5;
```

creates a Simulink block parameter object and sets the value of its Value property to 5. You can use dot notation recursively to access the fields of structure properties. For example, gain.RTWInfo.StorageClass returns the StorageClass property of the gain parameter.

Invoking Data Object Methods

Use the syntax

```
hObject.method
```

or

```
method(hObject)
```

to invoke a data object method, where hObject is the object's handle. Simulink defines the following methods for data objects.

- get
  Returns the properties of the object as a MATLAB structure
- copy
  Creates a copy of the object and returns a handle to the copy.

Saving and Loading Data Objects

You can use the MATLAB save command to save data objects in a MAT-file and the MATLAB load command to restore them to the MATLAB workspace in the same or a later session. Definitions of the classes of saved objects must exist on the MATLAB path for them to be restored. If the class of a saved object acquires new properties after the object is saved, Simulink adds the new properties to the restored version of the object. If the class loses properties after the object is saved, Simulink restores only the properties that remain.

Using Data Objects in Simulink Models

You can use data objects in Simulink models as parameters and signals. Using data objects as parameters and signals allows you to specify simulation and code generation options on an object-by-object basis.

Using Data Objects as Parameters

You can use an instance of Simulink.Parameter class or a descendant class as a block parameter. To use a parameter object as a block parameter,

1 Create the parameter object at the MATLAB command line or in the Simulink Data Explorer.

2 Set the value of the object's Value property to the value you want to specify for the block parameter.

3 Set the parameter objects storage class and type properties to select tunability (see "Creating Data Object Classes" on page 4-55) and/or code generation options (see the Real-Time Workshop documentation for more information) .

4 Specify the parameter object as the block parameter in the block's parameter dialog box or in a set_param command.

See "Creating Persistent Parameter and Signal Objects" on page 4-55 for information on how to create parameter objects that persist across a session.

Using Parameter Objects to Specify Parameter Tunability

If you want the parameter to be tunable even when the Inline parameters simulation option is set (see "Model parameter configuration" on page 5-30), set

the parameter object's RTWInfo. StorageClass property to any value but 'Auto' (the default).

gain. RTWInfo. StorageClass = 'SimulinkGlobal';

If you set the RTWInfo. StorageClass property to any value other than Auto, you should not include the parameter in the model's tunable parameters table (see "Model Parameter Configuration Dialog Box" on page 5-32).

Note Simulink halts the simulation and displays an error message if it detects a conflict between the properties of a parameter as specified by a parameter object and the properties of the parameter as specified in the Model Parameter Configuration dialog box.

Using Data Objects as Signals

You can use an instance of Simulink. Signal class or a descendant class to specify signal properties. To use a data object a signal object to specify signal properties,

1 Create the signal data object in the model workspace.

2 Set the storage class and type properties of the signal object to specify the visibility of the signal (see "Using Signal Objects to Specify Test Points" on page 4-54) and code generation options (see the Real-Time Workshop documentation for information on using signal properties to specify code generation options).

3 Change the label of any signal that you want to have the same properties as the signal data object to have the same name as the signal.

See "Creating Persistent Parameter and Signal Objects" on page 4-55 for information on creating signal objects that persist across Simulink sessions.

Using Signal Objects to Specify Test Points

If you want a signal to be a test point (i.e., always available for display on a floating scope during simulation), set the RTWInfo. StorageClass property of the corresponding signal object to any value but auto.

Note Simulink halts the simulation and displays an error message if it detects a conflict between the properties of a signal as specified by a signal object and the properties of the parameter as specified in the **Signal Properties** dialog box (see "Signal Properties Dialog Box" on page 4-39).

Creating Persistent Parameter and Signal Objects

To create parameter and signal objects that persist across Simulink sessions, first write a script that creates the objects or create the objects at the command line and save them in a MAT-file (see "Saving and Loading Data Objects" on page 4-53). Then use either the script or a load command as the PreLoadFcn callback routine for the model that uses the objects. For example, suppose you save the data objects in a file named data_objects.mat and the model to which they apply is open and active. Then, entering the following command

```
set_param(gcs, 'PreLoadFcn', 'load data_objects');
```

at the MATLAB command line sets load data_objects as the model's preload function. This in turn causes the data objects to be loaded into the model workspace whenever you open the model.

Creating Data Object Classes

Creating a new data object class entails writing M-file programs to construct and instantiate instances of the class. If you want to create a new package to contain the class, you must also write an M-file constructor for the new package.

Note The Simulink demos directory (matlabroot/toolbox/simulink/simdemos) contains a sample user-defined data object class definition called UserDefined. You can use this class definition as a template for creating your own classes. You can copy and edit this sample class to create your own class.

Package Directory Structure

You must store the programs that define a class in a directory that has a prescribed structure.
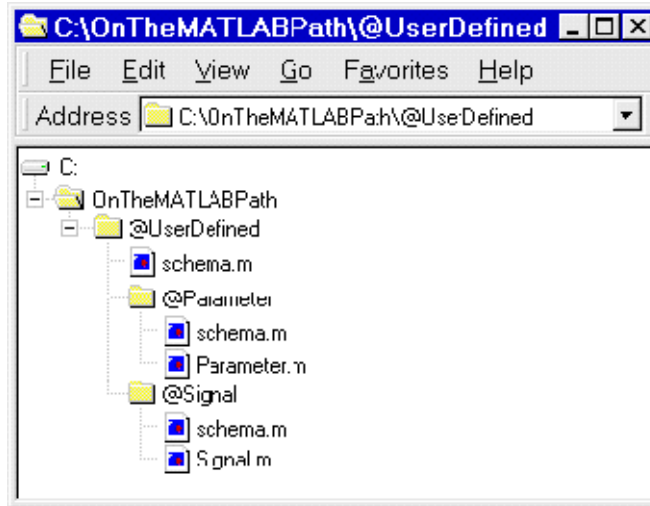
The directory structure must mee tthe following requirements.

- Each package must have its own directory, called the package directory, on the MATLAB command path. The package directory must be named @PackageName where PackageName is the name of the package.
- The code for each class in a package must reside in a separate subdirectory of the package directory called the class directory. The class directory must be named @ClassName where ClassName is the name of the new class.

The package directory must contain an M-file program, named schema.m, that constructs the package. Each class directory must contain a constructor, named schema.m, and an instantiation function, named ClassName.m, where ClassName is the name of the class.

Creating a Package

To create a package, first create a directory named @package_name in a directory on the MATLAB path, where @PackageName is the name of the new

package. Then create a M-file named schema.m in the package directory. The schema.m file MATLAB function.

```
function schema ()
% Package constructor function

schema.package('PackageName');
```

where PackageName is the name of the new package.

Creating a Class

To create a data object class,

1 Create a directory named @ClassName, where ClassName is the name of the new class, in the directory of the package in which you want the new class to reside.

2 Create a class constructor in the class directory.

3 Create a class instantiation function in the class directory.

Creating a Class Constructor

MATLAB finds the constructor for a class by looking for a function named schema in the class directory. You must therefore create this function in the class directory of the class you are creating. The constructor creates the class by invoking the create_user_class function (see "create_user_class" on page 4-59) as illustrated in the following example.

```
function schema()
% Class constructor function.

% Specify name of class to be created:
userClass = 'UserDefined.Parameter';

% Specify name of class from which user class is derived:
deriveFromClass = 'Simulink.Parameter';

% Call generalized constructor function for
% user-defined enumerations used by this class
create_user_enumtype('colors', {'red', 'green', 'blue'});
```

```
% Specify new properties to include in user class:
addProperties = {
                  'UserMATLABArray1',  'MATLAB array',  []; ...
                  'UserMATLABArray2',  'MATLAB array',  ''; ...
                  'UserDouble',        'double',        0; ...
                  'UserInt32',         'int32',         0; ...
                  'UserOnOff',         'on/off',        'off'; ...
                  'UserString',        'string',        ''; ...
                  'UserColorEnum',     'colors',        'red'; ...
                };
```

```
% Call generalized class creation function (built-in)
create_user_class(userClass, deriveFromClass, addProperties);
```

Creating a Class Instantiation Function

Simulink uses the class instantiation function to create an instance of a class. It finds the class instantiation function by looking in the class directory for an M-file that has the same name as the class. For example, if the name of the class is Parameter, Simulink looks for an M-file named Parameter.m and containing a function named Parameter that returns a handle to the function. A minimal instantiation function takes no arguments and simply invokes the default instantiation function for the class as illustrated in the following example.

```
function h = Parameter()
% Class instantiation function.
% Instantiate class
h = UserDefined.Parameter;
```

An instantiation function can optionally take a variable number of arguments. The function can use the optional arguments to initialize the properties of the object as illustrated in the following example.

```
function h = Parameter(varargin)
% Class instantiation function.
% Instantiate class
h = UserDefined.Parameter;

% Initialize property values (optional)
if nargin == 1
  % If only one argument provided, treat it as the "Value".
```

```
    h. Value = varargin{1};
  end
```

Creating Data Object Properties

A data object class inherits the properties of its parent class. You can define additional properties for the class in its constructor. To do so, pass an n-by-3 cell array to the class constructor function (see "create_user_class" on page 4-59) where n is the number of properties to be specified. Each row of the array should specify the name (e.g., 'angle'), type (e.g., 'double'), and default value of the corresponding property.

The Simulink.Signal and Simulink.Parameter classes are likely to acquire new properties in future releases. Consequently, when deriving classes from these classes, you should use property names that are not likely to conflict with names of future properties of these classes. One approach to avoid a naming conflict is to append your company's name to names of properties of derived classes.

Data Object Functions

Simulink provides the following functions for creating and manipulating Simulink data objects and classes.

create_user_class. Use this function in a class constructor file (schema.m) to create a new data object class. It takes three arguments

- The qualified name of the new class (e.g., 'UserDefined.Parameter')
- The qualified name of the parent of the new class (e.g., 'Simulink.Parameter')
- A cell array specifying the properties of the new class (see "Creating Data Object Properties" on page 4-59)

create_user_enumtype. Use this function in a class constructor to create an enumerated data type, that is, a data type with a specified set of valid values. You can then use the enumerated type as the type of one or more of a class's properties. The create_user_enumtype function takes two arguments.

- The name of the enumerated type
- A cell array specifying the set of values that are valid for instances of this type

4-59

For example, the following code creates an enumerated type named colors.

```
create_user_enumtype('colors', {'red', 'green', 'blue'});
```

findpackage. Returns a handle to a package object, for example,

```
h_SimulinkPackage = findpackage('Simulink');
```

findclass. Returns a handle to a class, for example,

```
h_SimulinkParameter = findclass(h_SimulinkPackage, 'Parameter');
```

findproperty. Returns a handle to an object property, for example,

```
h_ParamValue = findparameter(h_SimulinkParameter, 'Value');
```

The Simulink Data Explorer

The Simulink Data Explorer allows you to display and set the values of variables and data objects in the MATLAB workspace. To open the Data Explorer, choose **Data explorer** from the Simulink **Tools** menu or type slexplr at the MATLAB prompt. The Data Explorer dialog box appears.
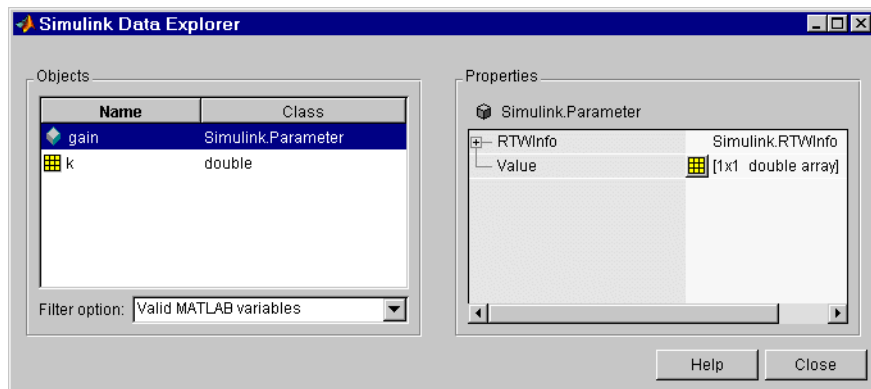
The Data Explorer contains two panes. The left pane lists variables defined in the MATLAB workspace. Use the Filter option control to specify the types of variables to be displayed (for example, all variables or Simulink data objects only). The right pane displays the value of the variable selected in the left pane. To create, rename, or delete an object, click the right mouse button in the left

pane. To display the fields of a property structure, click the + button next to the property's name.

To change a value, click the value. If the value is a string, edit the string. If the property must be set to one of a predefined set of values, the Data Explorer displays a drop down list displaying valid values. Select the value you want. If the value is an array, the Data Explorer displays an array editor

that allows you to set the dimensions of the array and the values of each element.

4-61

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 4-3: Manipulating Blocks

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

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

Table 4-4: Manipulating Lines

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

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

Table 4-5: Manipulating Signal Labels

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

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

Table 4-6: Manipulating Annotations

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

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.

   Simulink opens the subsystem in the current or a new model window, depending on the model window reuse mode that you have selected (see "Window Reuse" on page 4-67).

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:
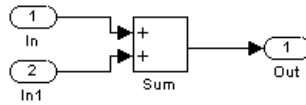
Creating a Subsystem by Grouping Existing Blocks

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 4–7.

For example, this figure shows a model that represents a counter. The Sum and Unit Delay blocks are selected within a bounding box.
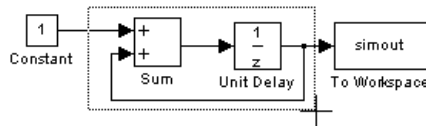
When you release the mouse button, the two blocks and all the connecting lines are selected.

2 Choose **Create Subsystem** from the **Edit** menu. 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).
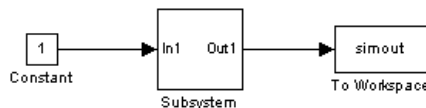
If you open the Subsystem block, Simulink displays the underlying system, as shown below. Notice that Simulink adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem.
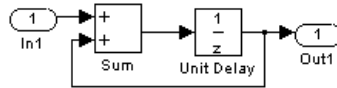
As with all blocks, you can change the name of the Subsystem block. Also, you can customize the icon and dialog box for the block using the masking feature, described in Chapter 7, "Using Masks to Customize Blocks."

Model Navigation Commands

Subsystems allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy, using the Simulink Model Browser (see "Searching and Browsing Models" on page 4-94) and/or the following model navigation commands.

- **Open**

  The **Open** command opens the currently selected subsystem. To execute the command, choose **Open** from the Simulink **Edit** menu, type Enter, or double-click the subsystem.

- **Go to Parent**

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

Window Reuse

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

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

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

Labeling Subsystem Ports

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

You can hide (or show) the port labels by

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

This figure shows two models. The subsystem on the left contains two Inport blocks and one Outport block. The Subsystem block on the right shows the labeled ports.
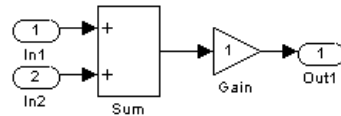
Subsystem with Inport and Outport blocks   Subsystem with labeled ports
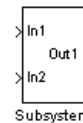
Controlling Access to Subsystems

Simulink allows you to control user access to subsystems that reside in libraries. In particular, you can prevent a user from viewing or modifying the contents of a library subsystem while still allowing the user to employ the subsystem in a model.

To control access to a library subsystem, open the subsystem's parameter dialog box and set its **Access** parameter to either ReadOnly or NoReadOrWrite. The first option allows a user to view the contents of the library subsystem and make local copies but prevents the user from modifying the original library copy. The second option prevents the user from viewing the contents of, creating local copies, or modifying the permissions of the library subsystem. See Subsystem on page 9-239 for more information on subsystem access options. Note that both options allow a user to use the library system in models by creating links (see "Libraries" on page 4-77).

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 10-27).

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.

Tracing Callbacks

Callback tracing allows you to determine which callbacks Simulink invokes and in what order Simulink invokes them when you open or simulate a model. To enable callback tracking, select the **Callback tracing** option on the Simulink Preferences dialog box (see "Setting Simulink Preferences" on page 2-15) or execute set_param(0, 'CallbackTracing', 'on'). This options causes Simulink to list callbacks in the MATLAB command window as they are invoked.

Model Callback Parameters

This table lists the parameters for which you can define model 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.

Parameter	When Executed
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.

Block Callback Parameters

This table lists the parameters for which you can define block 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.

Parameter	When Executed
CloseFcn	When the block is closed using the close_system 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 add_block 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.

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 open_system command is called with the block as an argument. The OpenFcn 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 new_system command (see new_system on page 10-22).
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. In the case of an S-Function block, StartFcn executes immediately before the first execution of the block's mdlProcessParameters function. See "Overview of the C MEX S-Function Routines" in Chapter 3 of *Writing S-Functions* for more information.
StopFcn	At any termination of the simulation. In the case of an S-Function block, StopFcn executes after the block's mdlTerminate function executes. See "Overview of the C MEX S-Function Routines" in Chapter 3 of *Writing S-Functions* for more information.
UndoDeleteFcn	When a block delete is undone.

Parameter	When Executed
CloseFcn	When the block is closed using the close_system 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 add_block 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.

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 open_system command is called with the block as an argument. The OpenFcn 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 new_system command (see new_system on page 10-22).
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. In the case of an S-Function block, StartFcn executes immediately before the first execution of the block's mdlProcessParameters function. See "Overview of the C MEX S-Function Routines" in Chapter 3 of *Writing S-Functions* for more information.
StopFcn	At any termination of the simulation. In the case of an S-Function block, StopFcn executes after the block's mdlTerminate function executes. See "Overview of the C MEX S-Function Routines" in Chapter 3 of *Writing S-Functions* for more information.
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 4–65. The Model Browser (see "The Model Browser" on page 4-99) 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 Names" on page 4–37 and "Drawing a Line Between Blocks" on page 4–22.

- 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.

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.

Library (Source)   Model or Library (Destination)
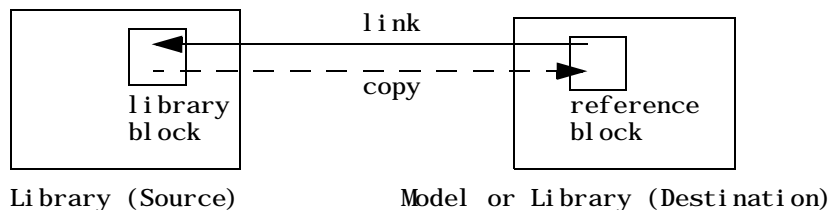
Creating a Library

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, "Model Construction Commands.".

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.

Creating a Library Link

To create a link to a library block in a model, copy the block's icon from the library to the model (see "Copying and Moving Blocks from One Window to Another" on page 4-10) or by dragging the block from the Library Browser (see "Browsing Block Libraries" on page 4-83) 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 change the values of the reference block's parameters 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 the link is to a subsystem, you can modify the contents of the reference subsystem (see "Modifying a Linked Subsystem" on page 4-79).

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).
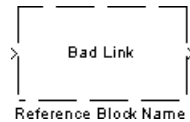
To fix a bad link, you must either:

- 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**.

Disabling Library Links

Simulink allows you to disable linked blocks in a model. Simulink ignores disabled links when simulating a model. To disable a link, select the link, choose **Link options** from the model window's **Edit** or context menu, then choose **Disable link**. To restore a disabled link, choose **Restore link** from the **Link Options** menu.

Modifying a Linked Subsystem

Simulink allows you to modify subsystems that are library links. If your modifications alter the structure of the subsystem, you must disable the link from the reference block to the library block . If you attempt to modify the structure of a subsystem link, Simulink prompts you to disable the link. Examples of structural modifications include adding or deleting a block or line or change the number of ports on a block. Examples of nonstructural changes include changes to parameter values that do not affect the structure of the subsystem.

Propagating Link Modifications

Simulink allows a model to have active links with nonstructural but not structural changes. If you restore a link that has structural changes, Simulink prompts you to either propagate or discard the changes. If you choose to propogate the changes, Simulink updates the library block with the changes

made in the reference block. If you choose to discard the changes, Simulink replaces the modified reference block with the original library block. In either case, the end result is that the reference block is an exact copy of the library block.

If you restore a link with nonstructural changes, Simulink enables the link without prompting you to propagate or discard the changes. If you want to propagate or discard the changes at a later time, select the reference block, choose **Link options** from the model window's **Edit** or context menu, then choose **Propagate/Discard changes**. If you want to view the nonstructural parameter differences between a reference block and its corresponding library block, choose **View changes** from the **Link options** menu.

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 "Library Link Status" on page 4-81)
- 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, first disable the block. Then select the block and choose **Break Library Link** from the **Link options** menu. You can also break the link between a reference block and its library block from the command line by changing the value of the LinkStatus parameter to 'none' using this command.

```
set_param('refblock', 'LinkStatus', 'none')
```

You can 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 **Link options** submenu of the model window's **Edit** or context 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.

Library Link Status

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

Status	Description
none	Block is not a reference block.
resolved	Link is resolved.
unresolved	Link is unresolved.
implicit	Block is within a linked block.
inactive	Link is disabled.

Displaying Library Links

Simulink optionally displays an arrow in the bottom left corner of each icon that represents a library link in a model.
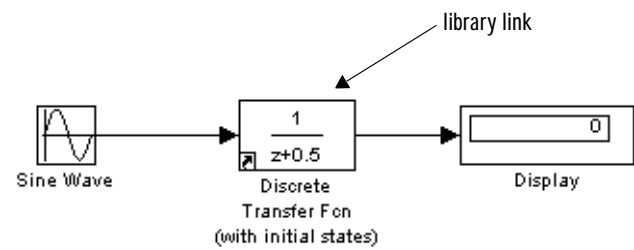
This arrow allows you to tell at a glance whether an icon represents a link to a library block or a local instance of a block. To enable display of library links, select **Library Link Display** from the model window's **Format** menu and then select either **User** (displays only links to user libraries) or **All** (displays all links).

The color of the link arrow indicates the status of the link.

Color	Status
Black	Active link
Grey	Inactive link
Red	Active and modified

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. To display the Library Browser, click the **Library Browser** button in the toolbar of the MATLAB desktop or Simulink model window or type `simulink` at the MATLAB command line.

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

The Library Browser contains three panes.

Documentation Pane
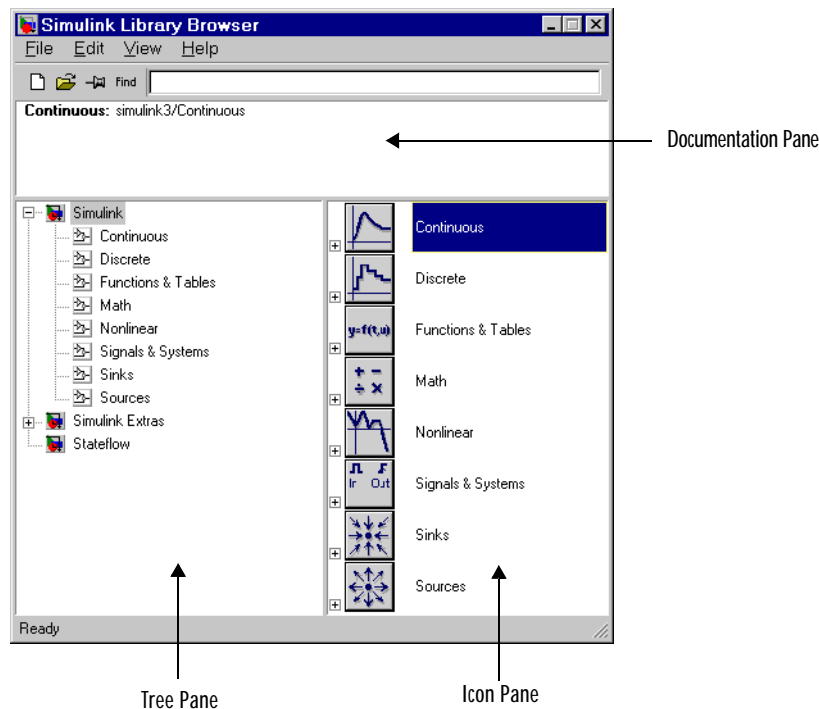
Tree Pane

Icon Pane

The tree pane displays all the block libraries installed on your system. The icon pane displays the icons of the blocks that reside in the library currently seleted in the tree pane. The documentation pane displays documentation for the block selected in the icon pane.

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.

Adding Libraries to the Library Browser

If you want a library that you have created to appear in the Library Browser, you must create an slblocks.m file that describes the library in the directory that contains it. The easiest way to create an slblocks.m file is to use an existing slblocks.m file as a template. You can find all existing slblocks.m files on your system by typing

```
which('slblocks.m', '-all')
```

at the MATLAB command prompt. Copy any of the displayed files to your library's directory. Then, open the copy, edit it, following the instructions included in the file, and save the result. Finally, add your library's directory to the MATLAB path, if necessary. The next time you open the Library Browser, your library should appear among the libraries displayed in the browser.

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.
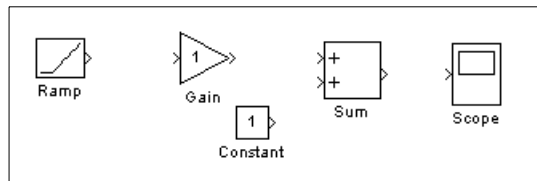
Assign parameter values to the Gain and Constant blocks by opening (double-clicking on) each block and entering the appropriate value. Then, click on the **Close** button to apply the value and close the dialog box.

Now, connect the blocks.
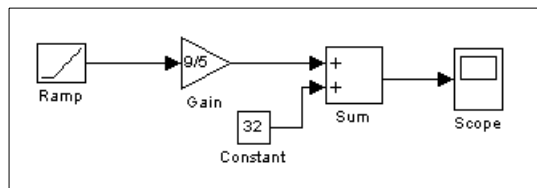
The Ramp block inputs Celsius temperature. Open that block and change the **Initial output** parameter to 0. The Gain block multiplies that temperature by the constant 9/5. The Sum block adds the value 32 to the result and outputs the Fahrenheit temperature.

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 4–23. Now you can connect all the blocks.
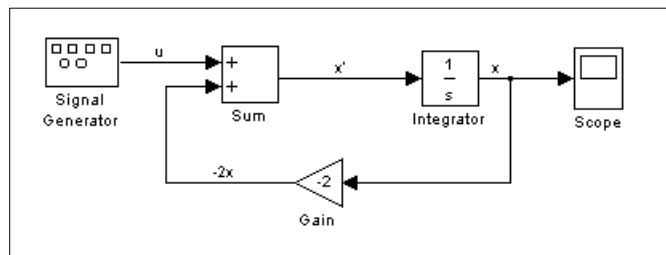
An important concept in this model is the loop that includes the Sum block, the Integrator block, and the Gain block. In this equation, *x* is the output of the Integrator block. It is also the input to the blocks that compute *x'*, on which it is based. This relationship is implemented using a loop.

The Scope displays *x* at each time step. For a simulation lasting 10 seconds, the output looks like this.
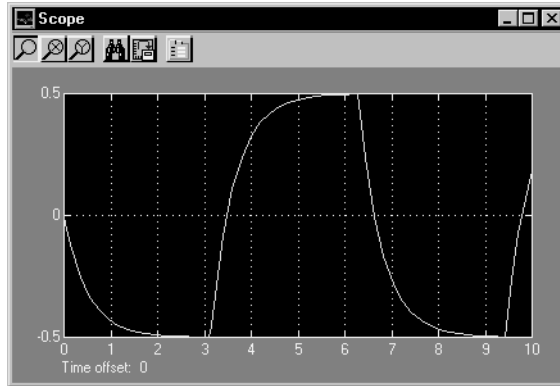
The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts *u* as input and outputs *x*. So, the block implements *x/u*. If you substitute *sx* for *x′* in the above equation, you get

$$sx = -2x + u$$

Solving for *x* gives

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

or,

$$x/u = 1/(s+2)$$

The Transfer Fcn block uses parameters to specify the numerator and denominator coefficients. In this case, the numerator is 1 and the denominator is s+2. Specify both terms as vectors of coefficients of successively decreasing powers of s. In this case the numerator is [1] (or just 1) and the denominator is [1 2]. The model now becomes quite simple.
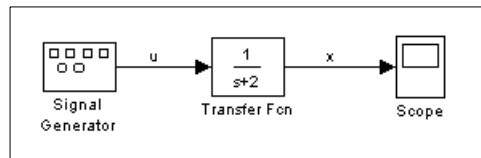
The results of this simulation are identical to those of the previous model.

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, "Model File Format."

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.
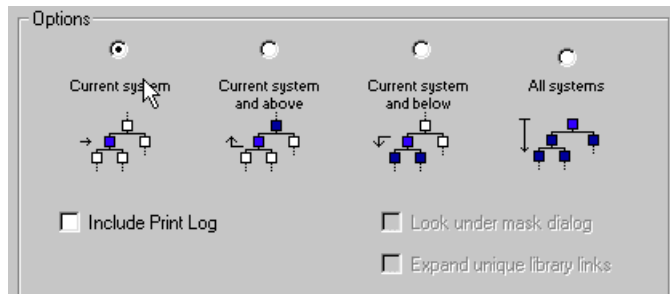
When you select either the **Current system and below** or **All systems** option, two check boxes become enabled. In this figure, **All systems** is selected.
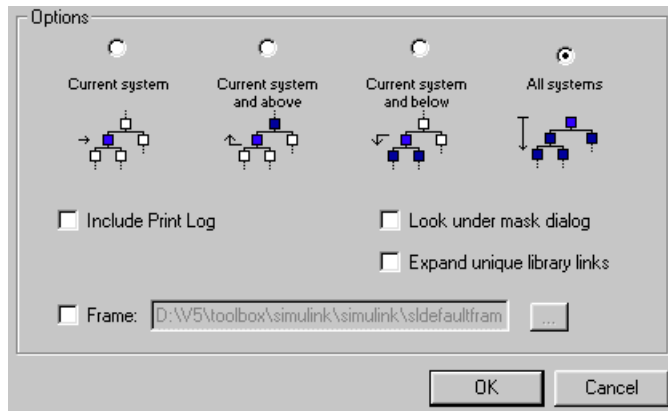
Selecting the **Look Under Mask Dialog** check box prints the contents of masked subsystems when encountered at or below the level of the current block. When printing all systems, the top-level system is considered the current block so Simulink looks under any masked blocks encountered.

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 4-77.

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
```

4-91

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'])
```

To print the currently selected subsystem, enter

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

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 and Block Parameters" on page A-1), using the set_param command. You can set the paper orientation alone, using MATLAB's orient command. On Windows,

the **Print** and **Printer Setup** dialog boxes 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.

Searching and Browsing Models

Simulink provides you with tools for searching and browsing models. These can be useful when you need to view or modify an object but do not know where it is located.

Searching for Objects

To find a block, signal, state, or other object in a model, select **Find** from Simulink's **Edit** menu. Simulink displays the **Find** dialog box.
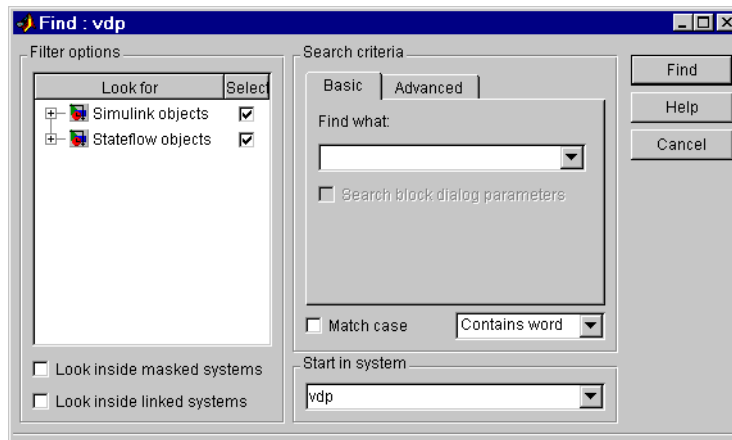
To find an object with the **Find** dialog box, first use the **Filter options** (see "Filter Options" on page 4-96) and **Search criteria** (see "Search Criteria" on page 4-96) panels to specify the characteristics of the object you want to find. Next, if you have more than one system or subsystem open, select the system or subsystem where you want the search to begin from the **Start in system** list. Finally, select the **Find** button. Simulink searches the selected system for objects that meet the criteria you have specified. Any objects that satisfy the criteria appear in the results panel at the bottom of the **Find** dialog box.
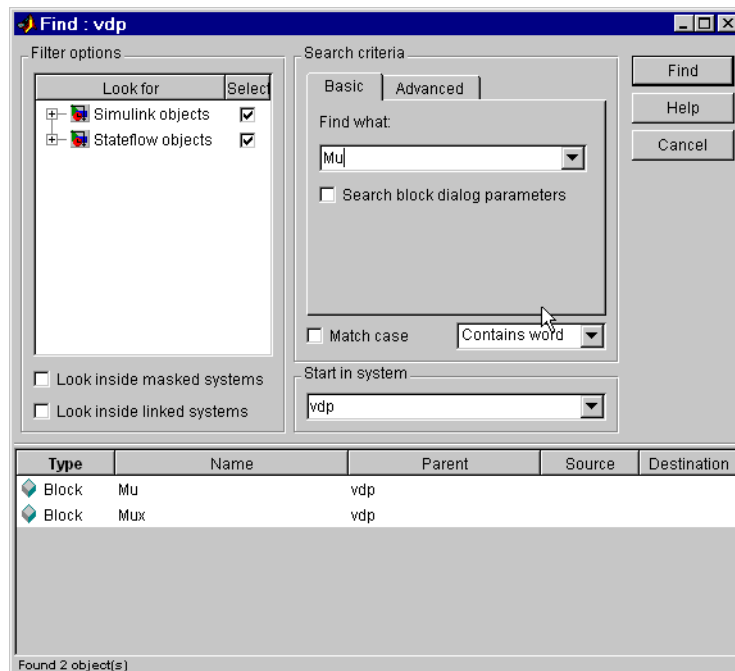
You can display an object by double-clicking its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object. To sort the results list, click any of the buttons at the top of each column. For example, to sort the results by object type, click the **Type** button. Clicking a button once sorts the list in ascending order, clicking it twice sorts it in descending order. To display an object's parameters or properties, select the object in the list. Then press the right mouse button and select **Parameter** or **Properties** from the resulting context menu.

Filter Options

The **Filter options** panel allows you to specify what kinds of objects to look for and where to search for them.
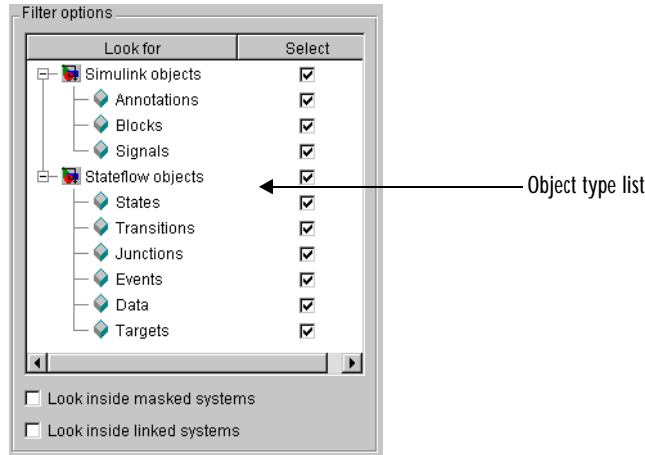
Object type list

Object type list. The object type list lists the types of objects that the Simulink can find. By unchecking a type, you can exclude it from the Finder's search.

Look inside masked subsystem. Checking this option causes Simulink to look for objects inside of masked subsystems.
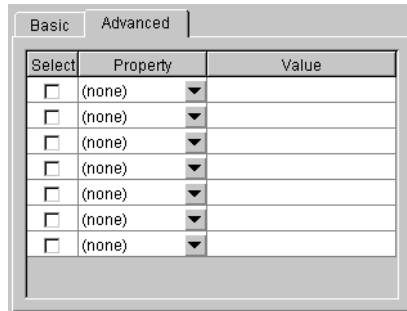
Look inside linked systems. Checking this option causes Simulink to look for objects inside subsystems linked to libraries.

Search Criteria

The **Search criteria** panel allows you to specify the criteria that objects must meet to satisfy your search request.

Basic. The **Basic** panel allows you to search for objects whose name and, optionally, dialog parameters match a specified text string. Enter the search text in the panel's **Find what** field. To display previous search text, select the dropdown list button next to the **Find what** field. To reenter text, click it in the dropdown list. Check **Search block dialog parameters** if you want dialog parameters to be included in the search.

Advanced. The **Advanced** panel allows you to specify a set of as many as seven properties that an object must have to satisfy your search request.

To specify a property, type its name in one of the cells in the **Property** column of the **Advanced** pane or select the property from the cell's property list. To display the list, select the down arrow button next to the cell. Next enter the value of the property in the **Value** column next to the property name. When you enter a property name, the **Finder** checks the check box next to the property name in the **Select** column. This indicates that the property is to be included in the search. If you want to exclude the property, uncheck the check box.

Match case. Check this option if you want Simulink to consider case when matching search text against the value of an object property.

Other match options. Next to the **Match case** option is a list that specifies other match options that you can select.

• Match whole word

Specifies a match if the property value and the search text are identical except possibly for case.

• Contains word

Specifies a match if a property value includes the search text.

- Regular expression

  Specifies that the search text should be treated as a regular expression when matched against property values. The following characters have special meanings when they appear in a regular expression.

Character	Meaning
^	Matches start of string.
$	Matches end of string.
.	Matches any character.
\	Escape character. Causes the next character to have its ordinary meaning. For example, the regular expression \. . matches . a and . 2 and any other wo-character string that begins with a period.
*	Matches zero or more instances of the preceding character. For example, ba* matches b, ba, baa, etc.
+	Matches one or more instances of the preceding character. For example, ba+ matches ba, baa, etc.
[ ]	Indicates a set of characters that can match the current character. A hyphen can be used to indicate a range of characters. For example, [a-zA-Z0-9_]+ matches foo_bar1 but not foo$bar. A ^ indicates a match when the current character is not one of the following characters. For example, [^0-9] matches any character that is not a digit.
\w	Matches a word character (same as [a-z_A-Z0-9]).
\W	Matches a nonword character (same as [^a-z_A-Z0-9]).
\d	Matches a digit (same as [0-9]).
\D	Matches a nondigit (same as [^0-9]).
\s	Matches white space (same as [\t\r\n\f]).

Character	Meaning
\S	Matches nonwhite space (same as [^ \t\r\n\f]).
\<WORD\>	Matches WORD where WORD is any string of word characters surrounded by white space.

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
- Use your source control system to manage the model. Refer to "Interfacing with Source Control Systems" in the MATLAB documentation.

The browser operates differently on Microsoft Windows and UNIX platforms.

Using the Model Browser on Windows

To display the Model Browser, 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.

Note The **Browser initially visible** preference causes Simulink to open models by default in the Model Browser. To set this preference, select **Preferences** from the Simulink **File** menu.
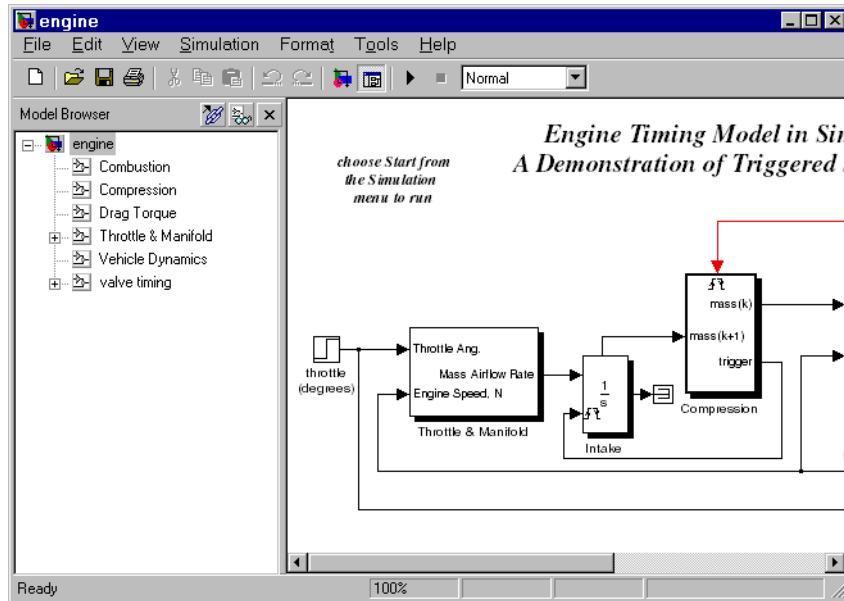
The top entry in the tree view corresponds to your model. A button next to the model name allows you to expand or contract the tree view. The expanded view shows the model's subsystems. A button next to a subsystem indicates that the subsystem itself contains subsystems. You can use the button to list the subsystem's children. To view the block diagram of the model or any subsystem displayed in the tree view, select the subsystem. You can use either the mouse or the keyboard to navigate quickly to any subsystem in the tree view.

Navigating with the Mouse. Click any subsystem visible in the tree view to select it. Click the + button next to any subsystem to list the subsystems that it contains. Click the button again to contract the entry.

Navigating with the Keyboard. Use the up/down arrows to move the current selection up or down the tree view. Use the left/right arrow or +/- keys on your numeric keypad to expand an entry that contains subsystems.

Showing Library Links. The Model Browser can include or omit library links from the tree view of a model. Use the Simulink **Preferences** dialog box to specify whether to display library links by default. To toggle display of library links,

select **Show library links** from the **Model browser options** submenu of the
Simulink **View** menu.

Showing Masked Subsystems. The Model Browser can include or omit masked
subsystems from the tree view. If the tree view includes masked subsystems,
selecting a masked subsystem in the tree view displays its block diagram in the
diagram view. Use the Simulink **Preferences** dialog box to specify whether to
display masked subsystems by default. To toggle display of masked
subsystems, select **Look under masks** from the **Model browser options**
submenu of the Simulink **View** menu.

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.
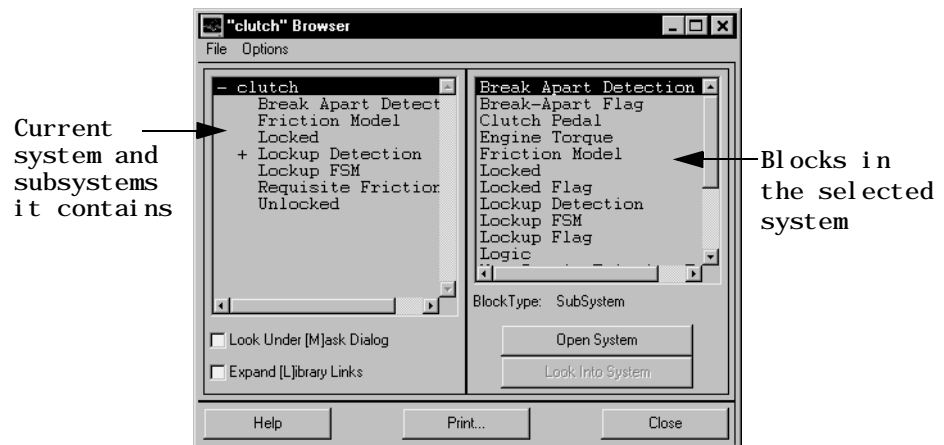
Contents of the Browser Window

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 the menu items **Open System**, **Look Into System**, **Display Alphabetical/Hierarchical List**, **Expand All**, **Look Under Mask Dialog**, and **Expand Library Links**.
- The **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 0penFcn 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 7, "Using Masks to Customize Blocks."
- [L] indicates that the block is a reference block. For more information, see "Connecting Blocks" on page 4-22.
- [O] indicates that an open function (0penFcn) callback is defined for the block. For more information about block callbacks, see "Using Callback Routines" on page 4-70.
- [S] indicates that the system is a Stateflow 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.

Managing Model Versions

Simulink has features that help you to manage multiple versions of a model.

- As you edit a model, Simulink generates version control information about the model, including a version number, who created and last updated the model, and an optional change history. Simulink saves the automatically generated version control information with the model. See "Version Control Properties" on page 4-111 for more information.

- The Simulink **Model Parameters** dialog box lets you edit some of the version control information stored in the model and select various version control options (see "Model Properties Dialog" on page 4–106).

- The Simulink Model Info block lets you display version control information, including information maintained by an external version control system, as an annotation block in a model diagram (see "Model Info" on page 9-162).

- Simulink version control parameters let you access version control information from the MATLAB command line or an M-file.

- The **Source Control** submenu of the Simulink **File** menu allows you to check models into and out of your your source control system. See "Interfacing with Source Control Systems," in the MATLAB documentation for more information.

Specifying the Current User

When you create or updates a model, Simulink logs your name in the model for version control purposes. Simulink assumes that your name is specified by at least one of the following environment variables: USER, USERNAME, LOGIN, or LOGNAME. If your system does not define any of these variables, Simulink does not update the user name in the model.

UNIX systems define the USER environment variable and set its value to the name you use to log on to 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 is connected to a network. Use the MATLAB command getenv to determine which of the environment variables is defined. For example, enter

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

at the MATLAB command line to determine whether the USER environment variable exists on your Windows system. If not, you must set it yourself. On Windows 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. Save the file autoexec.bat and reboot your computer for the changes to take effect.

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

On Windows NT, use the Environment pane of the Windows NT **System Properties** dialog box to set the USER environment variable (if it is not already defined).
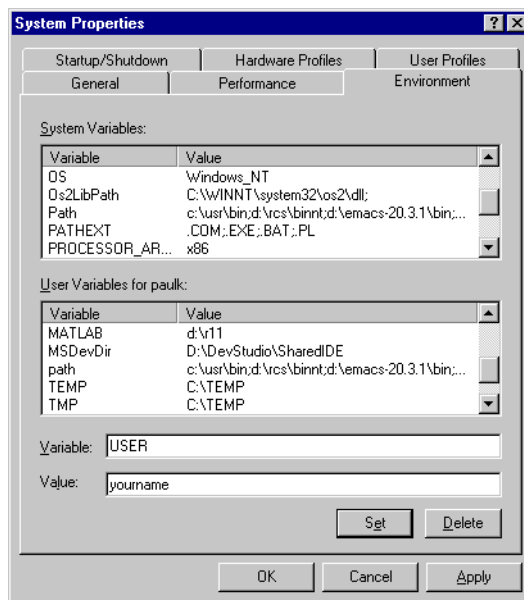
To display the **System Properties** dialog box, select **Start**->**Settings**->**Control Panel** to open the Control Panel. Double-click the **System** icon. To set the USER variable, type USER in the **Variable** field and type your login name in the **Value** field. Click **Set** to save the new environment variable. Then click **OK** to close the dialog box.

Model Properties Dialog

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 Properties** from the Simulink **File** menu.
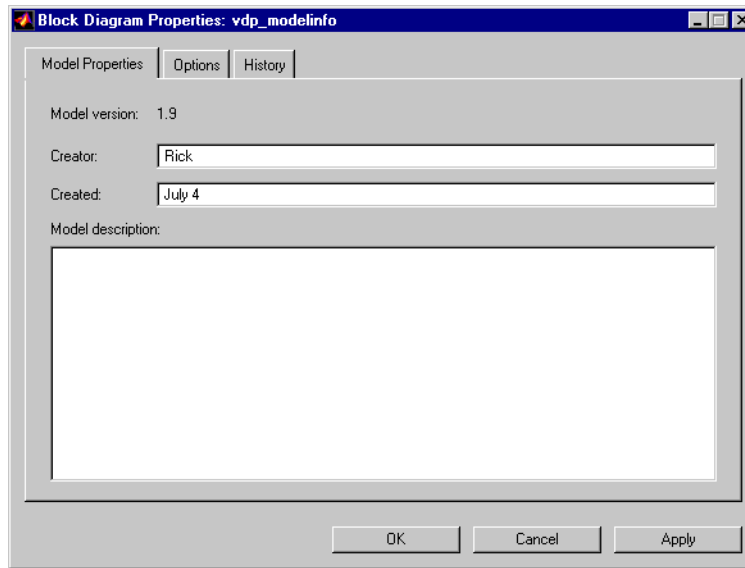
Model Properties Pane

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 indicate a configuration manager and specify version control information formats.
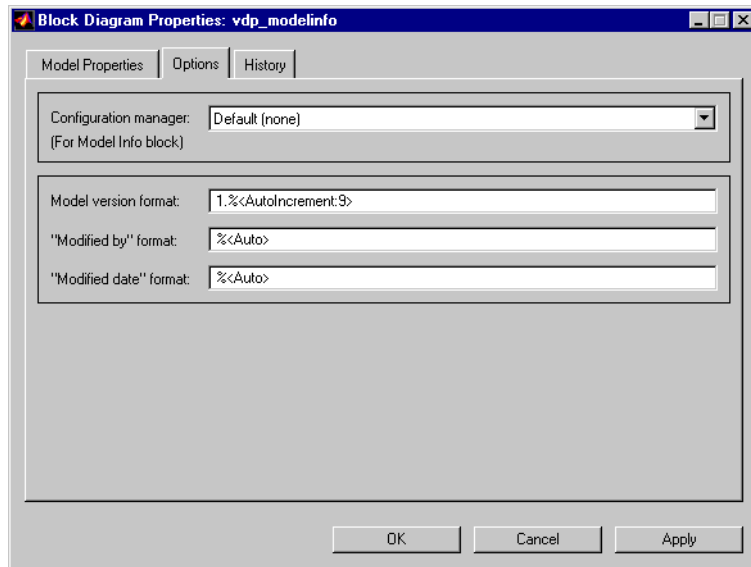
Configuration manager. Identifies the external configuration manager used to manage this model. Selecting a configuration manager from the list allows you to include information from the configuration manager in a Model Info block for annotation. Setting this option does not determine or activate configuration management for the model. The default **Configuration manager** setting is Default(none), indicating that information for a Model Info block is not available from a configuration management system. See "Model Info" on page 9-162 for more information.

Model version format. Format used to display the model version number in the Model Properties pane and in Model Info blocks. The value of this parameter can be any text string. The text string can include occurrences of the tag %<AutoIncrement:#> where # is an integer. Simulink replaces the tag with an

integer when displaying the model's version number. For example, it displays the tag

    1.%<AutoIncrement: 2>

as

    1.2

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

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

becomes

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

and Simulink reports the model version number as 1.3.

"Modified by" format. Format used to display the **Last 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 to enable, view, and edit this model's change history.
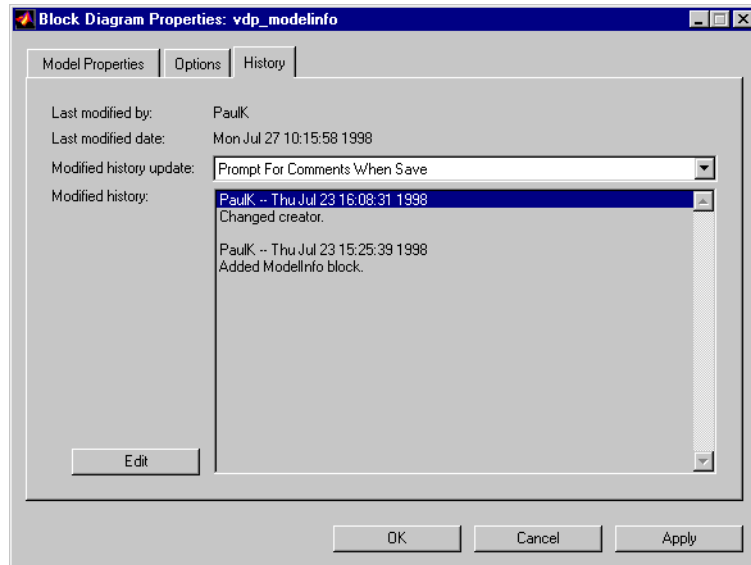
Last modified by. Name of the person who last modified this model. Simulink sets the value of this parameter to the value of the USER environment variable when you save a model. You cannot edit this field.

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 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 4–110 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 clicking **Edit**.

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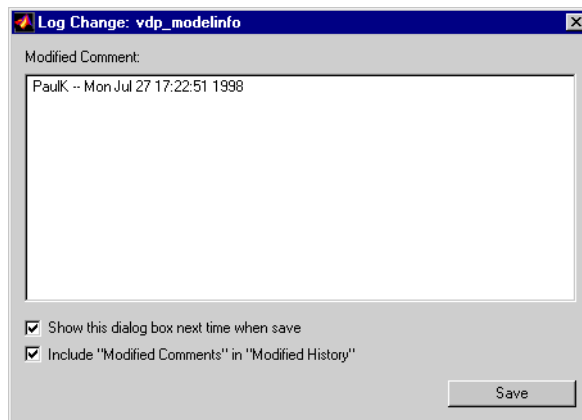
To add an item to the model's change history, enter the item in the **Modified Comments** edit field and click **Save**. If you do not want to enter an item for this session, clear the **Include "Modified Contents" in "Modified History"** option. To discontinue change logging, clear the **Show this dialog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

Editing the Change History

To edit the change history for a model, click the **Edit** button on the History pane of the Simulink **Model Properties** dialog box. Simulink displays the model's history in a **Modification History** dialog box.
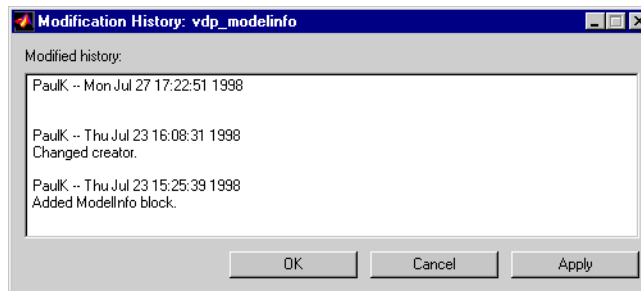
Edit the history displayed in the dialog and select **Apply** or **OK** to save the changes.

Version Control Properties

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

Property	Description
Created	Date created.
Creator	Name of the person who created this model.
ModifiedBy	Person who last modified this model.
ModifiedByFormat	Format of the ModifiedBy parameter. Value can be an string. The string can include the tag %<Auto>. Simulink replaces the tag with the current value of the USER environment variable.

Property	Description
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**

5

Running a Simulation

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 allows you to change simulation and block parameters iteratively. For more information, see "Running a Simulation from the Command Line" on page 5–36.

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 "panes" to manage simulation parameters:

- The **Solver** pane 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** pane manages input from and output to the MATLAB workspace.
- The **Diagnostics** pane allows you to select the level of warning messages displayed during a simulation.

Each pane of the dialog box, including the parameters you set on the pane, is discussed in detail in "The Simulation Parameters Dialog Box" on page 5–8.

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.

Click to display
error message.

Double-click to
display error
source.

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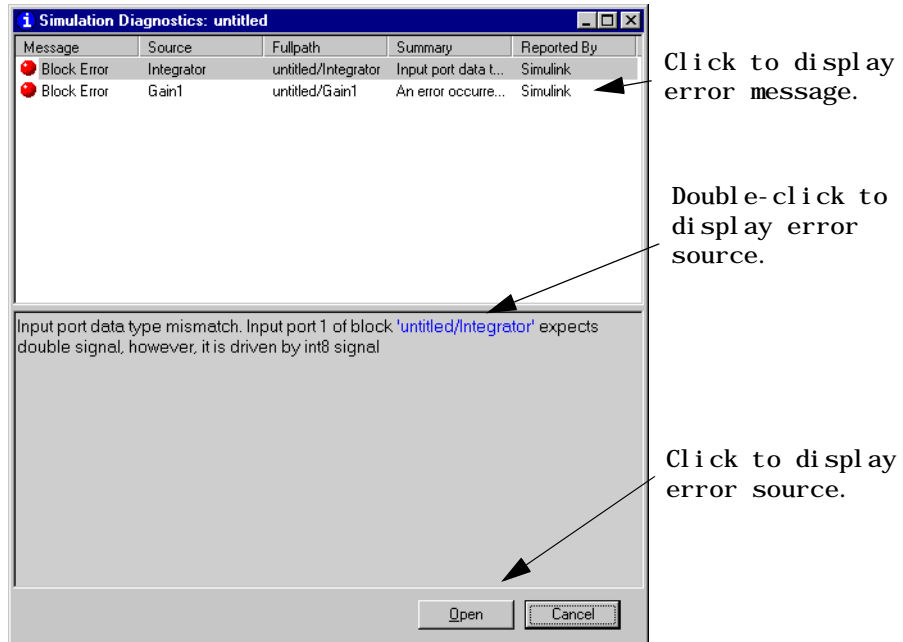
The dialog box has two panes. The upper pane consist of columns that display
the following information for each error.

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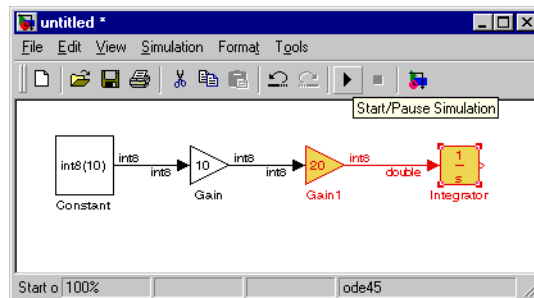
You can similarly display other error sources by double-clicking on the corresponding error message in the top pane, by double-clicking on the name of the error source in the error message (highlighted in blue), or by selecting the **Open** button on the dialog box.

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 5-37) and simset (see simset on page 5-41) commands. Parameters are described as they appear on the dialog box panes.

This table summarizes the actions performed by the dialog box buttons that appear on the bottom of each dialog box pane.

Table 5-1: Simulation Parameters Dialog Box Buttons

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

The Solver Pane

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

The **Solver** pane allows you to:

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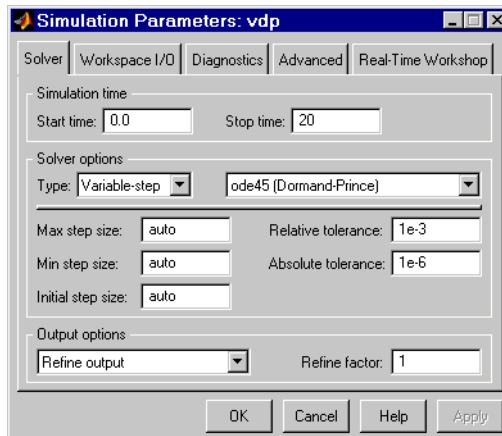
Simulation Time

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 the MATLAB documentation.

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 5-10.

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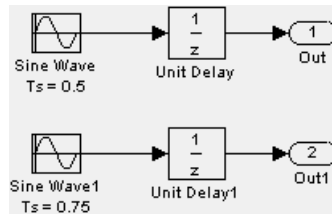
With sample times of 0.5 and 0.75, the *fundamental sample time* for the model is 0.25 second. The difference between the variable-step and the fixed-step discrete solvers is the time vector that each generates.

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 5–34.

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 5–34). You can tune the selected solver by changing parameter values on the **Solver** pane.

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 5–16.

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.

Minimum step size. Specifies the smallest time step the solver can take. If the solver needs to take a smaller step to meet error tolerances, it issues a warning indicating the current effective relative tolerance. This parameter can be either a real number greater than zero or a two-element vector where the first element is the minimum step size and the second element is the maximum number of minimum step size warnings to be issued before issuing an error. Setting the second element to zero results in an error the first time the solver must take a step smaller than the specified minimum. This is equivalent to changing the minimum step size violation diagnostic to error on the **Diagnostics** panel. Setting the second element to -1 results in an unlimited number of warnings. This is also the default if the input is a scalar. The default values for this parameter are a minimum step size on the order of machine precision and an unlimited number of warnings.

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

$$e_i \leq max(rtol \times |x_i|, atol_i)$$

The following figure shows a plot of a state and the regions in which the acceptable error is determined by the relative tolerance and the absolute tolerance.
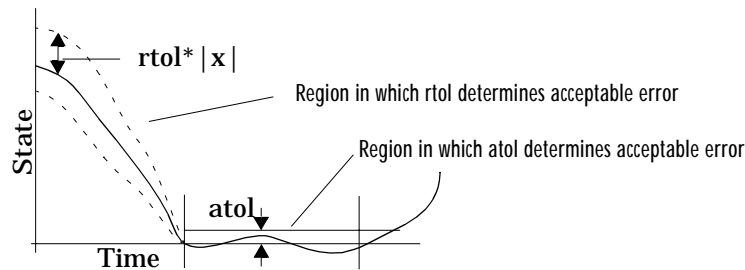
If you specify `auto` (the default), Simulink sets the absolute tolerance for each state initially to 1e-6. As the simulation progresses, Simulink resets the absolute tolerance for each state to the maximum value that the state has assumed thus far times the relative tolerance for that state. Thus, if a state goes from 0 to 1 and `reltol` is 1e-3, then by the end of the simulation the `abstol` is set to 1e-3 also. If a state goes from 0 to 1000, then the `abstol` is set to 1.

If the computed setting is not suitable, you can determine an appropriate setting yourself. You might have to run a simulation more than once to determine an appropriate value for the absolute tolerance. 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** pane 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 9-267) and Zero-Order Hold (see Zero-Order Hold on page 9-275). 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 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.

Note   This option will not help the solver to locate zero crossings (see "Zero Crossing Detection" on page 3-14).

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 **Ouput Times** field on the **Solver** pane. 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

In general, you should specify output points as integers times a fundamental step size, e.g.,

    [1: 100]*0. 01

is more accurate than

    [1: 0. 01: 100]

The Workspace I/O Pane

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 pane appears.
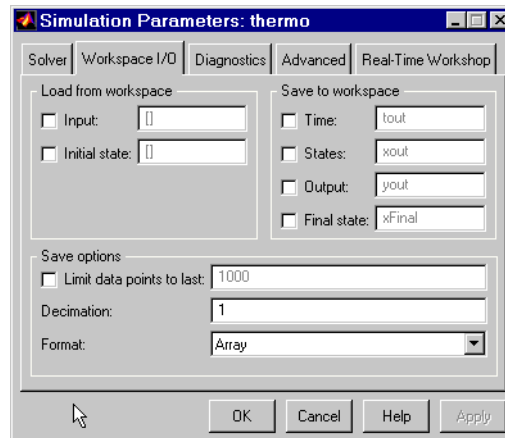
Loading Input from the Base Workspace

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** pane. Then, enter an external input specification (see below) in the adjacent edit box and select **Apply**.

The external (i.e., from workspace) input can take any of the following forms.

Array. To use this format, check **Input** in the **Load from workspace** pane and select the Matrix option from the **Format** list on the **Workspace I/O** pane. Selecting this option causes Simulink to evaluate the expression next to the **Input** check box and use the result as the input to the model.

The expression must evaluate to a real (noncomplex) matrix of data type double. The first column of the matrix must be a vector of times in ascending order. The remaining columns specify input values. In particular, each column represents the input for a different Inport block signal (in sequential order) and each row is the input value for the corresponding time point. 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 9-122).

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.

The default input expression for a model is [t, u] and the default input format is Matrix. So if you define t and u in the base workspace, you need only check the **Input** option to input data from the model's base workspace. For example, suppose that a model has two inports, one of which accepts two signals and the other of which accepts 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)];
```

Note The matrix input format allows you to load only real (noncomplex) scalar or vector data of type double. Use the structure format to input complex data, matrix (2-D) data, and/or data types other than double.

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 signals substructure must contain two fields named values and dimensions, respectively. The values field must contain an array of inputs for the corresponding input port where each input corresponds to a time point specified by the time field. The dimensions field specifies the dimension(s) of the input. If each input is a scalar or vector (1-D array) value, the dimensions field must be a scalar value that specifies the length of the vector (1 for a scalar). If each input is a matrix (2-D array), the dimensions field must be a two-element vector whose first element specifies the number of rows in the matrix and whose second element specifies the number of columns.

If the inputs for a port are scalar or vector values, the values field must be an M-by-N array where M is the number of time points specified by the time field and N is the length of each vector value. For example, the following code creates an input structure for loading 11 time samples of a two-element signal vector of type int8 into a model with a single input port.

```
a.time = (0:0.1:1)';
c1 = int8([0:1:10]');
c2 = int8([0:10:100]');
a.signals(1).values = [c1 c2];
a.signals(1).dimensions = 2;
```

To load this data into the model's inport, you would check the **Input** option on the **Workspace I/O** pane and enter a in the input expression field.

If the inputs for a port are matrices (2-D arrays), the values field must be an M-by-N-by-T array where M and N are the dimensions of each matrix input and T is the number of time points. For example, suppose that you want to input 51 time samples of a 4-by-5 matrix signal into one of your model's input ports. Then, the corresponding dimensions field of the workspace structure must equal [4 5] and the values array must have the dimensions 4-by-5-by-51.

As another example, consider the following model, which has two inputs.
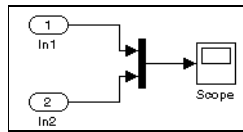
Suppose that you want to input a sine wave into the first port and a cosine wave into the second port. To do this, define a vector, a, as follows in the base workspace.

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

Then, check the **Input** box for this model, enter a in the adjacent text field, and select StructureWithTime as the I/O format.

Note Simulink can read back simulation data saved to the workspace in the **Structure with time** output format. See "Structure with time" on page 5-23 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 5-24 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.

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 pane. 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 the vector 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 9-251).

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 listed below.

Array. If you select this option, Simulink saves a model's states and outputs in a state and output array, respectively.

The state matrix has the name specified in the **Save to Workspace** area (for example, xout). Each row of the state matrix corresponds to a time sample of the model's states. Each column corresponds to an element of a state. For example, suppose that your model has two continuous states, each of which is a two-element vector. Then the first two elements of each row of the state matrix contains a time sample of the first state vector. The last two elements of each row contain a time sample of the second state vector.

The model output matrix has the name specified in the **Save to Workspace** area (for example, yout). Each column corresponds to a model outport, each row to the outputs at a specific time.

Note You can use array format to save your model's outputs and states only if the outputs are either all scalars or all vectors (or all matrices for states), are either all real or all complex, and are all of the same data type. Use the Structure or StructureWithTime output formats (see the following) if your model's outputs and states do not meet these conditions.

Structure with time. If you select this format, Simulink saves the model's states and outputs in structures having the names specified in the **Save to Workspace** area (for example, xout and yout).

The structure used to save outputs has two top-level fields: time 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 outport. Each substructure has four fields: values, dimensions, label, and blockName. The values field contains the outputs for the corresponding outport. If the outputs are scalars or vectors, the values field is a matrix each of whose rows represents an output at the time specified by the corresponding element of the time vector. If the outputs are matrix (2-D) values, the values field is a 3-D array of dimensions M-by-N-by-T where M-by-N is the dimensions

of the output signal and T is the number of output samples. If T = 1, MATLAB drops the last dimension. Therefore, the value field will be an M-by-N matrix. The dimensions field specifies the dimensions of the output signal. The label field specifies the label of the signal connected to the outport or the type of state (continuous or discrete). The blockName field specifies the name of the corresponding outport or block with states.

The structure used to save states has a similar organization. The states 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 one of the model's states. Each signals structure has four fields: values, dimension, label, and blockName. The values field contains time samples of a state of the block specified by the blockName field. The label field for built-in blocks indicates the type of state: either CSTATE (continuous state) or DSTATE (discrete state). For S-Function blocks, the label contains whatever name is assigned to the state by the S-Function block.

The time samples of a state are stored in the values field as a matrix of values. Each row corresponds to a time sample. Each element of a row corresponds to an element of the state. If the state is a matrix, the matrix is stored in the values array in column-major order. For example, suppose that the model includes a 2-by-2 matrix state and that Simulink logs 51 samples of the state during a simulation run . Then the values field for this state would contain a 51-by-4 matrix where each row corresponds to a time sample of the state and where the first two elements of each row corresponds to the first column of the sample and the last two elements corresponds to the second column of the sample.

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 data samples saved, select the check box labeled **Limit data points to last** and specify the number of samples to save. To apply

a decimation factor, enter a value in the field to the right of the **Decimation** label. For example, a value of 2 saves every other point generated.

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 pane.

You can also save the final states for the current simulation run and apply them to a subsequent simulation run. 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 pane.

To save the final states (the values of the states at the termination of the simulation), select the **Final State** check box and enter a variable in the adjacent edit field.

To load states, select the **Initial State** check box and specify 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 array is empty ([ ]), Simulink uses the initial conditions defined in the blocks.

The Diagnostics Pane

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.
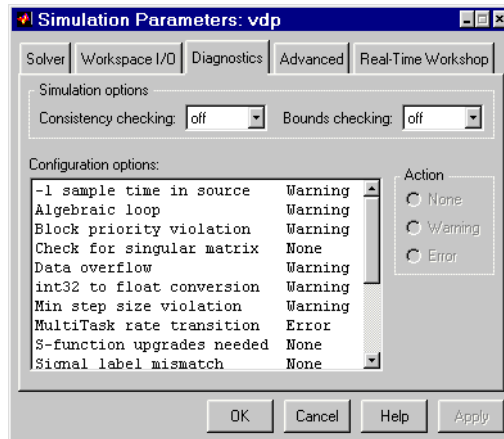
The dialog box includes the following options.

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

Bounds Checking

This option causes Simulink to check whether a block writes outside the memory allocated to it during simulation. Typically this can happen only if your model includes a user-written S-function that has a bug. If enabled, this check is performed for every block in the model every time the block is executed. As a result, enabling this option slows down model execution considerably. Thus, to avoid slowing down model execution needlessly, you should enable the option only if you suspect that your model contains a user-written S-function that has a bug. See *Writing S-Functions* for more information on using this option.

Configuration options

This control lists abnormal types of events that can occur during execution of the model 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.

Event	Description
`-1 sample time in source`	A source block (e.g., a Sine Wave block) specifies a sample time of -1.
`Algebraic loop`	Simulink detected an algebraic loop while simulating the model. See "Algebraic Loops" on page 3–18 for more information.

Event	Description (Continued)
Check for singular matrix	The Product block detected a singular matrix while inverting one of its inputs in matrix multiplication mode (see Product on page 9-178).
Data overflow	The value of a signal or parameter is too large to be represented by the signal or parameter's data type. See "Working with Data Types" on page 4-44 for more information.
int32 to float conversion	A 32-bit integer value was converted to a floating-point value. Such a conversion can result in a loss of precision. See "Working with Data Types" on page 4-44 for more information.
Min step size violation	The next simulation step is smaller than minimum step size specified for the model. This can occur if the specified error tolerance for the model requires a step size smaller than the specified minimum step size. See "Step Sizes" on page 5–12 and "Error Tolerances" on page 5–13 for more information.
Multitask rate transition	An invalid rate transition occurred between two blocks operating in multitasking mode (see "Multitasking Options" on page 5-15).
S-function upgrades needed	A block was encountered that has not been upgraded to use features of the current release.
Signal label mismatch	The simulation encountered virtual signals that have a common source signal but different labels (see "Virtual Signals" on page 4–29).
SingleTask rate transition	A rate transition occurred between two blocks operating in single-tasking mode (see "Multitasking Options" on page 5-15).
Unconnected block input	Model contains a block with an unconnected input.

Event	Description (Continued)
`Unconnected block output`	Model contains a block with an unconnected output.
`Unconnected line`	Model contains an unconnected line.
`Unneeded type conversions`	A data type conversion block is used where no type conversion is necessary. See `Data Type Conversion` on page 9-49 for more information.
`Vector/Matrix conversion`	A vector-to-matrix or matrix-to-vector conversion occurred at a block input (see "Vector or Matrix Input Conversion Rules" on page 4–34).
`Block Priority Violation`	Simulink detected a block priority specification error while simulating the model.

The Advanced Pane

The **Advanced** pane allows you to set various options that affect simulation performance.
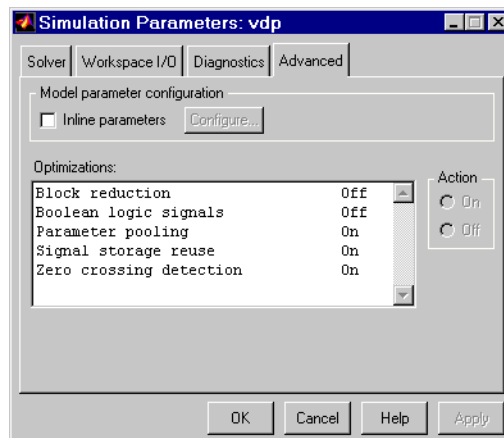
Model parameter configuration

Inline parameters. By default you can modify ("tune") many block parameters during simulation (see "Tunable Parameters" on page 3–5). Selecting this option makes all parameters nontunable except those that you specify. Making parameters nontunable enables Simulink to treat them as constants, thereby speeding up simulation. Using the **Model Parameter Configuration** dialog box (see "Model Parameter Configuration Dialog Box" on page 5-32) to specify the parameters you want to remain tunable when this option is selected. To display the dialog, select the adjacent **Configure** button.

When this option is selected, the only parameters that you can change during simulation are parameters that meet the following conditions:

- The value of the parameter must be a variable defined in the MATLAB workspace.
- The parameter must be specified as global (tunable) in the **Model Parameter Configuration** dialog box.

To tune a parameter that meets these conditions, change the value of the corresponding workspace variable and choose **Update Diagram** (**Ctrl+D)** from the Simulink **Edit** menu.

If you select this option, Simulink moves constant signals out of the simulation loop. This speeds up the simulation.

Optimizations

Block reduction. Replaces a group of blocks with a synthesized block, thereby speeding up execution of the model.

Boolean logic signals. Causes blocks that accept Boolean signals to require Boolean signals. If this option is off, blocks that accept inputs of type bool ean also accept inputs of type doubl e. For example, consider the following model.
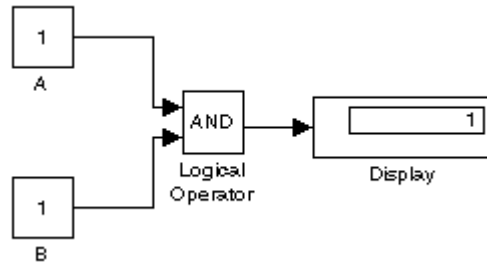
This model connects signals of type doubl e to a Logical Operator block, which accepts inputs of type bool ean. If **Boolean logic signals** option is on, this model generates an error when executed. If **Boolean logic signals** option is off, this model runs without error.

Note This option allows the current version of Simulink to run models that were created by earlier versions of Simulink that supported only signals of type doubl e.

Parameter pooling. This option is used for code generation (see the Real-Time Workshop documentation for more information). Leave this option on if you are not doing code generation.

Signal storage reuse. Causes Simulink to reuse memory buffers allocated to store block input and output signals. If this option is off, Simulink allocates a separate memory buffer for each block's outputs. 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 signal storage reuse if you need to:

• Debug a C-MEX S-function
• Use a floating Scope or Display block to inspect signals in a model that you are debugging

5-31

Simulink opens an error dialog if **Signal storage reuse** is enabled and you attempt to use a floating Scope or Display block to display a signal whose buffer has been reused.

Zero-crossing detection. Enables zero crossing detection during variable-step simulation of the model. For most models, this speeds up simulation by enabling the solver to take larger time steps. If a model has extreme dynamic changes, disabling this option can speed up the simulation but can also decrease the accuracy of simulation results. See "Zero Crossing Detection" on page 3–14 for more information.

Model Parameter Configuration Dialog Box

The **Model Parameter Configuration** dialog box allows you to override the **Inline parameters** option (see "Model parameter configuration" on page 5–30) for selected parameters.
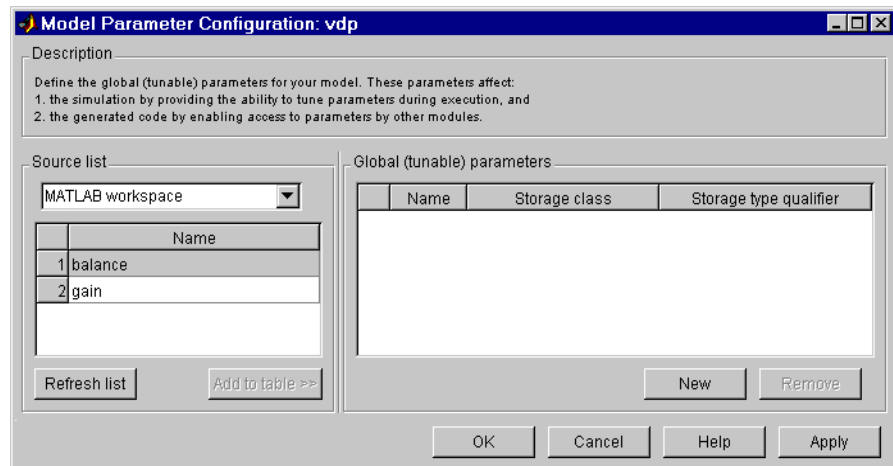
The dialog box has the following controls.

Source list. Displays a list of workspace variables. The options are:

- MATLAB workspace
  List all variables in the MATLAB workspace that have numeric values.
- Referenced workspace variables
  List only those variables referenced by the model.

Refresh list. Updates the source list. Click on this button if you have added a variable to the workspace since the last time the list was displayed.

Add to table. Adds the variable(s) selected in the source list to the adjacent table of tunable parameters.

New. Defines a new parameter and adds it to the list of tunable parameters. Use this button to create tunable parameters that are not yet defined in the MATLAB workspace.

Note This option does not create the corresponding variable in the MATLAB workspace. You must create the variable yourself.

Storage Class. Used for code generation. See the Real-Time Workshop documentation for more information.

Storage type qualifier. Used for code generation. See the Real-Time Workshop documentation for more information.

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 5–13.

- 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 3-18.
- 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 5-37. 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 5-41). The simulation parameters are discussed in "The Simulation Parameters Dialog Box" on page 5-8.

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).

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 executes a Simulink model, using all simulation parameter dialog settings including Workspace I/O options.

You can supply a null ([ ]) matrix for any right-side argument except the first (the model name). The sim command uses default values for unspecified arguments and arguments specified as null matrices. The default values are the values specified by the model. 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 do not specify the left side arguments, the command logs the simulation data specified by the **Workspace I/O** pane of the **Simulation parameters** dialog box (see "The Workspace I/O Pane" on page 5-18).

If you want to simulate a continuous system, you must specify the solver parameter, using simset (see simset on page 5-41). The solver defaults to VariableStepDiscrete for purely discrete models.

Arguments

t	Returns the simulation's time vector.
x	Returns the simulation's state matrix consisting of continuous states followed by discrete states.
y	Returns the simulation's output matrix. Each column contains the output of a root-level Outport block, in port number order. If any Outport block has a vector input, its output takes the appropriate number of columns.
y1, . . . , yn	Each y_i returns the output of the corresponding root-level Outport block for a model that has n such blocks.
model	Name of a block diagram.

timespan	Simulation start and stop time. Specify as one of these: tFinal to specify the stop time. The start time is 0. [tStart tFinal] to specify the start and stop times. [tStart OutputTimes tFinal] to specify the start and stop times and time points to be returned in t. Generally, t will include more time points. OutputTimes is equivalent to choosing **Produce additional output** on the dialog box.
options	Optional simulation parameters specified as a structure created by the simset command (see simset on page 5-41).
ut	Optional external inputs to top-level Inport blocks. ut can be a 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, ut1, ut2, ..., each of which corresponds to a specific port. Tabular input for all ports may be in the form of a MATLAB array 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 5-19 for a description of the array 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

Purpose   Plot simulation data in a figure window.

Syntax   simplot(data);
                   simplot(time, data);

Description   The simplot command plots output from a simulation in a Handle Graphics
                   figure window. The plot looks like the display on the screen of a Scope block.
                   Plotting the output on a figure window allows you to annotate and print the
                   output.

Arguments   data   Data produced by one of Simulink's output blocks (for example,
                               a root-level Outport block or a To Workspace block) or in one of
                               the output formats used by those blocks: **Array**, **Structure**,
                               Structure with time (see "The Workspace I/O Pane" on page
                               5-18).

                   time   The vector of sample times produced by an output block when
                               you have selected **Array** or **Structure** as the simulation's
                               output format. The simplot command ignores this argument if
                               the format of the data is **Structure with time**.

simplot

Examples   The following sequence of commands

```
vdp
set_param(gcs, 'SaveOutput', 'on')
set_param(gcs, 'SaveFormat', 'StructureWithTime')
sim(gcs)
simplot(yout)
```

plots the output of the vdp demo model on a figure window as follows.

See Also   sim, set_param

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.

MaxDataPoints   nonnegative integer {0}

Limit number of output data points. This property limits the number of data points returned in t, x, and y to the last MaxDataPoints 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

$$e(i) <= \max(RelTol * abs(x(i)), AbsTol(i))$$

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 MaxDataPoints and Refine parameters, using default values for other parameters.

```
myopts = simset('MaxDataPoints', 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

Purpose   Get options structure properties and parameters.

Syntax   struct = simget(model)
                   value = simget(model, property)
                   value = simget(OptionStructure, 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

simget

**6**

Analyzing Simulation Results

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.
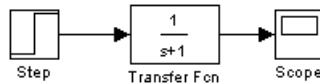

The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

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

These blocks are described in Chapter 9, "Block Reference."

Using Return Variables

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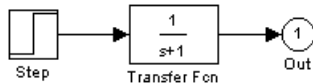

The block labeled Out is an Outport block from the Signals & Systems library. The output trajectory, yout, is returned by the integration solver. For more information, see "The Workspace I/O Pane" on page 5-18.

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.
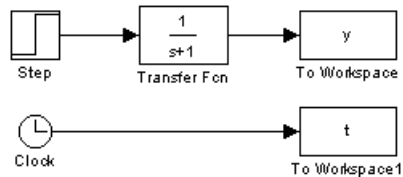
The variables y and t appear in the workspace when the simulation is complete. The time vector is stored by feeding a Clock block into a To Workspace block. The time vector can also be acquired by entering a variable name for the time on the **Workspace I/O** pane of the **Simulation Parameters** dialog box for menu-driven simulations, or by returning it using the sim command (see "The Workspace I/O Pane" on page 5-18 for more information).

The To Workspace block can accept an array input, with each input element's trajectory stored in the resulting workspace variable.
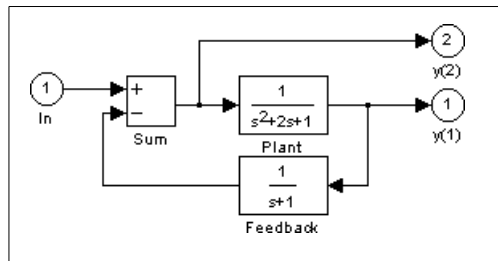
Linearization

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 6-9.

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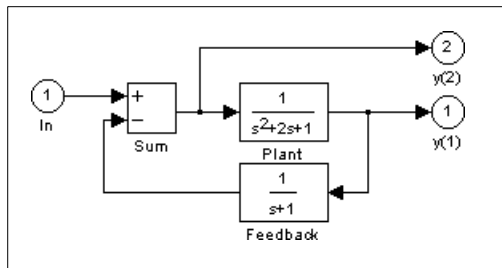
$$\frac{s}{s+a}$$

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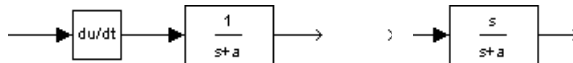
Equilibrium Point Determination

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 6-12.

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

Syntax   [A, B, C, D] = *linfun*('sys', x, u)
                 [num, den] = *linfun*('sys', x, u)
                 sys_struc = *linfun*('sys', x, u)

Arguments   *linfun*   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.

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.

$$x = Ax + Bu$$
$$y = Cx + Du$$

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

[A, B, C, D] = linmod('sys', x, u) obtains the linearized model of sys around
an operating point with the specified state variables x and the input u. If you
omit x and u, the default values are zero.

[num, den] = *linfun*('sys', x, u) returns the linearized model in transfer
function form.

sys_struc = *linfun*('sys', x, u) returns a structure that contains the
linearized model, including state names, input and output names, and
information about the operating point.

Discrete-Time System Linearization

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

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

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

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

• Ts is an integer multiple of all the sampling times in the system.
• The system is stable.

For systems that do not meet the first condition, in general the linearization is a time-varying system, which cannot be represented with the [*A*,*B*,*C*,*D*] state-space model that dlinmod returns.

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

```
all(abs(eig(Ad))) < 1
```

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

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

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

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

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

Notes   By default, the system time is set to zero. For systems that are dependent on time, you can set the variable 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..

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 ith row is the block name associated with the ith 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 6-4.

trim

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 $[x–x0, u, y]$. 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 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

```
abs([x–x0; u–u0; y–y0])
```

The command

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

6-12

finds the trim point closest to x0, u0, y0 that satisfies a specified set of state, input, and/or output conditions. The integer vectors ix, iu, and iy select the values in x0, u0, and y0 that must be satisfied. If trim cannot find an equilibrium point that satisfies the specified set of conditions exactly, it returns the nearest point that satisfies the conditions, namely

    abs([x(ix)-x0(ix); u(iu)-u0(iu); y(iy)-y0(iy)])

Use the syntax

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

to find specific nonequilibrium points, that is, points where the system's state derivatives have some specified, nonzero value. Here, dx0 specifies the state derivative values at the search's starting point and idx selects the values in dx0 that the search must satisfy exactly.

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 (Continued)
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

$$x = Ax + Bu$$

$$y = Cx + Du$$

The *A*, *B*, *C*, and *D* matrices are as follows in a system called sys.

```
A = [-0.09 -0.01;   1    0];
B = [ 0     -7;      0   -2];
C = [ 0      2;      1   -5];
D = [-3      0;      1    0];
```

Example 1   To find an equilibrium point, 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
```

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.

trim

7

Using Masks to Customize Blocks

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, $y = mx + b$.
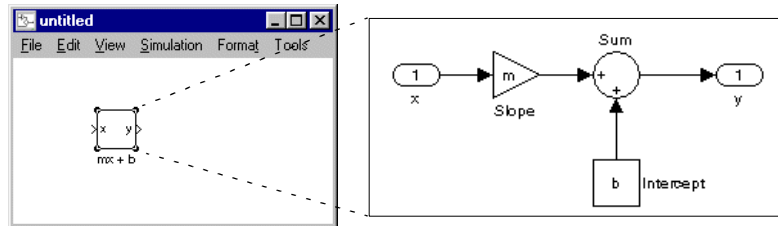
Ordinarily, when you double-click on a Subsystem block, the Subsystem block opens, displaying its blocks in a separate window. The $mx + b$ subsystem contains a Gain block, named Slope, whose Gain parameter is specified as m, and a Constant block, named Intercept, whose Constant value parameter is specified 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.
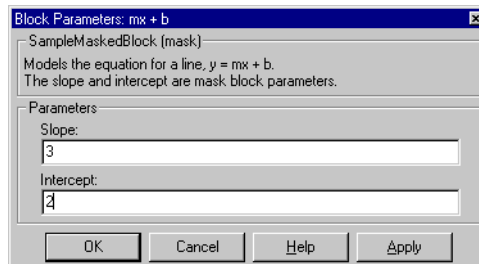
The mask dialog box

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** pane of the Mask Editor. For this sample model, the pane looks like this.
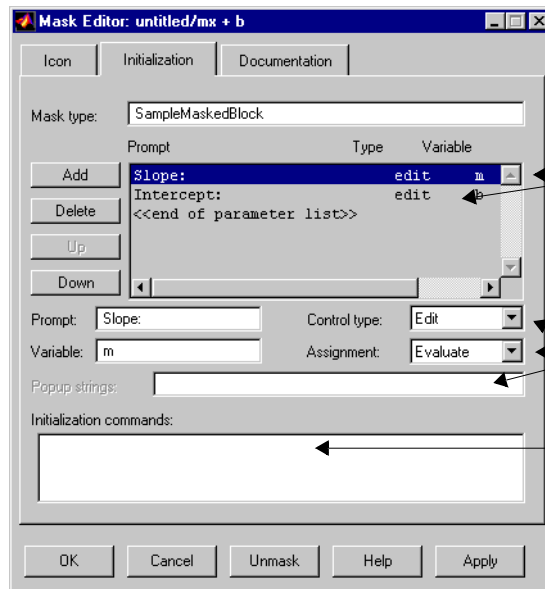
The Mask Editor enables you to specify these attributes of a mask parameter:

- 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 7-14). 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.
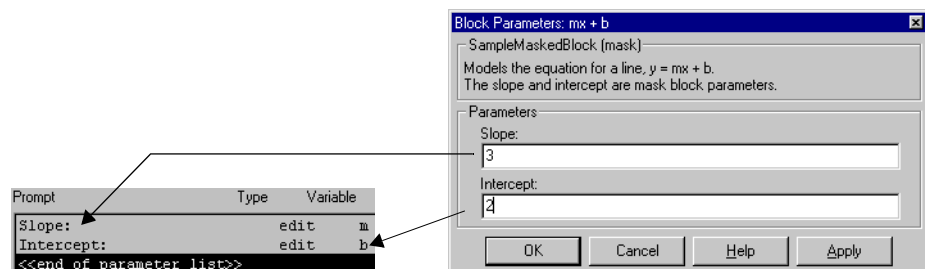
After you have created the mask parameters for slope and intercept, press the **OK** button. Then, double-click on the Subsystem block to open the newly constructed dialog box. Enter 3 for the **Slope** and 2 for the **Intercept** parameter.

7-5

Creating the Block Description and Help Text

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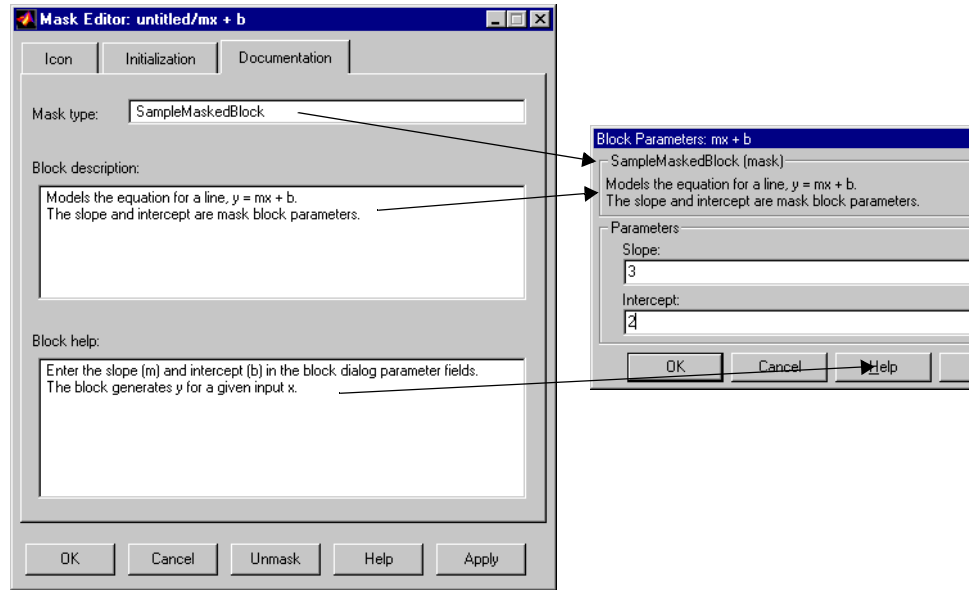
Creating the Block Icon

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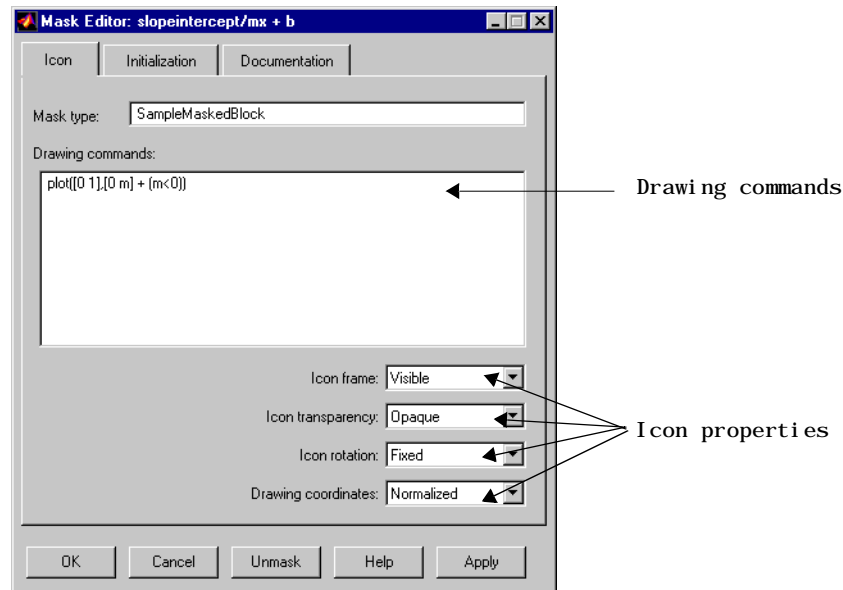
The drawing command plots a line from $(0, 0)$ to $(1, m)$. If the slope is negative, Simulink shifts the line up by 1 to keep it within the visible drawing area of the block.

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). See "Displaying Graphics on the Block Icon" on page 7-19 for more information.

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 panes, each handling a different aspect of the mask:

- The **Initialization** pane enables you to define and describe mask dialog box parameter prompts, name the variables associated with the parameters, and specify initialization commands.
- The **Icon** pane enables you to define the block icon.
- The **Documentation** pane 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 panes 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 panes. 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 Pane

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** pane. The **Initialization** pane for the mx+b sample masked system looks like this.
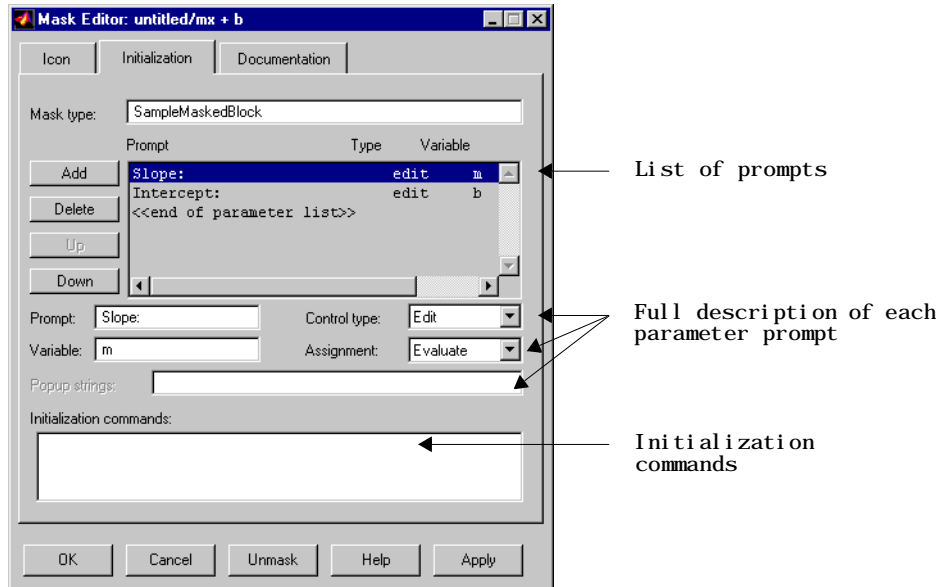
Prompts and Associated Variables

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 box which uses all three styles of controls (with the pop-up control open).
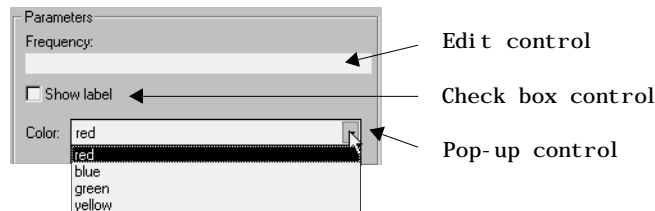
Defining an Edit Control

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.
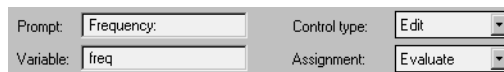
The value of the variable associated with the parameter (freq) is determined by the **Assignment** type defined for the prompt.

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, see "Libraries" on page 4–77.

Tunable Parameters

A tunable parameter is a 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.
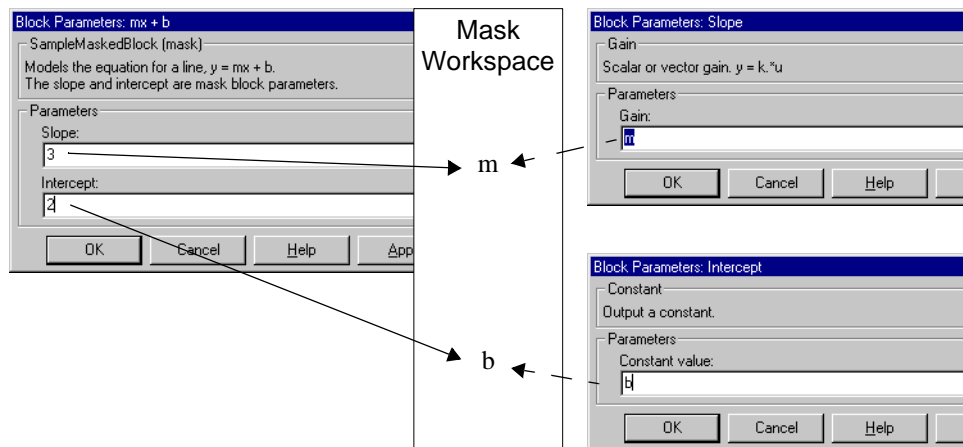
The Mask Workspace

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.

The contents of a mask workspace include the variables associated with the mask's parameters and variables defined by initialization commands.

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. 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).

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.

Masked subsystems create a hierarchy of workspaces. The workspace of a masked block is a subspace of the model workspace and of the workspaces of any blocks that contain the masked block. A masked block can access all variables that are uniquely defined in its workspace hierarchy. The blocks in a masked subsystem can similarly access any uniquely defined variable in the masked subsystem's workspace hierarchy.

If a variable is defined in more than one place in the hierarchy, the masked block can access only the most local definition. For example, suppose that model M contains masked subsystem A, which contains masked subsystem B. Further suppose that B refers to a variable x that exists in both A's and M's workspaces. In this case, the reference resolves to the value of x in A's workspace.

Note A masked block's initialization code can access only variables defined in the masked block's local workspace.

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 Pane

The **Icon** pane 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** pane.
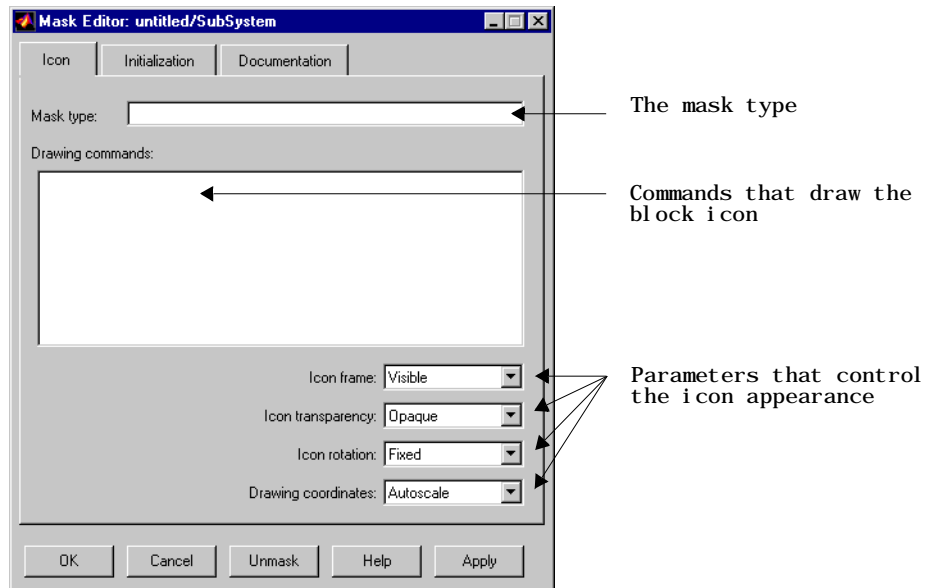
Drawing commands have access to all variables in the mask workspace.

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 7–22.

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.

Option	Aligns
left	The left end of the text at the specified point
right	The right end of the text at the specified point
center	The center of the text at the specified point

The text command offers the following vertical alignment options.

Option	Aligns
base	The baseline of the text at the specified point
bottom	The bottom line of the text at the specified point
middle	The midline of the text at the specified point

Option	Aligns
cap	The capitals line of the text at the specified point
top	The top of the text at the specified point

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.
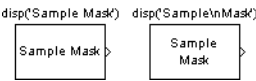
The `port_label` command lets you specify the labels of ports displayed on the icon. The command's syntax is

```
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])
```

Ramp

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 7–22.

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.
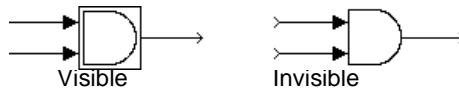
Icon transparency

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.
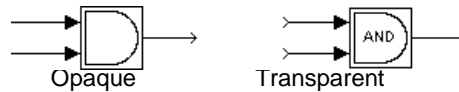
Icon rotation

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**.
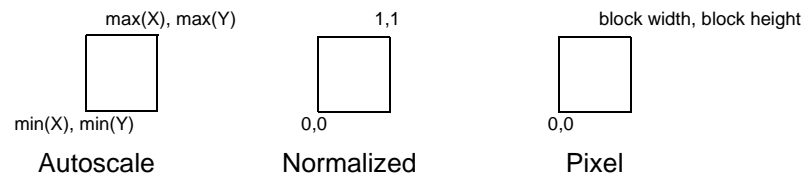
- **Autoscale** automatically scales the icon within the block frame. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors.

  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 mx + 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 Pane

The **Documentation** pane enables you to define or modify the type, description, and help text for a masked block. This figure shows how fields on the **Documentation** pane correspond to the mx+b sample mask block's dialog box.
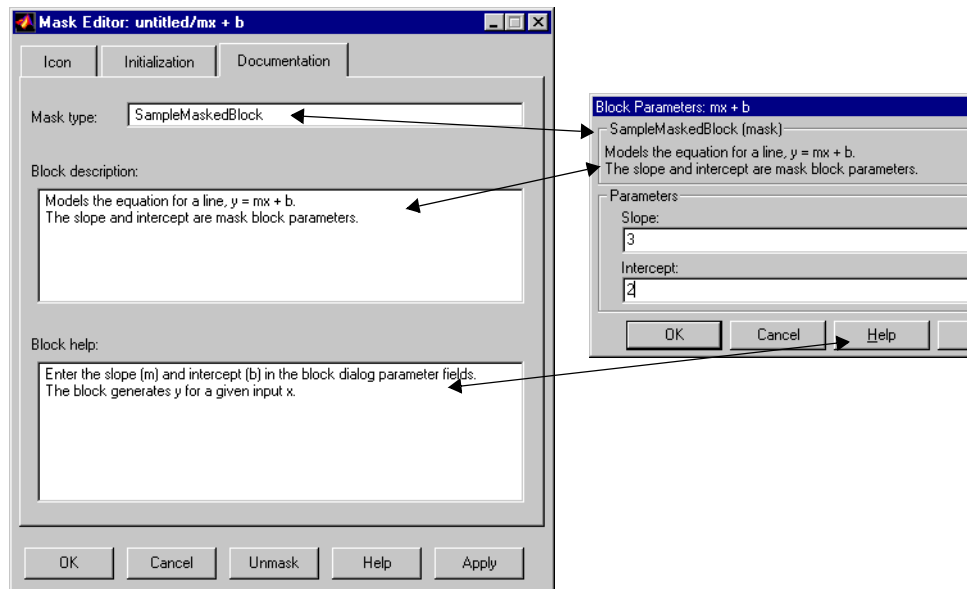
The Mask Type Field

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 panes 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(gcb,'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 Self-Modifying Masked Blocks

A masked block can modify itself based on user input. In particular, a masked block can change the contents of its underlying system block and set the parameters of those blocks based on user input. For example, you can create a block that adds or deletes input and output ports depending on some user setting.

When creating a self-modifying masked block, you must set its MaskSelfModifiable parameter to 'on'. Otherwise, Simulink generates an error when the block tries to modify itself, that is, when any code in the masked block's workspace tries to add or delete blocks from the underlying system block or modify the parameters of any blocks in the underlying system block.

To set the MaskSelfModifiable parameter, select the self-modifying block and enter the following command

```
set_param(gcb, 'MaskSelfModifiable', 'on');
```

at the MATLAB prompt. Then, save the block.

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(gcb, 'MaskCallbacks', {'parm1_callback', '',...
'parm3_callback'});
```

defines callbacks for the first and third parameters of the masked dialog for the currently selected block. To save the callback settings, save the model or library containing the masked block.

MaskDescription

The value of this parameter is a string specifying the description of this block. You can change a masked block's description dynamically by setting this parameter.

MaskEnables

The value of this parameter is a cell array of strings that define the enabled state of the user-defined parameter controls for this dialog. The first cell defines the enabled state of the control for the first parameter, the second for the second parameter, etc. A value of 'on' indicates that the corresponding control is enabled for user input; a value of 'off' indicates that the control is disabled.

You can enable or disable user input dynamically by setting this parameter in a callback. For example, the following command in a callback

```
set_param(gcb, 'MaskEnables', {'on', 'on', 'off'});
```

would disable the third control of the currently open masked block's dialog. 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(gcb, '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.

8

Conditionally Executed Subsystems

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 8-3.

- 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 8-8.

- 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 8-11 for more information.
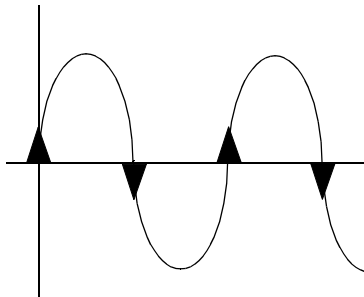
Enabled Subsystems

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.
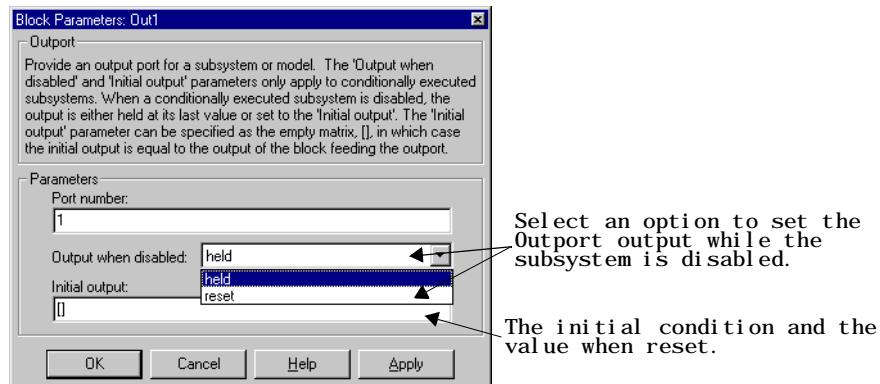

Subsystem

Setting Output Values While the Subsystem Is Disabled

Although an enabled subsystem does not execute while it is disabled, the output signal is still available to other blocks. While an enabled subsystem is disabled, you can choose to hold the subsystem outputs at their previous values or reset them to their initial conditions.

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

Select an option to set the Outport output while the subsystem is disabled.

The initial condition and the value when reset.

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.
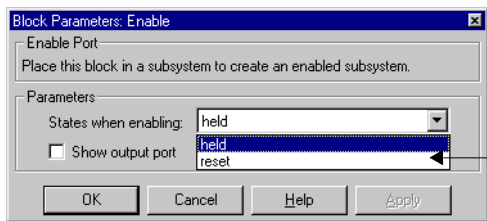
Select an option to set the states
when the subsystem is re-enabled.

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.
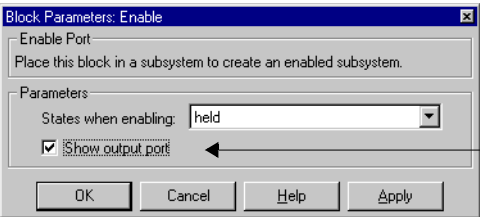
Select this check box to show the
output port.

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.

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


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

- Block A, which has a sample time of 0.25 second
- Block B, which has a sample time of 0.5 second
- Block C, within the Enabled subsystem, which has a sample time of 0.125 second
- Block D, also within the Enabled subsystem, which has a sample time of 0.25 second

The enable control signal is generated by a Pulse Generator block, labeled Signal E, which changes from 0 to 1 at 0.375 second and returns to 0 at 0.875 second.
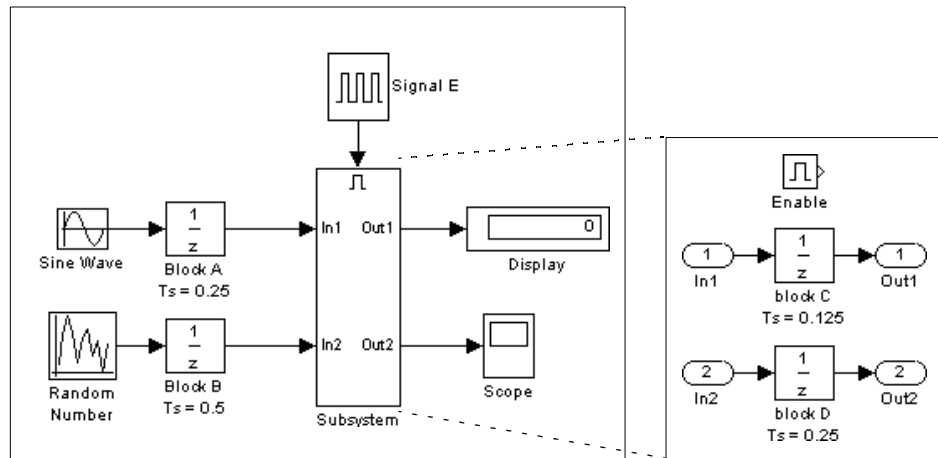
The chart below indicates when the discrete blocks execute.
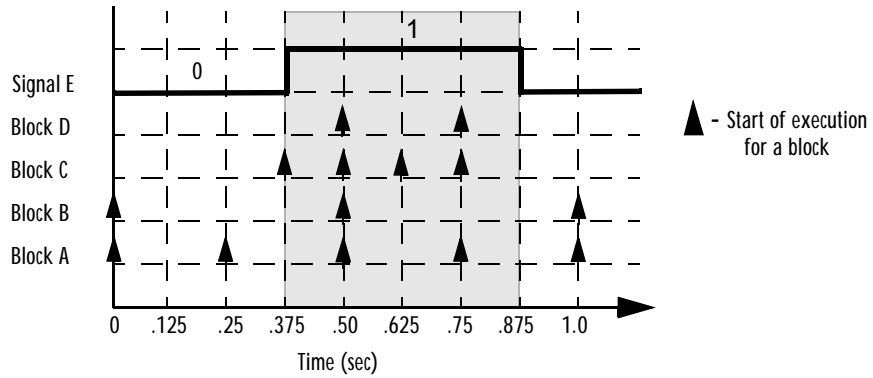
Blocks A and B execute independent of the enable signal because they are not part of the enabled subsystem. When the enable signal becomes positive, blocks C and D execute at their assigned sample rates until the enable signal becomes zero again. Note that block C does not execute at 0.875 second when the enable signal changes to zero.
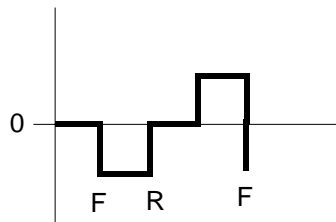
Triggered Subsystems

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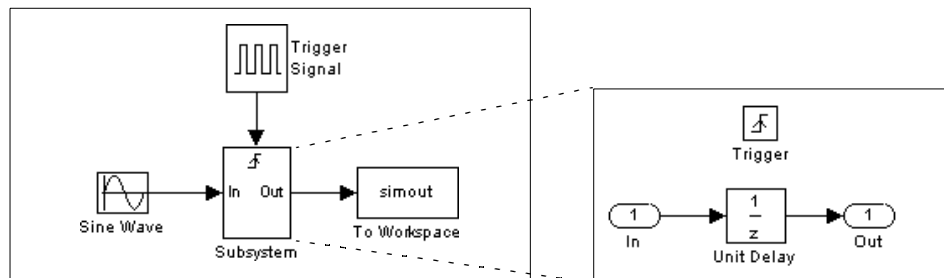
In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

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.

Subsystem

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.
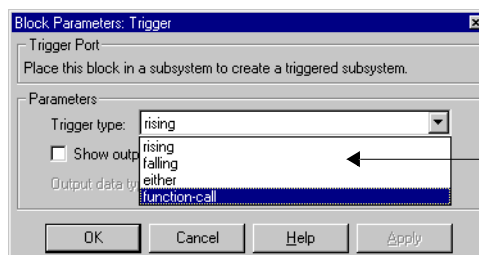
Simulink uses different symbols on the Trigger and Subsystem blocks to indicate rising and falling triggers (or either). This figure shows the trigger symbols on Subsystem blocks.

Subsystem with Rising trigger   Subsystem with Falling trigger   Subsystem with Rising or Falling trigger

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.
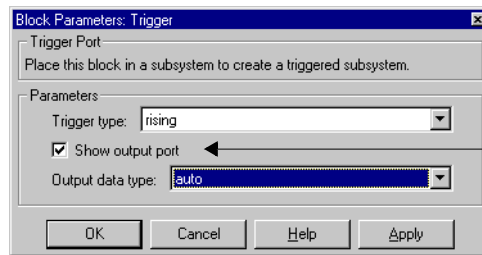
Select this check box to show the output port.

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*.
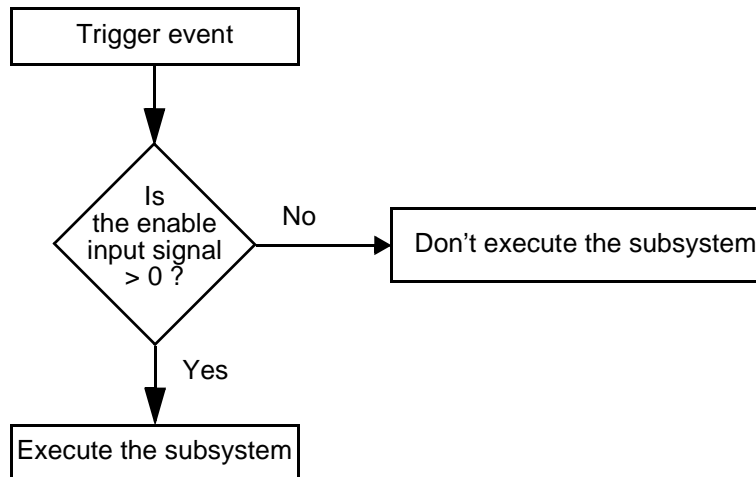
Blocks That a Triggered Subsystem Can Contain

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.
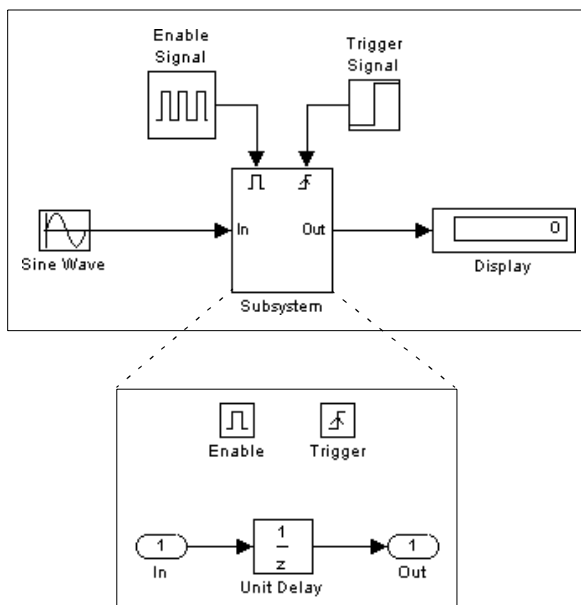

Subsystem

You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see "Setting Output Values While the Subsystem Is Disabled" on page 8–4. 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 8–4.

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 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.
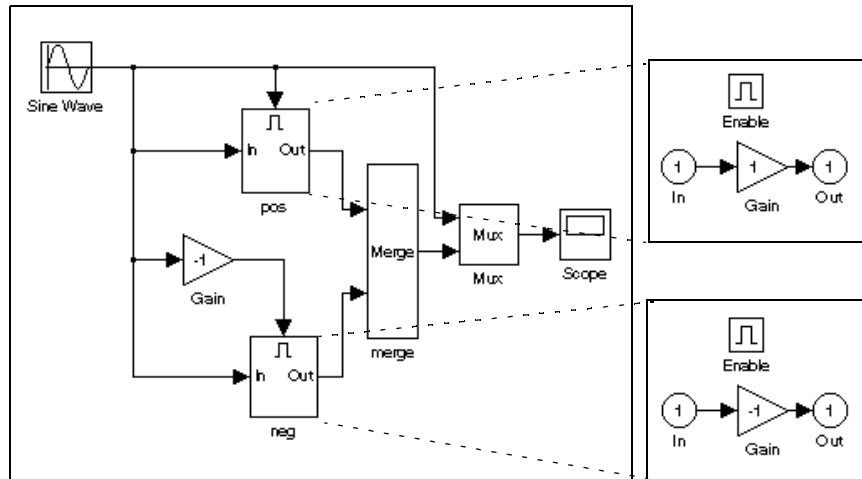
In this example, the block labeled "pos" is enabled when the AC waveform is positive; it passes the waveform unchanged to its output. The block labeled "neg" is enabled when the waveform is negative; it inverts the waveform. The Merge block passes the output of the currently enabled block to the Mux block, which passes the output, along with the original waveform, to the Scope block to create the following display.

9
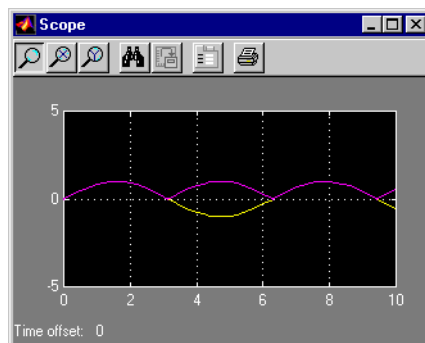
Block Reference

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 data types and numeric type (complex or real) accepted and generated by the block
- 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 3-18.
  - 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 3–23.
  - Scalar Expansion – whether or not scalar values are expanded to arrays. Some blocks expand scalar inputs and/or parameters as appropriate. For more information, see "Scalar Expansion of Inputs and Parameters" on page 4-34.
  - States – the number of discrete and continuous states.
  - Dimensionalized– whether the block accepts and/or generates multidimensional signal arrays. For more information, see "Working with Signals" on page 4–28.
  - Zero Crossings – whether the block detects zero-crossing events. For more information, see "Zero Crossing Detection" on page 3-14.

Simulink Block Libraries

Simulink organizes its blocks into block libraries according to their behavior.

- 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 *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 *Nonlinear* library contains blocks that describe nonlinear functions.
- 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.

Note You can use either the Simulink Library Browser (Windows only) or the MATLAB command **simulink3** to display and browse the block libraries.

The following tables list contents of all libraries except the *Blocksets and Toolboxes* and *Demos* libraries.

Table 9-1: 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.

Table 9-1: Sources Library Blocks (Continued)

Block Name	Purpose
Clock	Display and provide the simulation time.
Constant	Generate a constant value.
Digital Clock	Generate simulation time at the specified sampling interval.
Digital Pulse Generator	Generate pulses at regular intervals.
From File	Read data from a file.
From Workspace	Read data from a variable 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 9-2: Sinks Library Blocks

Block Name	Purpose
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.
To Workspace	Write data to a variable in the workspace.
XY Graph	Display an X-Y plot of signals using a MATLAB figure window.

Table 9-3: Discrete Library Blocks

Block Name	Purpose
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.

Table 9-3: Discrete Library Blocks (Continued)

Block Name	Purpose
Zero-Order Hold	Implement zero-order hold of one sample period.

Table 9-4: Continuous Library Blocks

Block Name	Purpose
Derivative	Output the time derivative of the input.
Integrator	Integrate a signal.
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 9-5: Math Library Blocks

Block Name	Purpose
Abs	Output the absolute value of the input.
Algebraic Constraint	Constrain the input signal to zero.
Bitwise Logical Operator	Logically mask, invert, or shift the bits of an unsigned integer signal.

Table 9-5: Math Library Blocks (Continued)

Block Name	Purpose
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.
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.

Table 9-5: Math Library Blocks (Continued)

Block Name	Purpose
Sum	Generate the sum of inputs.
Trigonometric Function	Perform a trigonometric function.

Table 9-6: Functions & Tables Library Blocks

Block Name	Purpose
Direct Look-Up Table (n-D)	
Fcn	Apply a specified expression to the input.
Look-Up Table	Perform piecewise linear mapping of the input.
Look-Up Table (2-D)	Perform piecewise linear mapping of two inputs.
Look-Up Table (n-D)	Perform piecewise linear or spline mapping of two or more inputs.
MATLAB Fcn	Apply a MATLAB function or expression to the input.
S-Function	Access an S-function.

Table 9-7: Nonlinear Library Blocks

Block Name	Purpose
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.

Table 9-7: Nonlinear Library Blocks (Continued)

Block Name	Purpose
Manual Switch	Switch between two inputs.
Multiport 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 9-8: Signals & Systems Library Blocks

Block Name	Purpose
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.

Table 9-8: Signals & Systems Library Blocks (Continued)

Block Name	Purpose
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.
Matrix Concatenation	Concatenate array inputs.
Merge	Combine several input lines into a scalar line.
Model Info	Display revision control information in a model.
Mux	Combine several input lines into a vector line.
Outport	Create an output port for a subsystem or an external output.
Reshape	Change the dimensionality of a signal.
Probe	Output an input signal's width, sample time, and/or signal type.
Selector	Select or reorder the elements of the input vector.
Signal Specification	Specify attributes of a signal.
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 any type and outputs a real value of the same data type as the input.

Dialog Box
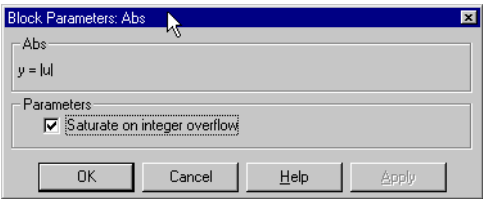
Saturate on integer overflow

When checked (default), the block maps signed integer input elements corresponding to the most negative value of that data type to the most positive value of that datatype.

- For 8-bit integers, -128 is mapped to 127.
- For 16-bit integers, -32768 maps to 32767.
- For 32-bit integers, -2147483648 maps to 2147483647.

When unchecked, the behavior of the block is undefined for signed integer input elements corresponding to the most negative value.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	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).
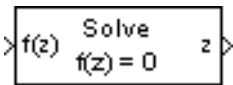
By default, the **Initial guess** parameter is zero. You can improve the efficiency of the algebraic loop solver by providing an **Initial guess** of the algebraic state z that is close to the solution value.

For example, the model below solves these equations.

```
z2 + z1 = 1
z2 – z1 = 1
```

The solution is $z2 = 1$, $z1 = 0$, as the Display blocks show.
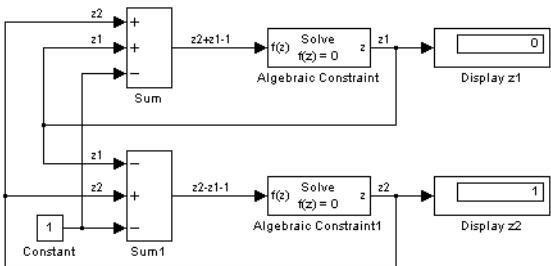
Data Type Support   An Algebraic Constraint block accepts and outputs real values of type double.

**Parameters
and Dialog Box**
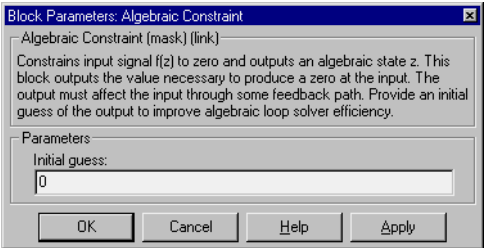
Initial guess

An initial guess of the solution value. The default is 0.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	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.
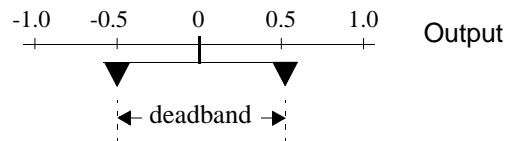
A system with play can be in one of three modes:

- 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).

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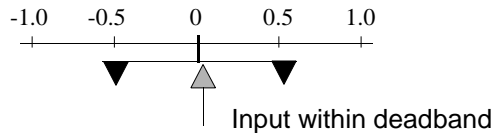
The next figure shows the state of the block when the input has reached the end of the deadband and engaged the output. The output remains at its previous value.

Input reaches end of deadband (engaged)
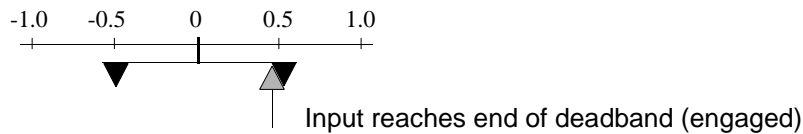
The final figure shows how a change in input affects the output while they are engaged.

Input moves in positive direction.
Output = Input - (deadband width/2)
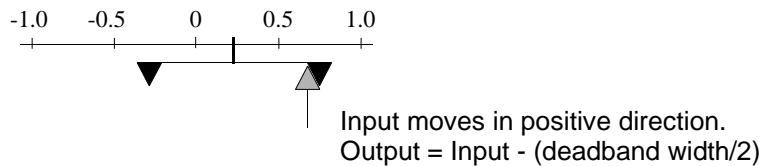
If the input reverses its direction, it disengages from the output. The output remains constant until the input either reaches the opposite end of the deadband or reverses its direction again and engages at the same end of the deadband. Now, as before, movement in the input causes equal movement in the 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.
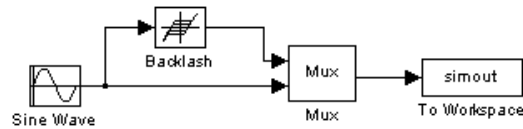
The Backlash block parameters are unchanged from their default values (the deadband width is 1 and the initial output is 0). Notice in the plotted output below that the Backlash block output is zero until the input reaches the end of the deadband (at 0.5). Now, the input and output are engaged and the 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.
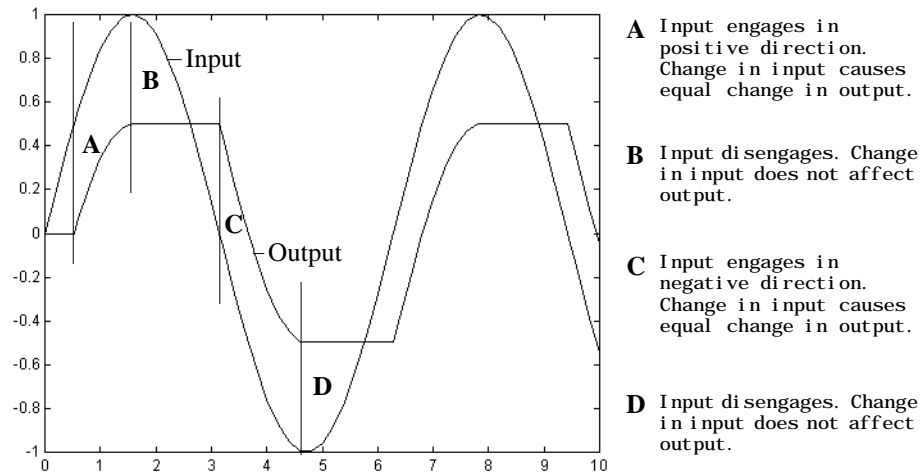
Data Type Support   A Backlash block accepts and outputs real values of type double.

Parameters and Dialog Box

Block Parameters: Backlash

Backlash

Model backlash where the deadband width specifies the amount of play in the system.

Parameters

Deadband width:

`1`

Initial output:

`0`

| OK | Cancel | Help | Apply |
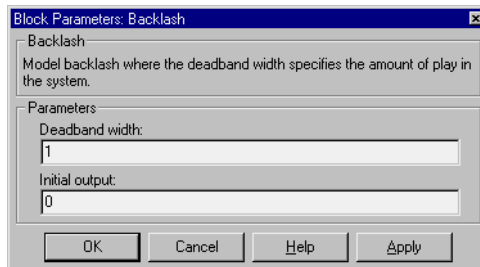
Deadband width

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
Dimensionalized	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

Block Parameters: Band-Limited White Noise

Continuous White Noise. (mask) (link)

White noise for continuous (s-domain) systems. Band-limited using zero-order-hold.

Parameters

Noise power:

[0.1]

Sample time:

0.1

Seed:

[23341]

| OK | Cancel | Help | Apply |
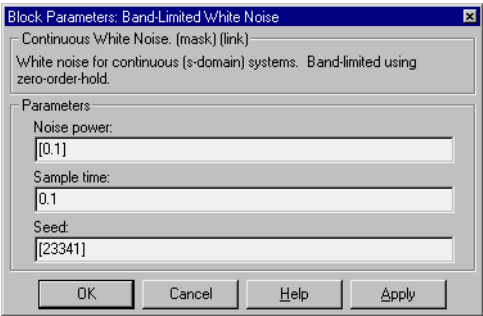
Noise power

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 **Noise power** and **Seed** parameters and output
Dimensionalized	Yes
Zero Crossing	No

Bitwise Logical Operator

Purpose   Logically mask, invert, or shift the bits of an unsigned integer signal.

Library   Math

Description   The Bitwise Logical Operator performs any of a set of logical masking (AND, OR, XOR) , inversion (NOT), and shifting (SHIFT_LEFT, SHIFT_RIGHT) operations on the bits of on an unsigned integer signal. The block's parameter dialog lets you choose the operation to perform. You can use the Bitwise Logical Operator block to perform bitwise operations on arrays of unsigned integer signals.

```
bitwise
AND
'FFFF'
```

Masking Operations

The Bitwise Logical Operator's masking operations (AND, OR, XOR) logically combine each bit of the input signal with the corresponding bit of a constant operand called the mask. You specify the mask's value and the logical operation via the block's parameter dialog. The mask and the logical operation determine the value of each bit of the output signal as follows.

Operation	Mask Bit	Input Bit	Output Bit
AND	1	1	1
	1	0	0
	0	1	0
	0	0	0
OR	1	1	1
	0	1	1
	1	0	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0

A Bitwise Operator block accepts arrays for both signals and masks. In general, the mask must have the same dimensionality as the input signal, i.e., a 5-by-4 input signal requires a 5-by-4 mask. The block applies each element of the mask to the corresponding input element. The following exceptions exist to the general rule that the input and the mask must have the same dimensionality:

- If the input is scalar and the mask is an array, the block outputs an array consisting of the result of applying each mask element to the input.
- If the input is an array and the mask is a scalar, the block outputs an array consisting of the result of applying the mask to each element of the input.
- If the input is a 1-D array (i.e., a vector), the mask may be a row or a column vector.

When selecting a masking operation, use the **Second operand** field of the block's parameter dialog to specify the mask or masks. You can enter any MATLAB expression that evaluates to a scalar, matrix, or cell array. Use strings in your mask expression to specify hexadecimal values (e.g., 'FFFF').

If necessary, the block truncates the high order bits of the mask value to fit the word size of the input signal's data type. For example, suppose you specify the mask value as 'FF00' and the input signal is of type uint8. The block truncates the specified value to '00'.

You can use matrices to specify hexadecimal masks, but beware of the pitfalls of such an approach. For example, the MATLAB expression ['00' 'FF'] represents a single string 'FF00' rather than two strings. Similarly, the expression ['FFFF'; '0000'] represents two strings but the expression ['FFFF'; '00'] is invalid and hence causes MATLAB to signal an error. You can avoid these pitfalls by always using cell arrays to specify hexadecimal values, or to mix decimal and hexadecimal values, for masks. For example, the following model
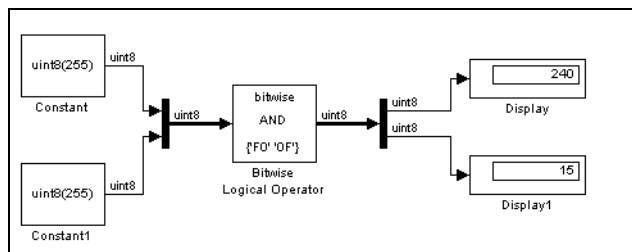
uses a cell array ({'F0' '0F'}) to specify hexadecimal values for the masks for a two-element input vector.

Inversion Operation

The Bitwise Logical Operator's NOT operation inverts the bits of the input signal. In particular, it performs a one's complement operation on the input signal to produce an output signal each of whose bits is 1 if the corresponding input bit is 0 and vice-versa.

Shift Operations

The Bitwise Logical Operator's shift operations, SHIFT_LEFT and SHIFT_RIGHT, shift the bits of the input signal left or right to produce the output signal. You specify the amount of the shift in the **Second operand** field of the block's parameter dialog. If you specify a shift amount that is greater than the word size of the input signal, the block uses the input word size as the shift amount, resulting in a zero output signal. The dimensionality rules that apply to masks and inputs also apply to shift factors and inputs.

Data Type Support

The Bitwise Logical Operator accepts real-valued inputs of any of the unsigned integer data types: uint8, uint16, uint32. All the elements of a vector input must be of the same data type. The output signal is of the same data type as the input.

Parameters and Dialog Box

Block Parameters: Bitwise Logical Operator

BitwiseOperator (mask) (link)

Perform a bitwise operation on uint8, uint16 or uint32 input port data with values from the Second operand parameter. Hex values can be entered as strings, e.g., 'FF00'.

Parameters

Bitwise operator: AND

Second operand:
'FFFF'

OK   Cancel   Help   Apply
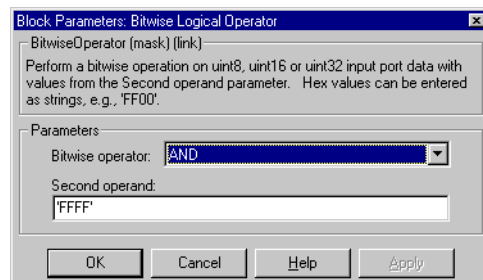
Bitwise operator

Specifies the bitwise operator applied to the input signal.

Second operand

Specifies the mask operand for masking operations and the shift amount for shift operations. You can enter any MATLAB expression that evaluates to a scalar, matrix, or cell array. If the block input is an array, the block applies each parameter value to the corresponding element of the input. If the input is a scalar, the block outputs an array, each of whose elements is the result of applying the corresponding parameter value to the input.

Characteristics	Sample Time	Inherited from driving block
	Scalar Expansion	Of inputs and **Second operand** parameter
	Dimensionalized	Yes
	States	None
	Zero Crossing	No
	Direct Feedthrough	Yes

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** check box. 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
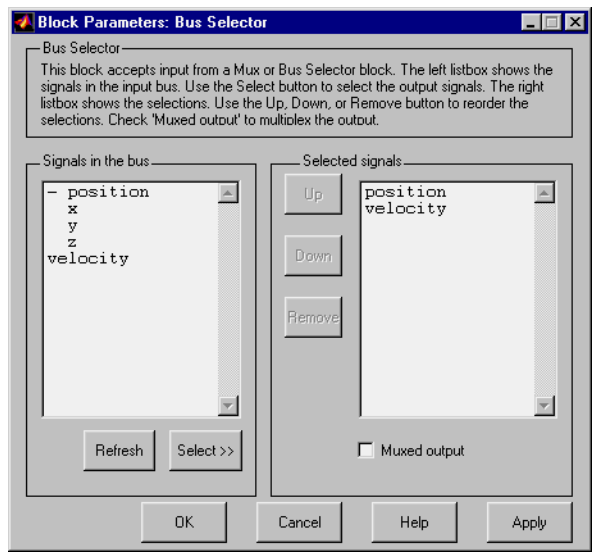
Signals in the bus

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 ouput port is automatically set by the block except when you check the **Muxed output** check box. 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
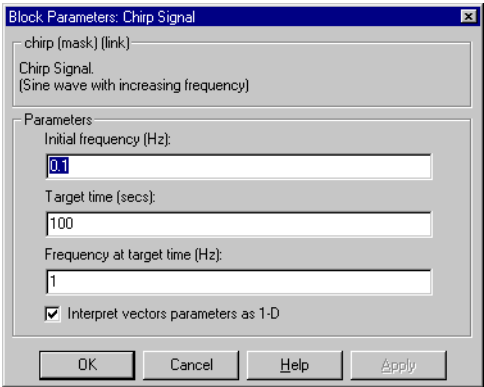
Description

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.

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

Data Type Support

A Chirp Signal block outputs a real-valued signal of type doubl e.

Parameters and Dialog Box

Initial frequency

The initial frequency of the signal, specified as a scalar or matrix value. The default is 0. 1 Hz.

Target time

> The time at which the frequency reaches the **Frequency at target time** parameter value, a scalar or matrix value. The frequency continues to change at the same rate after this time. The default is 100 seconds.

Frequency at target time

> The frequency of the signal at the target time, a scalar or matrix value. The default is 1 Hz.

Interpret vector parameters as 1-D

> If selected, column or row matrix values for the **Initial frequency**, **Target time**, and **Frequency at target time** parameters result in a vector output whose elements are the elements of the row or column.

Characteristics	Sample Time	Continuous
	Scalar Expansion	Of parameters
	Dimensionalized	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 doubl e.

Parameters and Dialog Box
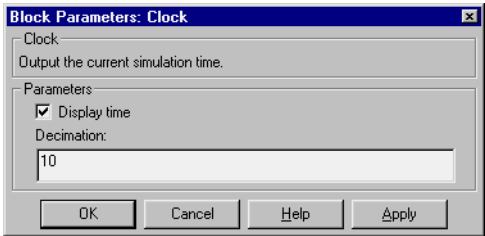
Display time

   Use the **Display time** check box to display the current simulation time inside the Clock block icon.

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. Note that if this parameter is not zero, the simulation must use a fixed-step solver to ensure accurate clock updates.

Characteristics

Sample Time	Continuous
Scalar Expansion	N/A

Dimensionalized	No
Zero Crossing	No

Combinatorial Logic

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

                         number of rows = 2 ^ (number of inputs)

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

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

                     ### 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.
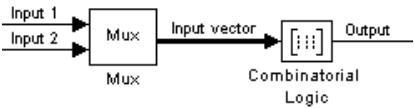
                     The table below indicates the combination of inputs that generate each output.
                     The input signal labeled "Input 1" corresponds to the column in the table
                     labeled Input 1. Similarly, the input signal "Input 2" corresponds to the column

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

Combinatorial Logic

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

The type of signals accepted by a Combinatorial Logic block depends on whether you have selected Simulink's Boolean logic signals option (see "Enabling Strict Boolean Type Checking" on page 4-48). If this option is enabled, the block accepts real signals of type boolean or double. The truth table may have Boolean values (0 or 1) of any data type. If the table contains nonBoolean values, the table's data type must be double. The type of the ouput is the same as that of the input except that the block outputs double if the input is boolean and the truth table contains nonboolean values. If Boolean compatibility mode is disabled, the Combinatorial Logic block accepts only signals of type boolean. The block outputs double if the truth table contains nonBoolean values of type double. Otherwise, the output is boolean.

Parameters and Dialog Box

Block Parameters: Combinatorial Logic

Combinatorial Logic

Look up the elements of the input vector (treated as boolean values) in the truth table and outputs the corresponding row of the 'Truth table' parameter. The input side of the truth table is implicit.

Parameters

Truth table:

[0 0;0 1;0 1;1 0;0 1;1 0;1 0;1 1]

OK   Cancel   Help   Apply
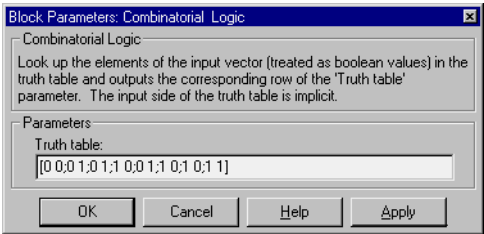
Truth table

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
Dimensionalized	Yes; the output width is the number of columns of the **Truth table** parameter
Zero Crossing	No

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 doubl e. 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 doubl e. The input may be an array of complex signals, in which case the output signals are also arrays. The magnitude signal array contains the magnitudes of the corresponding complex input elements. The angle output similarly contains the angles of the input elements.
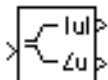
Data Type Support   See the description above.

Parameters and Dialog Box
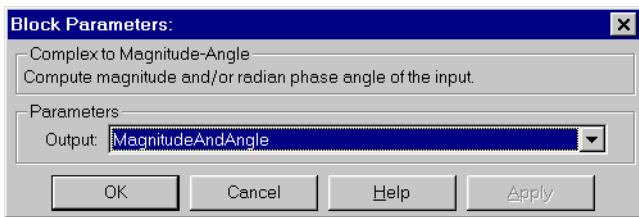
Output

Determines the output of this block. Choose from the following values: Magni tudeAndAngl e (outputs the input signal's magnitude and phase angle in radians), Magni t ude (outputs the input's magnitude), Angl e (outputs the input's phase angle in radians).

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Zero Crossing	No

Complex to Real-Imag

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 any type.
                  It outputs the real and/or imaginary part of the input signal, depending on the
                  setting of the **Output** parameter. The real outputs are of the same data type as
                  the complex input. The input may be an array (vector or matrix) of complex
                  signals, in which case the output signals are arrays of the same dimensions.
                  The real array contains the real parts of the corresponding complex input
                  elements. The imaginary output similarly contains the imaginary parts of the
                  input elements.

Data Type   See the description above.
Support

Parameters
and Dialog Box

Block Parameters: ✕

Complex to Real-imag
Output the real and/or imaginary components of the input.

Parameters
Output: RealAndImag ▼

| OK | Cancel | Help | Apply |

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).

Characteristics   Direct Feedthrough   Yes

                      Sample Time   Inherited from driving block

                      Scalar Expansion   No

                      Dimensionalized   Yes

                      Zero Crossing   No

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

Configurable Subsystem

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.
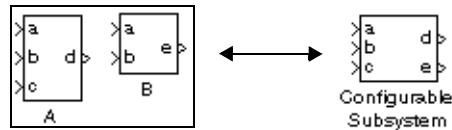
In this example, port a on the Configurable Subsystem block connects to port a of the selected library block no matter which block is selected. On the other hand, port c on the Configurable Subsystem block functions only if library block A is selected. Otherwise, it simply terminates.

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.

To create a configurable subsytem:

1 Create a library of blocks representing the various configurations of the configurable subsystem.

2 Create an instance of the Configurable Subsystem block in the library. To do this, drag a copy of the Configurable Subsystem block from the Simulink Signals and Systems library into the library you created in the preceding step.

3 Display the Configurable Subsystem block's dialog by double-clicking it. The dialog displays a list of the other blocks in the library.

4 Check the blocks that represent the various configurations of the configurable subsystems you are creating.

5 Close the dialog.

Save the library.

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.

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.
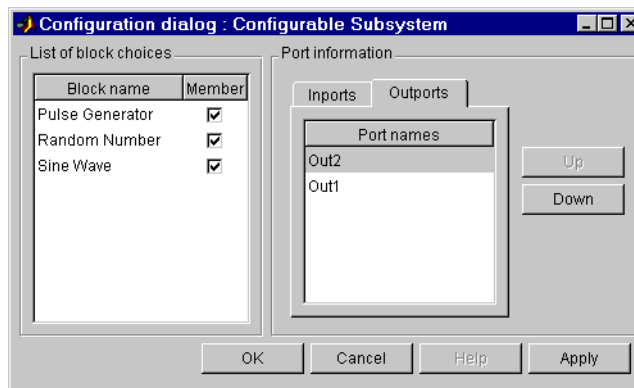
List of block choices

> Check the blocks you want to include as members of the configurable subsystem. You can include user-defined subsystems as blocks.

Port information

> Lists of input and output ports of member blocks. In the case of multiports, you can rearrange selected port positions by pressing the **Up** and **Down** buttons.

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

Configurable Subsystem

The following figure shows the dialog box for a Configurable Subsystem block.
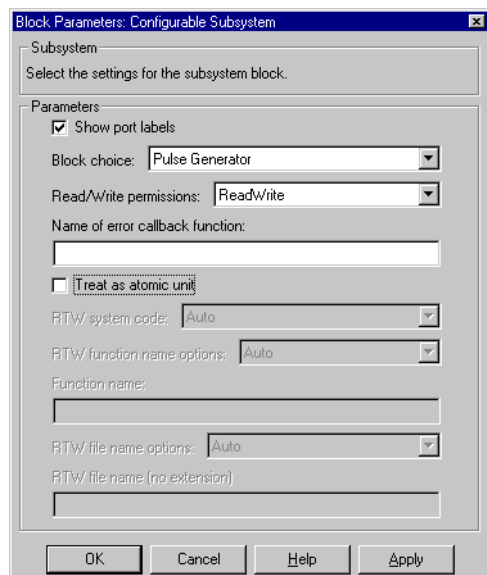
Block choice

> The block that this Configurable Subsystem block current represents. This menu lists all the blocks in your configurable subsystem library.

> The parameters below **Block choice** are related to subsystem behavior. See the Subsystem block reference page for more information.

Characteristics   A Configurable Subsystem block has the characteristics of the block that it currently represents. Double-clicking the block opens the dialog box for the block that it currently represents.

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, a vector, or a matrix, depending on the dimensionality of the **Constant value** parameter and the setting of the **Interpret vector parameters as 1-D** parameter. If the **Interpret vector parameters as 1-D** parameter is selected and the **Constant value** parameter is a column or row matrix, the output is a 1-D array (i.e., a vector) whose elements are the elements of the parameter. Otherwise, the output is a 2-D array (i.e., a matrix) that has the same dimensions as the parameter and whose elements are the parameter elements.

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

Block Parameters: Constant

―Constant―
Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

―Parameters―
Constant value:

1

☑ Interpret vector parameters as 1-D

[OK]   [Cancel]   [Help]   [Apply]
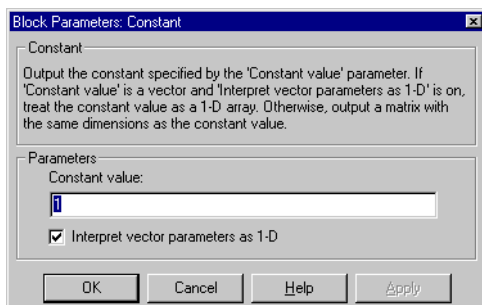
Constant value

The output of the block.

Interpret vector parameters as 1-D

If selected, a column or row matrix value for the **Constant value** parameter results in a vector output whose elements are the elements of the row or column.

Constant

Characteristics

Sample Time	Constant
Scalar Expansion	No
Dimensionalized	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
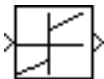
$$y = \text{sign}(u) * (\text{Gain} * \text{abs}(u) + \text{Offset})$$

where y is the output, u is the input, and Gain and 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

Block Parameters: Coulomb & Viscous Friction

Coulombic and Viscous Friction (mask) (link)

A discontinuity offset at zero models coulomb friction. Linear gain models viscous friction.
y = sign(x) * (Gain * abs(x) + Offset)

Parameters

Coulomb friction value (Offset):

[1 3 2 0]

Coefficient of viscous friction (Gain):

1

| OK | Cancel | Help | Apply |
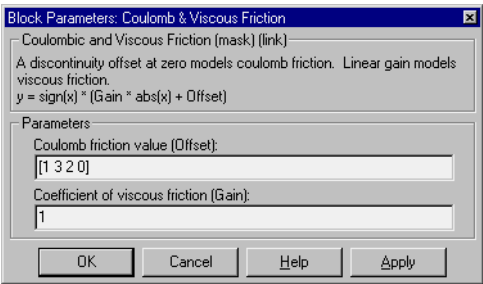
Coulomb friction value

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.

Characteristics | Direct Feedthrough | Yes |
|---|---|
| Sample Time | Inherited from driving block |
| Scalar Expansion | No |

Coulomb and Viscous Friction

Dimensionalized	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 dimensionality 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
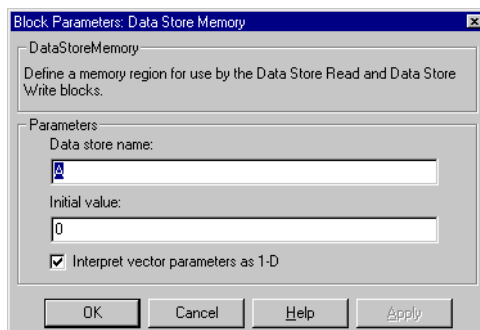
Data Store Memory

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.

Interpret vector parameters as 1-D

If selected, a column or row matrix value for the **Initial value** parameters initializes the data store to a 1-D array (vector) whose elements are equal to the elements of the row or column vector.

Characteristics	Sample Time	N/A
	Dimensionalized	Yes

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 9-43.

                       More than one Data Store Read block can read from the same data store.

                       Note To avoid block output consistency errors, be careful not to set an
                       execution priority on a Data Store Read block such that the block reads from
                       the data store before the store is updated by any Data Store Write blocks that
                       write to the store in the same time step.

**Data Type   A Data Store Read block outputs a real signal of type double.
Support**

**Parameters
and Dialog Box**
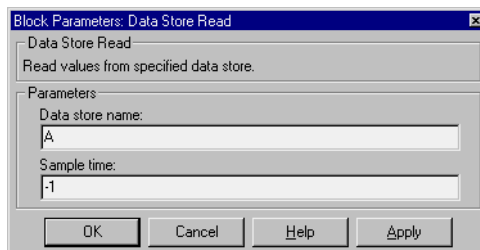
Data store name

    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.

Data Store Read

Characteristics

Sample Time	Continuous or discrete
Dimensionalized	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 9-43. 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
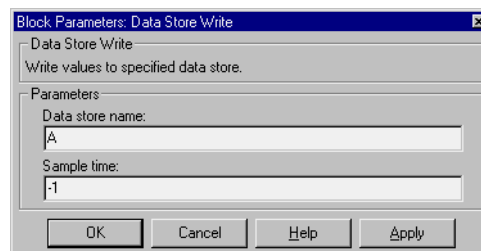
Data store name

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

Characteristics   Sample Time   Continuous or discrete

Dimensionalized   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
specfied 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   See block description above.
Support**

**Parameters
and Dialog Box**

Block Parameters: Data Type Conversion

Data Type Conversion
Convert input signal to specified data type.

Parameters
Data type: double
☑ Saturate on integer overflow

OK   Cancel   Help   Apply
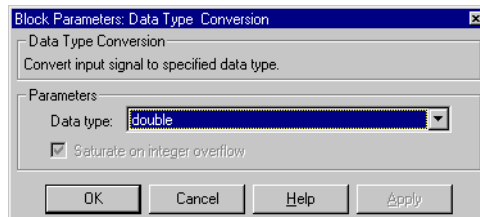
Data type

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 Pane" on page 5–26).

Data Type Conversion

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Zero Crossing	No

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:
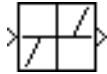
- If the input is within the dead zone (greater than the lower limit and less than the upper limit), the output is zero.
- If the input is greater than or equal to the upper limit, the output is the input minus the upper limit.
- If the input is less than or equal to the lower limit, the output is the 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.
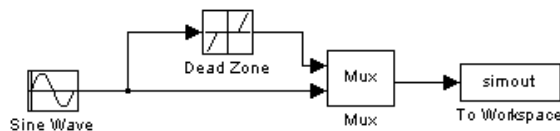
This plot shows the effect of the Dead Zone block on the sine wave. While the input (the sine wave) is between -0.5 and 0.5, the output is zero.
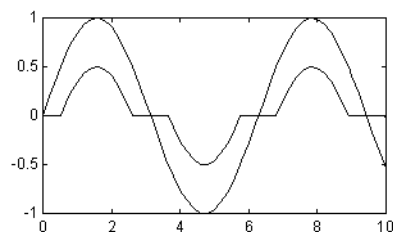
Data Type Support   A Dead Zone block accepts and outputs a real signal of any data type.

Dead Zone

Parameters and Dialog Box
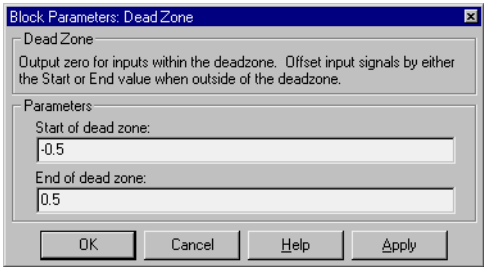
Start of dead zone

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
Dimensionalized	Yes
Zero Crossing	Yes, to detect when the limits are reached

Purpose   Extract and output the elements of a bus or vector signal.

Library   Signals & Systems

Description   The Demux block extracts the components of an input signal and outputs the components as separate signals. The block accepts either vector (1-D array) signals or bus signals (see "Signal Buses" on page 4-30 for more information). The **Number of outputs** parameter allows you to specify the number and, optionally, the dimensionality of each output port. If you do not specify the dimensionality of the outputs, the block determines the dimensionality of the outputs for you.

The Demux block operates in either vector or bus selection mode, depending on whether you have selected the **Bus selection mode** parameter. The two modes differ in the types of signals they accept. Vector mode accepts only a vector-like signal, that is, either a scalar (one-element array), vector (1-D array), or a column or row vector (one row or one column 2-D array). Bus selection mode accepts only the output of a Mux block or another Demux block.

The Demux block's **Number of outputs** parameter determines the number and dimensionality of the block's outputs, depending on the mode in which the block operates.

Specifying the Number of Outputs in Vector Mode

In vector mode, the value of the parameter can be a scalar specifying the number of outputs or a vector whose elements specify the widths of the block's output ports. The block determines the size of the block's outputs from the size of the input signal and the value of the **Number of outputs** parameter.

Demux

The following table summarizes how the block determines the outputs for an input vector of width n.

Parameter Value	Block outputs...	Comments
$p = n$	p scalar signals.	For example, if the input is a three-element vector and you specify three outputs, the block outputs three scalar signals.
$p > n$	Error	
$p < n$ $n \bmod p = 0$	p vector signals each having n/p elements	If the input is a six-element vector and you specify three outputs, the block outputs three two-element vectors.
$p < n$ $n \bmod p = m$	m vector signals each having $(n/p) + 1$ elements and $p - m$ signals have n/p elements.	If the input is a five-element vector and you specify three outputs, the block outputs two two-element vector signals and one scalar signal.
$[p_1 \ p_2 \ \cdots \ p_m]$ $p_1 + p_2 + \cdots + p_m = n$ $p_i > 0$	m vector signals having widths p_1, p_2, $\cdots \ p_m$	If the input is a five-element vector and you specify [3, 2] as the output, the block outputs three of the input elements on one port and the other two elements on the other ports.

Parameter Value	Block outputs...	Comments
$[p_1 \; p_2 \; \cdots \; p_m]$ $p_1 + p_2 + \ldots + p_m = n$ some or all $p_i = -1$	m vector signals	If p_i is greater than zero, the corresponding output has width p_i. If p_i is -1, the width of the corresponding output is dynamically sized.
$[p_1 \; p_2 \; \cdots \; p_m]$ $p_1 + p_2 + \ldots + p_m \mathrel{!}= n$ $p_i = > 0$	Error	

Note that you can specify the number of outputs as less than the the number of input elements, in which case the block distributes the elements as evenly as possible over the outputs as illustrated in the following example.
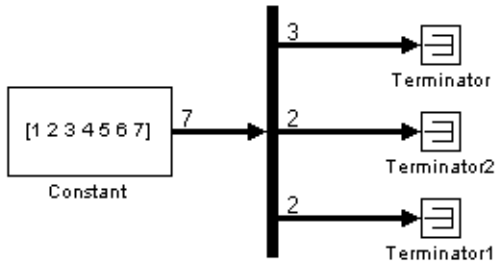
You can use -1 in a vector expression to indicate that the block should dynamically size the corresponding port. For example, the expression $[-1, \; 3 \; -1]$ causes the block to output three signals in which the second signal always has three elements while the size of the first and second signals depends on the size of the input signal.

If a vector expression comprises positive values and -1 values, the block assigns as many elements as needed to the ports with positive values and distributes the remain elements as evenly as possible over the ports with -1 values. For example, suppose that the block input is seven elements wide and you specify

the output as [-1, 3 -1]. In this case, the block outputs two elements on the first port, three elements on the second, and two elements on the third.
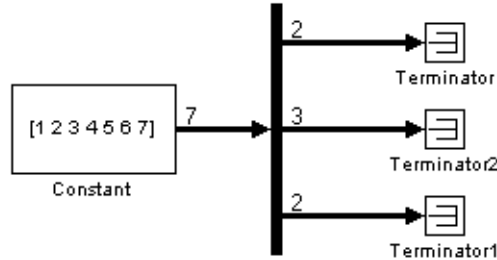
Specifying the Number of Outputs in Bus Selection Mode

In bus selection mode, the value of the **Number of outputs** parameter can be a:

- Scalar specifying the number of output ports

  The specified value must equal the number of input signals. For example, if the input bus comprises two signals and the value of this parameter is a scalar, the value must equal 2.
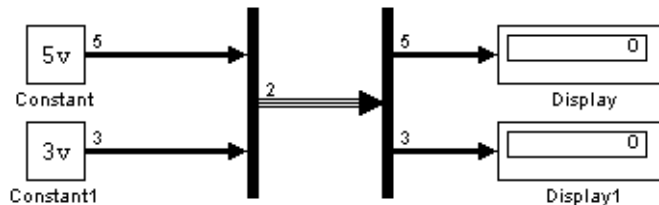
- Vector each of whose elements specifies the number of signals to output on the corresponding port

  For example, if the input bus contains five signals, you can specify the output as [3, 2], in which case the block outputs three of the input signals on one port and the other two signals on a second port.

- Cell array each of whose elements is a cell array of vectors specifying the dimensions of the signals output by the corresponding port

The cell array format constrains the Demux block to accept only signals of specified dimensions. For example, the cell array {{[2 2], 3} {1}} tells the block to accept only a bus signal comprising a 2-by-2 matrix, a three-element vector, and a scalar signal. You can use the value -1 in a cell array expression to let the block determine the dimensionality of a particular output, based on the input. For example, the following diagram uses the cell array expression {{-1}, {-1,-1}} to specify the output of the leftmost Demux block.
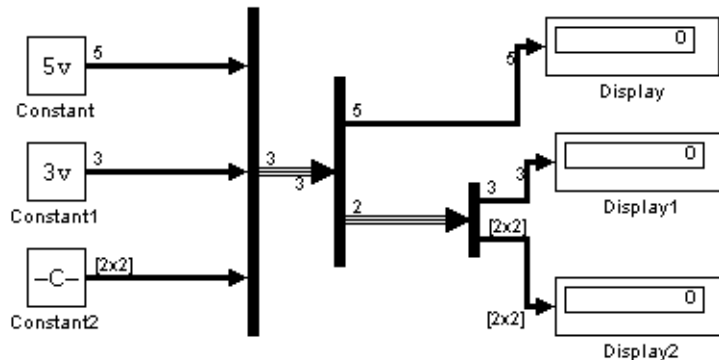
In bus selection mode, if you specify the dimensionality of an output port, i.e., specify any other value than -1, the corresponding input element must match the specified dimensionality.

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

Data Type Support   A Demux block accepts and outputs signals of any numeric (complex or real) and data type.

Demux

**Parameters
and Dialog Box**
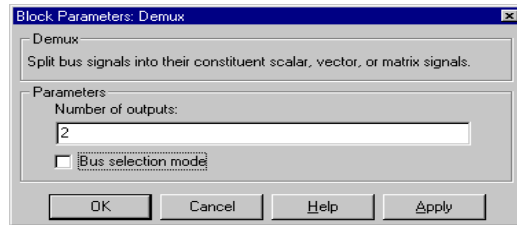
Number of outputs

The number and dimensions of outputs.

Bus selection mode

Enable bus selection mode.

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 Δu is the change in input value and Δt 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 6–4.

Data Type Support   A Derivative block accepts and outputs a real signal of type double.

Derivative

Dialog Box

```
Block Parameters: Derivative                    ☒
┌─ Derivative ──────────────────────────────────┐
│ Numerical derivative:  du/dt.                  │
└────────────────────────────────────────────────┘
   ┌───OK───┐  ┌─Cancel─┐  ┌──Help──┐  ┌─Apply──┐
   └────────┘  └────────┘  └────────┘  └────────┘
```
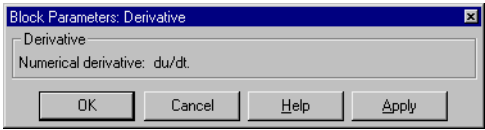
Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	N/A
States	0
Dimensionalized	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.

12:34

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 doubl e.

Parameters and Dialog Box

Block Parameters: Digital Clock

Digital Clock
Output current simulation time at the specified rate.

Parameters
Sample time:
1

OK   Cancel   Help   Apply
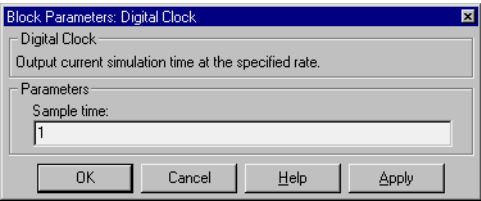
Sample time

The sampling interval. The default value is 1 second.

Characteristics

Sample Time	Discrete
Scalar Expansion	No
Dimensionalized	No
Zero Crossing	No

Direct Look-Up Table (n-D)

Purpose   Index into an N-dimensional table to retrieve a scalar, vector or 2-D matrix.

Library   Functions & Tables

Description   The Direct Look-Up Table (n-D) block uses its block inputs as zero-based indices into an n-D table. The number of inputs varies with the shape of the output desired. The output can be a scalar, a vector, or a 2-D matrix. The look-up table uses zero-based indexing, so integer datatypes can fully address their range. For example, a table dimension using the ui nt 8 data type can address all 256 elements.

You define a set of output values as the **Table data** parameter. You specify what the output shape is: a scalar, a vector or a 2-D matrix. The first input specifies the zero-based index to the first dimension higher than the number of dimensions in the output, the second input specifies the index to the next table dimension, and so on, as shown by this figure:
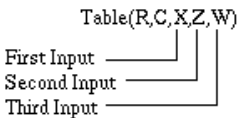
The figure shows a 5-D table with an output shape set to "2-D Matrix"; the output is a 2-D Matrix with R rows and C columns.

This figure shows the set of all the different icons that the Direct Look-Up Table block shows (depending on which options you choose in the block's dialog box).
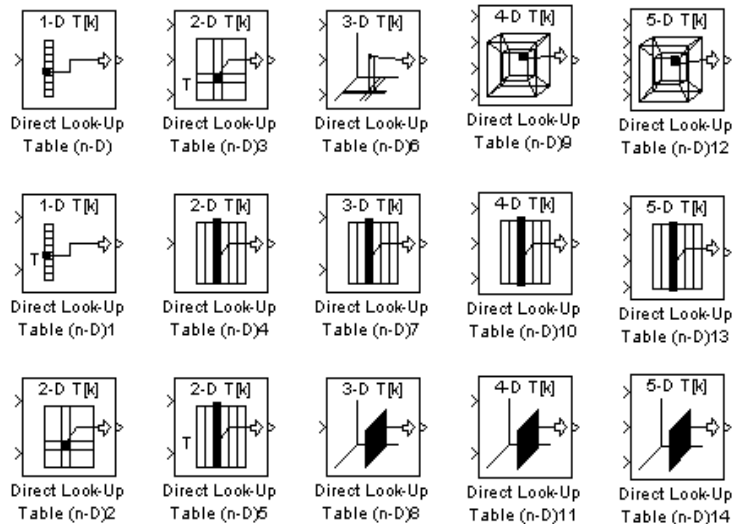
With dimensions higher than 4, the icon matches the 4-D icons, but will show the exact number of dimensions in the top text, e.g., "8-D T[k]." The top row of icons is used when the block output is made from one or more single-element lookups on the table. The blocks labelled "n-D Direct Table Lookup5," 6, 8 and 12 are configured to extract a column from the table and the two blocks ending in 7 and 9 are extracting a plane from the table. Blocks in the figure ending in 10, 11 and 12 are configured to have the table be an input instead of a parameter.

Example

In this example, the block parameters are defined as

```
Invalid input value:   "Clip and Warn"
Output shape:          "Vector"
Table data:            int16(a)
```

where a is a 4-D array of linearly increasing numbers calculated using MATLAB.

```
a = ones(20, 4, 5, 7);  L = prod(size(a));
a(1:L) = [1:L]';
```

Remembering that the table indices are zero-based, the figure shows the block outputting a vector of the 20 values in the second column of the fourth element of the third dimension from the third element of the fourth dimension.
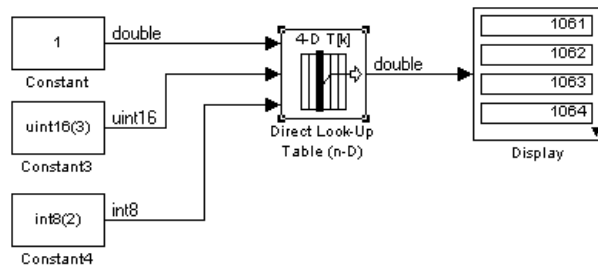
The output values in this example can be calculated manually in MATLAB (which uses 1-based indexing):

```
a(:, 1+1, 1+3, 1+2)
```

```
ans =
```

```
            1061
            1062
            1063
            1064
            1065
            1066
            1067
            1068
            1069
            1070
            1071
            1072
            1073
```

1074
1075
1076
1077
1078
1079
1080

Data Type Support

The Direct Look-Up Table (n-D) block accepts mixed-type signals of type doubl e, si ngl e, i nt 8, ui nt 8, i nt 16, ui nt 16, i nt 32 and, ui nt 32. The output type can differ from the input type and can be any of the types listed for input; the output type is inherited from the data type of the **Table data** parameter.

In the case that the table comes into the block on an input port, the output port type is inherited from the table input port. Inputs for indexing must be real; table data can be complex.

Dialog Box
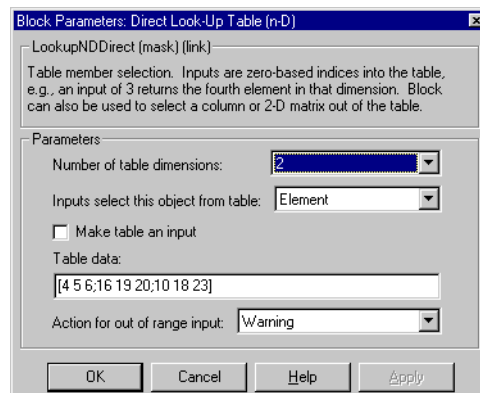
Number of table dimensions

The number of dimensions that the **Table data** parameter must have. This determines the number of independent variables for the table and hence the number of inputs to the block The number of dimensions that the **Table data** parameter must have. This determines the number of independent variables for the table and hence the number of inputs to the block (see descriptions for "Explicit Number of dimensions" and "Use one (vector) input port instead of N ports," below).

Inputs select this object from table

Specify whether the output data is a single element, an n-d column, or a 2-D matrix. The number of ports changes for each selection:

Element — # of ports = # of dimensions

Column — # of ports = # of dimensions - 1

2-D Matrix — # of ports = # of dimension -2

This numbering agrees with MATLAB's indexing. For example, if you have a 4-D table of data, to access a single element you must specify four indices, as in array(1, 2, 3, 4). To specify a column, you need three indices, as in array(:, 2, 3, 4). Finally, to specify a 2-D matrix, you only need two indices, as in array(:, :, 3, 4).

Make table an input

Checking this box forces the Direct Look-Up Table (n-D) to ignore the Table Data parameter. Instead, a new port appears with "T" next to it. Use this port to input table data.

Table data

The table of output values. The matrix size must match the dimensions defined by the N **breakpoint set** parameter or by the **Explicit number of dimensions** parameter when the number of dimensions exceeds four. During block diagram editing, you can leave the **Table data** field empty, but for running the simulation, you must match the number of dimensions in the **Table data** to the **Number of table dimensions**. For information about how to construct multidimensional arrays in MATLAB, see Multidimensional Arrays in MATLAB's online documentation.

Action for out of range input

None, Warning, Error.

Real-Time Workshop Note: in the generated code, the "Clip and Warn" and "Clip Index" options cause the Real-Time Workshop to generate clipping code with no code included to generate warnings. Code generated for the other option, "Generate Error", has *no* clipping code or error messages at all, working on the assumption that simulation during the design phase of your project will reveal model defects leading to

out-of-range cases. This assumption helps the code generated by the
Real-Time Workshop to be highly efficient.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving blocks
Scalar Expansion	For scalar lookups only (not when returning a column or a 2-D Matrix from the table)
Dimensionalized	For scalar lookups only (not when returning a column or a 2-D Matrix from the table)
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 9-82 for more information about coefficients.
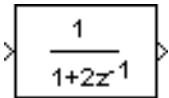
The Discrete Filter block represents the method often used by signal processing engineers, who describe digital filters using polynomials in z^{-1} (the delay operator). The Discrete Transfer Fcn block represents the method often used by control engineers, who represent a discrete system as polynomials 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 9-255.

Data Type Support   A Discrete Filter block accepts and outputs a real signal of type doubl e.

Parameters and Dialog Box
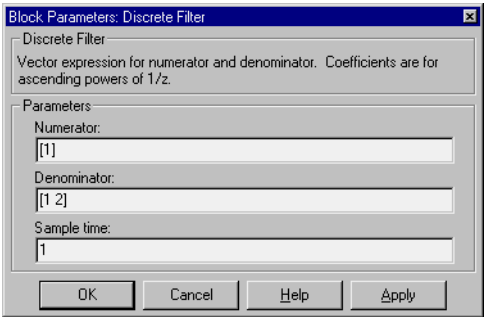
Numerator

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.

Characteristics	Direct Feedthrough	Only if the lengths of the **Numerator** and **Denominator** parameters are equal
	Sample Time	Discrete
	Scalar Expansion	No
	States	Length of **Denominator** parameter -1
	Dimensionalized	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.

You can specify the following pulse parameters. 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. All the parameters must have the same dimensions after scalar expansion of any scalar parameters.

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 9-183).

Data Type Support   A Discrete Pulse Generator block accepts and outputs a real signal of type double.

Parameters and Dialog Box
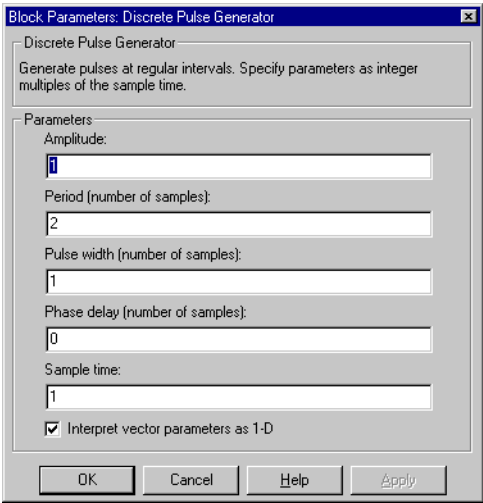
Amplitude

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.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the pulse generation parameters result in a vector output signal.

Characteristics		
	Sample Time	Discrete
	Scalar Expansion	Of parameters
	Dimensionalized	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

$$x(n+1) = Ax(n) + Bu(n)$$
$$y(n) = Cx(n) + Du(n)$$
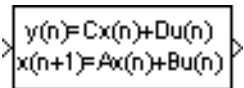
```
y(n)=Cx(n)+Du(n)
x(n+1)=Ax(n)+Bu(n)
```

where u is the input, x is the state, and y is the output. 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 doubl e.

Parameters and Dialog Box

Block Parameters: Discrete State-Space

Discrete State Space

Discrete state-space model:
x(n+1) = Ax(n) + Bu(n)
y(n)   = Cx(n) + Du(n)

Parameters

A:
1

B:
1

C:
1

D:
1

Initial conditions:
0

Sample time:
1

OK   Cancel   Help   Apply

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
Dimensionalized	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.
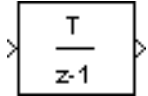
The Discrete-Time Integrator block allows you to:

- 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, $y(0) = x(0) = IC$ (clipped if necessary), i.e., the initial output of the block is always the initial condition.

- 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 $x(k+1) = x(k) + T*u(k)$, then we have:

  ```
  Step 0:     y(0)   = x(0) = IC (clip if necessary)
              x(1)   = y(0) + T*u(0)

  Step 1:     y(1)   = x(1)
              x(2)   = x(1) + T*u(1)

  Step k:     y(k)   = x(k)
              x(k+1) = x(k) + T*u(k) (clip if necessary)
  ```

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 $T*z/(z–1)$. This gives us

    `y(k) = y(k–1) + T * u(k).`

    Let $x(k) = y(k–1)$, then we have:

    ```
    Step 0:    y(0)    = x(0) = IC (clipped if necessary)
               x(1)    = y(0)

    Step 1:    y(1)    = x(1) + T*u(1)
               x(2)    = y(1)

    Step k:    y(k)    = x(k) + T*u(k)
               x(k+1) = y(k)
    ```

    With this method, input port 1 has direct feedthrough.

- Trapezoidal method. For this method, $1/s$ is approximated by $T/2*(z+1)/(z–1)$.

    When T is fixed (equal to the sampling period), let

    `x(k) = y(k–1) + T/2 * u(k–1).`

    Then we have:

    ```
    Step 0:    y(0)    = x(0) = IC (clipped if necessary)
               x(1)    = y(0) + T/2 * u(0)

    Step 1:    y(1)    = x(1) + T/2 * u(1)
               x(2)    = y(1) + T/2 * u(1)

    Step k:    y(k)    = x(k) + T/2 * u(k)
               x(k+1) = y(k) + T/2 * u(k)
    ```

Here, $x(k+1)$ is the best estimate of the next output. It isn't quite the state, in the sense that $x(k) != y(k)$.

If T is variable (i.e. obtained from the triggering times), then we have:

```
Step 0:     y(0)    = x(0) = IC (clipped if necessary)
            x(1)    = y(0)

Step 1:     x(1)    = x(1) + T/2 * (u(1) + u(0))
            x(2)    = y(1)

Step k:     y(k)    = x(k) + T/2 * (u(k) + u(k-1))
            x(k+1) = y(k)
```

With this method, input port 1 has direct feedthrough.

The block icon reflects the selected integration method, as this figure shows.

DT Int
Forward Euler

DT Int
Backward Euler

DT Int
Trapezoidal

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.

Input

Initial condition

$\frac{T}{z-1}$

Output

Discrete-Time
Integrator

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 reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than or equal to 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 or equal to 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.
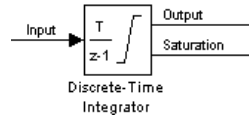
The signal has one of three values:

- 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.

- 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.
- Select level to trigger the reset and hold the output to the initial condition while the reset signal is nonzero.

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

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.
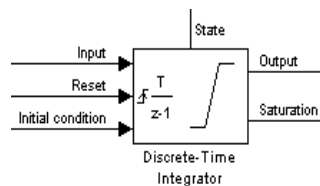
Data Type Support   A Discrete-Time Integrator block accepts and outputs real signals of type doubl e.

Discrete-Time Integrator

Parameters and Dialog Box

Block Parameters: Discrete-Time Integrator

┌ Discrete-Time Integrator ──────────────────────
Discrete-time integration of the input signal.

┌ Parameters ──────────────────────────────────

Integrator method ForwardEuler ▼

External reset none ▼

Initial condition source internal ▼

Initial condition:
0

☑ Limit output

Upper saturation limit:
inf

Lower saturation limit:
-inf

☐ Show saturation port
☐ Show state port

Sample time:
1

[OK] [Cancel] [Help] [Apply]

Integrator method

The integration method. The default is ForwardEuler.

External reset

Resets the states to their initial conditions when a trigger event (rising, falling, either, level) 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
	Dimensionalized	Yes
	Zero Crossing	One for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation

Discrete Transfer Fcn

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
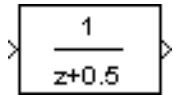
$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^n + num_1 z^{n-1} + \dots + num_m z^{n-m}}{den_0 z^n + den_1 z^{n-1} + \dots + den_n}$$

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 9- 255 for more information.

Data Type Support   A Discrete Transfer Function block accepts and outputs real signals of type double.
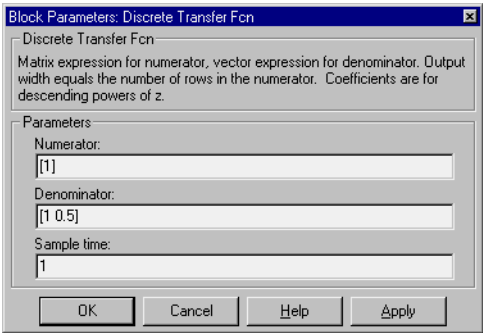
Parameters and Dialog Box

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.

Characteristics		
Direct Feedthrough	Only if the lengths of the **Numerator** and **Denominator** parameters are equal	
Sample Time	Discrete	
Scalar Expansion	No	
States	Length of **Denominator** parameter -1	
Dimensionalized	No	
Zero Crossing	No	

Discrete Zero-Pole

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
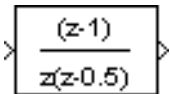
$$H(z) = K\frac{Z(z)}{P(z)} = K\frac{(z-Z_1)(z-Z_2)\ldots(z-Z_m)}{(z-P_1)(z-P_2)\ldots(z-P_n)}$$

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 9-276 for more information.

Data Type Support   A Discrete Zero-Pole block accepts and outputs real signals of type doubl e.

Parameters and Dialog Box

Block Parameters: Discrete Zero-Pole

Discrete Zero-Pole

Matrix expression for zeros. Vector expression for poles and gain. Output width equals the number of columns in zeros matrix, or one if zeros is a vector.

Parameters

Zeros:

[1]

Poles:

[0 0.5]

Gain:

1

Sample time:

1

OK   Cancel   Help   Apply
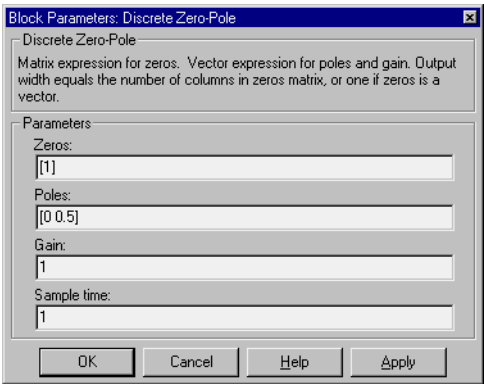
Zeros

The matrix of zeros. The default is [1].

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.

Characteristics		
	Direct Feedthrough	Yes, if the number of zeros and poles are equal
	Sample Time	Discrete
	Scalar Expansion	No
	States	Length of **Poles** vector
	Dimensionalized	No
	Zero Crossing	No

Display

Purpose	Show the value of the input.
Library	Sinks
Description	The Display block shows the value of its input.
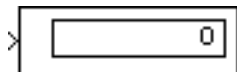
You can control the display format by selecting a **Format** choice:

- 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 signal storage reuse feature. See "Signal storage reuse" on page 5-31 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 nth 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 an array, 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 array elements. For example, the figure below shows a model that passes a vector (1-D array) to a Display block. The

top model shows the block before it is resized; notice the black triangle. The bottom model shows the resized block displaying both input elements.

Displays only one element of input vector but indicates there are more

Displays both elements of input vector
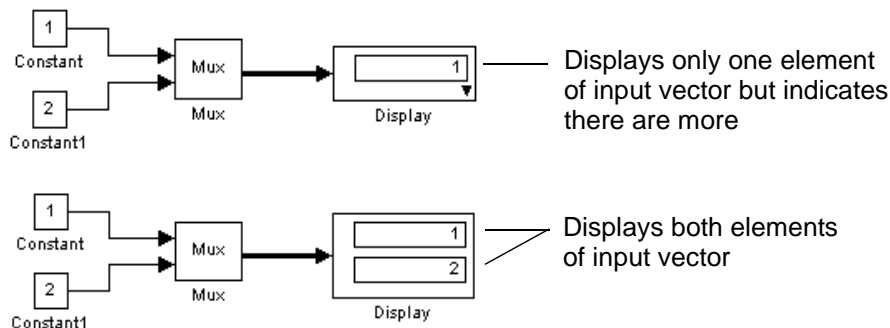
Data Type Support

A Display block accepts and outputs real or complex signals of any data type.

Parameters and Dialog Box
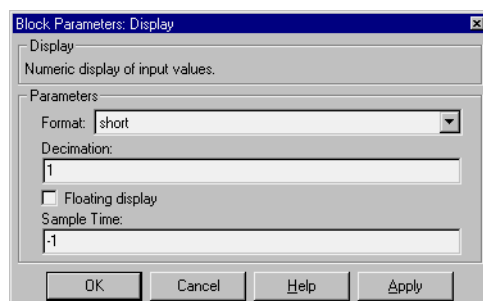
Format

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.

Display

Characteristics   Sample Time   Inherited from driving block

   Dimensionalized   Yes

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
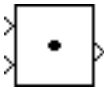
    y = sum(conj(u1) .* u2)

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.
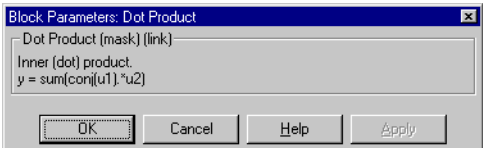
Dialog Box

Block Parameters: Dot Product
Dot Product (mask) (link)
Inner (dot) product.
y = sum(conj(u1).*u2)

OK   Cancel   Help   Apply

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No

Dot Product

States	0
Dimensionalized	Yes
Zero Crossing	No

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.

                   A subsystem can contain no more than one Enable block.

**Data Type   The data type of the input of the Enable port may be any data type. See
Support**   Chapter 8, "Conditionally Executed Subsystems" for more information about
                   enabled subsystems.

**Parameters
and Dialog Box**
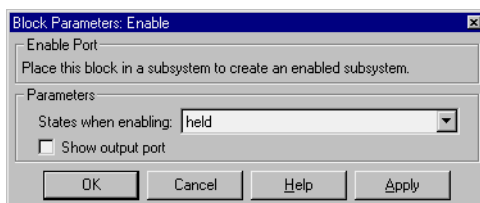
States when enabling

    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.

Enable

Characteristics   Sample Time   Determined by the signal at the enable port

                      Dimensionalized   Yes

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:
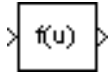
- u — the input to the block. If u is a vector, u(i) represents the ith element
  of the vector; u(1) or u alone represents the first element.
- Numeric constants
- Arithmetic operators (+ – * /)
- Relational operators (== != > < >= <=) — The expression returns 1 if the
  relation is TRUE; otherwise, 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 ||

Fcn

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
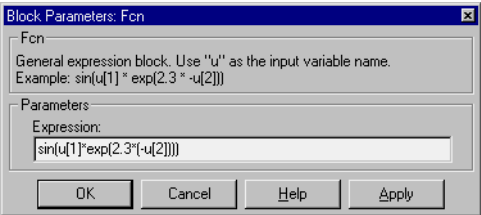
Expression

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
Dimensionalized	No
Zero Crossing	No

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 f ohdemo. This figure compares the output from a Sine Wave block and a First-Order Hold block.
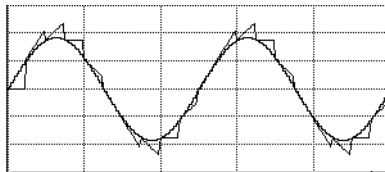
Data Type Support   A First-Order Hold block accepts and outputs signals of type doubl e.
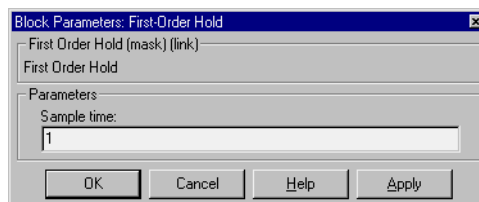
Parameters and Dialog Box

Sample time

The time interval between samples.

Characteristics
Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	No
States	1 continuous and 1 discrete per input element

First-Order Hold

| Dimensionalized | Yes |
| Zero Crossing | No |

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.
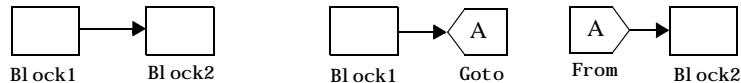
                   Associated Goto and From blocks can appear anywhere in a model with this
                   exception: if either block is in a conditionally executed subsystem, the other
                   block must be either in the same subsystem or in a subsystem below it in the
                   model hierarchy (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 9-111, and Goto Tag
                   Visibility on page 9-114. 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.

From

Data Type Support   A From block outputs signals of any real or complex data type.

Parameters and Dialog Box
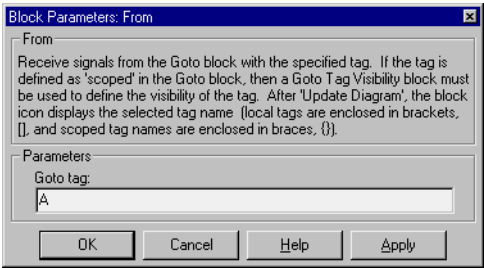
Goto tag

The tag of the Goto block passing the signal to this From block.

Characteristics   Sample Time   Inherited from block driving the Goto block

Dimensionalized   Yes

Purpose   Read data from a file.

Library   Sources

Description   The From File block outputs data read from a file. The block icon displays the pathname 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.

$$
\begin{bmatrix}
t_1 & t_2 & \dots t_{final} \\
u1_1 & u1_2 & \dots u1_{final} \\
\dots & & \\
un_1 & un_2 & \dots un_{final}
\end{bmatrix}
$$

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.

From File

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, "Creating a Model" 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
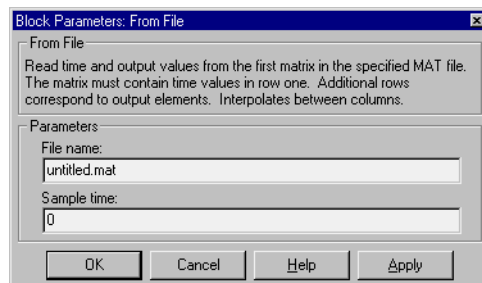
File name

The fully qualified path name or file name of the file that contains the data used as input. The default file name is untitled.mat. If you specify a file name, Simulink assumes the file resides in MATLAB's working directory. (To determine the working directory, type pwd at the MATLAB command line.) If Simulink cannot find the specified file name in the working directory, it displays an error message.

Sample time

Sample rate of data read from the file.

Characteristics	Sample Time	Inherited from driven block
	Scalar Expansion	No
	Dimensionalized	1-D array only
	Zero Crossing	No

From Workspace

Purpose	Read data from the workspace.
Library	Sources

Description

simin

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 (2-D array) or a structure containing an array of signal values and time steps. The format of the matrix or structure is the same as that used to load inport data from the workspace (see "Loading Input from the Base Workspace" on page 5-19). The From Workspace icon displays the expression in the **Data** parameter.
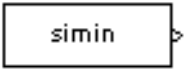
Note You must use the structure-with-time format to load matrix (2-D) data from the workspace. You can use either the array or the structure format to load scalar or vector (1-D) data.

The From Workspace block's **Interpolate data** parameter determines the block's output in the time interval for which workspace data is supplied. If the **Interpolate data** option is selected, the block interpolates between data values for time steps that occur between the times for which data is supplied from the workspace. Otherwise, the block uses the most recent data value supplied from the workspace.

The block's **Form output after final data value by** parameter determines the block's output after the last time step for which data is available from the workspace. The following table summarizes the output block based on the options that the parameter provides.

Form Output Option	Interpolate Option	Block Output After Final Data
Extrapolate	On	Extrapolated from final data value
Extrapolate	Off	Error
SettingToZero	On	Zero
SettingToZero	Off	Zero

Form Output Option	Interpolate Option	Block Output After Final Data
HoldingFinalValue	On	Final value from workspace
HoldingFinalValue	Off	Final value from workspace
CyclicRepetition	On	Error
CyclicRepetition	Off	Repeated from workspace. This option is valid only for workspace data in structure-without-time format.

If the input array contains more than one entry for the same time step, Simulink uses the signals specified by the last entry. For example, suppose the input array has this data.

```
time:     0 1 2 2
signal:   2 3 4 5
```

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 9-251) if the output is in structure-with-time format (see "Loading Input from the Base Workspace" on page 5-19 for a description of these formats).

Data Type Support

A From Workspace block accepts real or complex signals of any type from the workspace. Real signals of type double may be in either structure or matrix format. Complex signals and real signals of any type other than double must be in structure format.

From Workspace

Parameters and Dialog Box
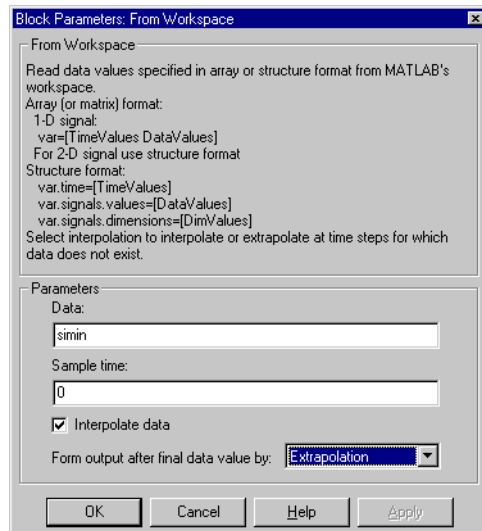
Data

An expression that evaluates to an array or a structure containing an array of simulation times and corresponding signal values. For example, suppose that the workspace contains a column vector of times named T and a vector of corresponding signal values named U. Then entering the expression, [T, U], for this parameter yields the required input array. If the required signal-versus-time array 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 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.

Form output after final data value by

Select method for generating output after the last time point for which data is available from the workspace.

Characteristics	Sample Time	Inherited from driven block
	Scalar Expansion	No
	Dimensionalized	Yes
	Zero Crossing	No

Function-Call Generator

Purpose	Execute a function-call subsystem a specified number of times 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 outports 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.
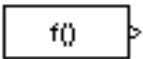
Data Type Support

A Function-Call block outputs a real signal of type doubl e.

Parameters and Dialog Box
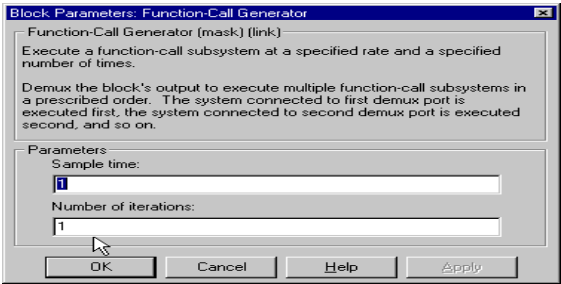
Sample time

The time interval between samples.

Number of iterations

Number of times to execute block per time step.

Characteristics

Direct Feedthrough	No
Sample Time	User-specified
Scalar Expansion	No

Dimensionalized	Yes
Zero Crossing	No

Gain

Purpose   Multiply block input by a specified value.

Library   Math

Description   The Gain block generates its output by multiplying its input by a specified gain factor. You can enter the gain as a numeric value, or as a variable or expression in the **Gain** parameter field. The input and gain can be a scalar, vector, or matrix. The **Multiplication** parameter lets you specify whether to use element-by-element or matrix multiplication of the input by the gain.
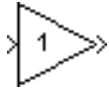
The Gain block icon displays the value entered in the **Gain** parameter field if the block is large enough. If the gain is specified as a variable, the block displays the variable's name.

To modify the gain during a simulation using a slider control (see "Slider Gain" on page 9-232).

Data Type Support   The Gain block's support for data types depends on whether you select matrix or element-wise multiplication.

For matrix multiplication, the input and the gain must be a real or complex scalar, vector, or matrix value of type si ngl e or doubl e.

For element-wise multiplication, a Gain block accepts a real- or complex-valued scalar, vector, or matrix input of any data type except bool ean 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, vector, or matrix 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 ui nt 8 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 output type is int8, the actual output is 127 if the computed output is greater than 127 and -128 if the computed output is less than -128.

Parameters and Dialog Box
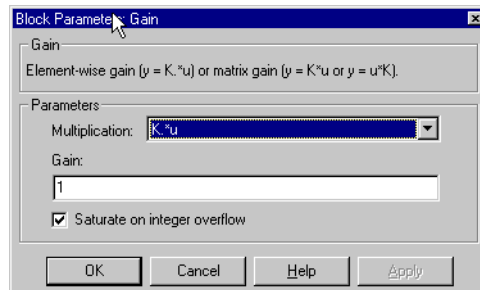
Multiplication

Specifies the type of operation used to multiply the input:

- K.*u (element-wise multiplication)
- K*u (matrix multiplication with the gain as the left operand)
- u*K (matrix multiplication with the gain as the right operand)

Gain

The gain, specified as a scalar, vector, matrix, variable name, or expression. The default is 1. If not specified, the data type of the **Gain** parameter is double. If the **Gain** parameter value is too long to be displayed in the block and element-wise multiplication is selected, the string –K– is displayed.

Saturate on integer overflow

This option is enable only for element-wise multiplication. 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 Pane" on page 5-26).

Gain

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of input and **Gain** parameter
States	0
Dimensionalized	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 9-97. 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.

                       Note A scoped Goto block in a masked system is visible only in that
                       subsystem and in the subsystems it contains. Simulink generates an error if
                       you run or update a diagram that has a Goto Visibility block at a higher level
                       in the block diagram than the corresponding scoped Goto block in the masked
                       subsystem.

                       Use local tags when the Goto and From blocks using the same tag name reside
                       in the same subsystem. You must use global or scoped tags when the Goto and
                       From blocks using the same tag name reside in different subsystems. When
                       you define a tag as global, all uses of that tag access the same signal. A tag

defined as scoped can be used in more than one place in the model. This example shows a model that uses two scoped tags with the same name (A).
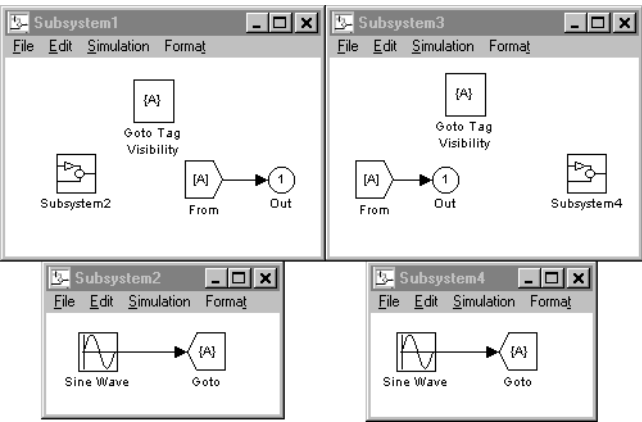
Data Type Support

A Goto block accepts real or complex signals of any data type.

Parameters and Dialog Box
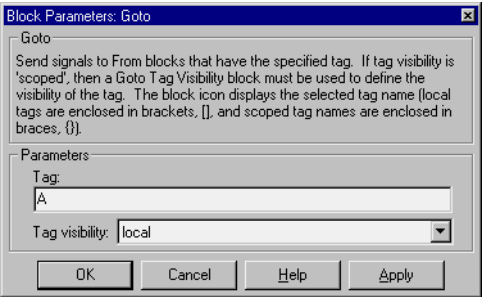
Tag

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**.

Characteristics

Sample Time	Inherited from driving block
Dimensionalized	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}

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   Not applicable.
Support

Parameters
and Dialog Box

Block Parameters: Goto Tag Visibility
— GotoTagVisibility
Used in conjunction with Goto and From blocks to define the visibility of
scoped tags. For example, if this block resides in a subsystem (or root
system) called MYSYS, then the tag is visible to From blocks that reside in
MYSYS or in subsystems of MYSYS.

— Parameters
Goto tag:
A

OK   Cancel   Help   Apply
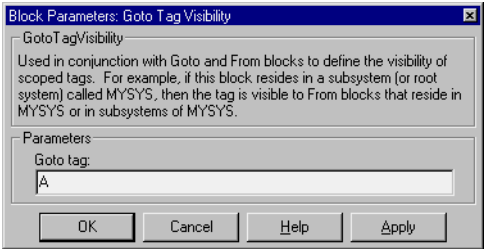
Goto tag
The Goto block tag whose visibility is defined by the location of this block.

Characteristics Sample Time   N/A

Dimensionalized   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.
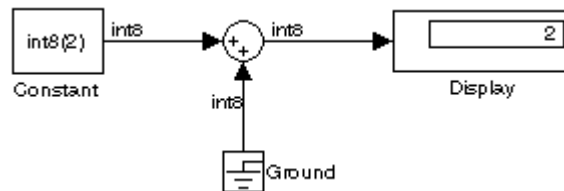
In this example, the output of the constant block determines the data type (int8) of the port to which the ground block is connected. That port in turn determines the type of the signal output by the ground block.

Parameters and Dialog Box
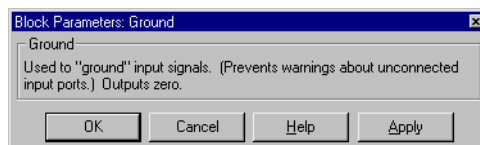
Characteristics   Sample Time   Inherited from driven block

                      Dimensionalized   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.
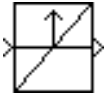
The block accepts one input of type doubl e. If the **Show output port** check box
is selected, the block output indicates when the crossing occurs. If the input
signal is exactly the value of the offset value after the hit crossing is detected,
the block continues to output a value of 1. If the input signals at two adjacent
points bracket the offset value (but neither value is exactly equal to the offset),
the block outputs a value of 1 at the second time step. If the **Show output port**
check box is *not* selected, the block ensures that the simulation finds the
crossing point but does not generate output.

When the block's **Hit crossing direction** parameter is set to ei ther, the block
serves as an "Almost Equal" block, useful in working around limitations in
finite mathematics and computer precision. Used for these reasons, this block
may be more convenient than adding logic to your model to detect this
condition.

The hardstop and cl utch 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 cl utch demo, the Hit Crossing block is in the Lockup Detection
subsystem.

Data Type   A Hit Crossing block outputs a signal of type bool ean if Boolean logic signals
Support   are enabled (see "Enabling Strict Boolean Type Checking" on page 4-48).
Otherwise, the block outputs a signal of type doubl e.

Parameters and Dialog Box
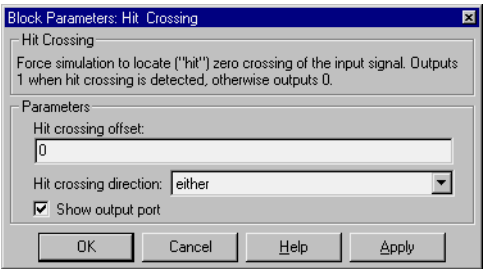
Hit crossing offset

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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	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.
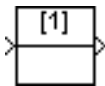
For example, these blocks illustrate how the IC block initializes a signal labeled "test signal."
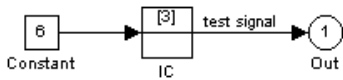
At t = 0, the signal value is 3. Afterwards, the signal value is 6.

The IC block is also useful in providing an initial guess for the algebraic state variables in the loop. For more information, see "Algebraic Loops" on page 3-18.

Data Type Support   A IC block accepts and outputs a signal of type doubl e.

Dialog Box
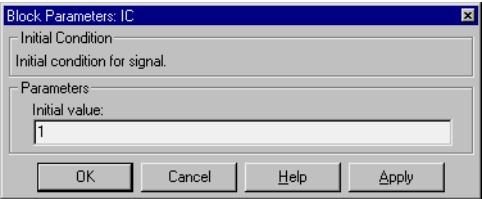
Initial value

   The initial value for the signal. The default is 1.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Parameter only
States	0
Dimensionalized	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.

You can specify the dimensions of the input to the Inport block , using the **Port dimensions** 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**

Inport

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 5-19) or the `ut` argument of the `sim` command (see `sim` on page 5-37).

- 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 can 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 array 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 outport, the input elements must be of the same type. For example, consider the follow enabled subsystem.
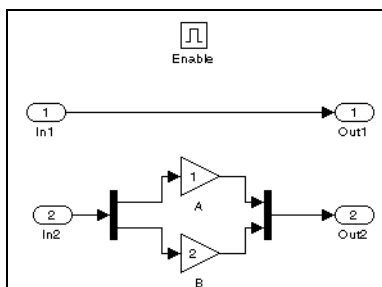
In this example, the elements of a signal vector connected to In1 must be of the same type. The elements connected to In2, however, may be of differing types.

Parameters and Dialog Box
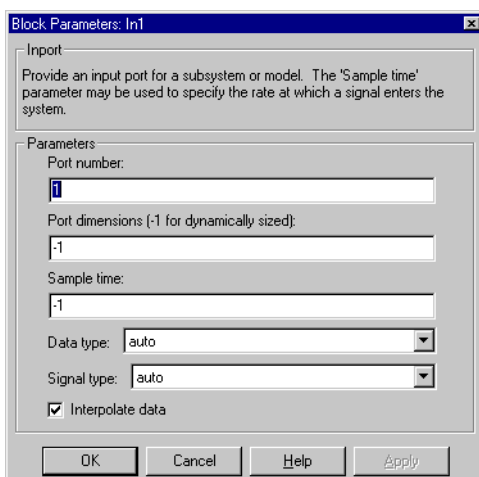
Port number

The port number of the Inport block.

Port dimensions

Dimensions of the input signal to the Inport block. Specify -1 to have it automatically determined.

Sample time

The rate at which the signal is coming into the system.

Data type

The data type of the external input.

Signal type

The signal type (real or complex) of the external input.

Note The next parameter applies only to root-level inports. It does not appear on subsystem inport dialogs.

Interpolate data

Selecting this option causes this block, when loading data from the workspace, to interpolate or extrapolate output at time steps for which no corresponding workspace data exists. See "Loading Input from the Base Workspace" on page 5-19 for more information.

Characteristics	Sample Time	Inherited from driving block
	Dimensionalized	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:
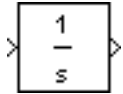
$$\frac{1}{s}$$

- 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, 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.

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 reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The output is determined as follows:

- When the integral is less than or equal to 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 or equal to 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.
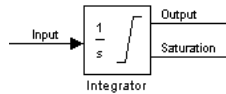
The signal has one of three values:

- 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.
- Select level to trigger the reset and hold the output to the initial condition while the reset signal is nonzero.

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.

Integrator

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 5-13.

Choosing All Options

When all options are selected, the icon looks like this.
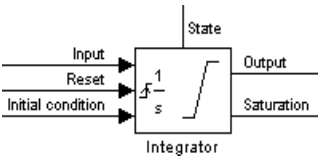
Data Type Support

An Integrator block accepts and outputs signals of type doubl e on its data ports. Its external reset port accepts signals of type doubl e or bool ean.

Parameters and Dialog Box
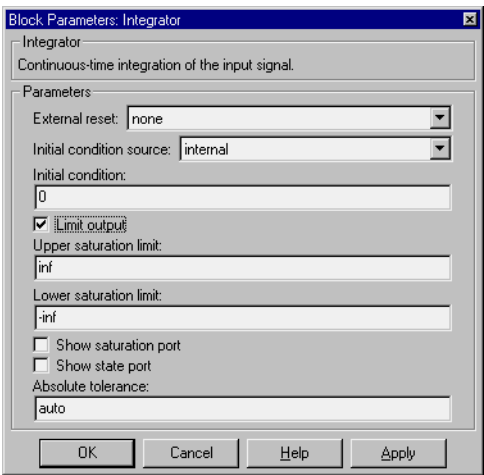
External reset

Resets the states to their initial conditions when a trigger event (ri si ng, fal l i ng, ei ther, or l evel) 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.

Characteristics	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
	Dimensionalized	Yes
	Zero Crossing	If the **Limit output** option is selected, one for detecting reset; one each to detect upper and lower saturation limits, one when leaving saturation

Interpolation (n-D) Using PreLook-Up

Purpose
Perform high performance constant or linear interpolation, mapping N input values to a sampled representation of a function in N variables via output from PreLook-Up Index Search block.

Library
Functions & Tables

Description
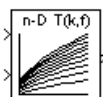
The Interpolation (n-D) Using PreLook-Up block uses the precalculated indices and interval fractions from the PreLook-Up Index Search block to perform the equivalent operation that the Look-Up Table (n-D) performs. By using this combination of blocks, multiple Interpolation (n-D) blocks can be fed by one set of PreLook-Up Index Search blocks. In models that have many interpolation blocks, simulation performance be greatly increased.

This block supports two interpolation methods: flat (constant) interval look-up and linear interpolation. These operations can be applied to 1-D, 2-D, 3-D, 4-D and higher dimensioned tables.

You define a set of output values as the **Table data** parameter. These table values must correspond to the breakpoint data sets that are in the PreLook-Up Index Search block. The block generates its output by interpolating the table values based on the (index,fraction) pairs fed into the block by each PreLook-Up Index Search block.

The block generates output based on the input values:

• If the inputs match breakpoint parameter values, the output is the table value at the intersection of the row, column and higher dimensions' breakpoints.

• If the inputs do not match row and column parameter values, the block generates output by 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 in each corresponding dimension.

Data Type Support
An Interpolation (n-D) Using PreLook-Up block accepts signals of types double or single, but for any given block, the inputs must all be of the same type. The **Table data** parameter must be of the same type as the inputs. The output data type is set to the **Table data** data type.

Parameters and Dialog Box
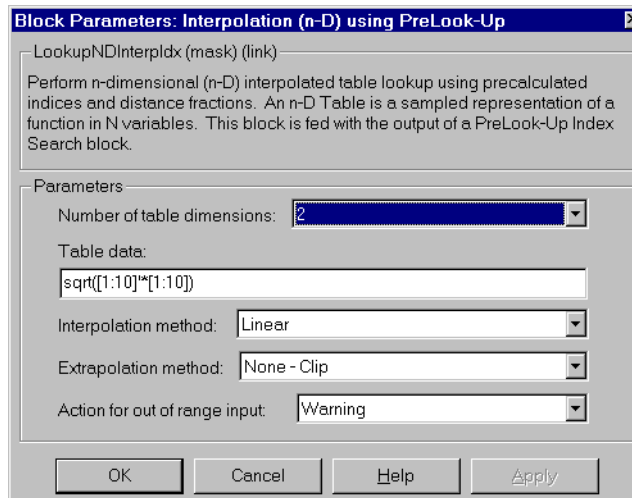
Number of table dimensions

The number of dimensions that the **Table data** parameter must have. This determines the number of independent variables for the table and hence the number of inputs to the block (see descriptions for "Explicit Number of dimensions" and "Use one (vector) input port instead of N ports," below).

Table data

The table of output values. The matrix size must match the dimensions defined by the N **breakpoint set** parameter or by the **Explicit number of dimensions** parameter when the number of dimensions exceeds four. During block diagram editing, you can leave the **Table data** field empty, but for running the simulation, you must match the number of dimensions in the **Table data** to the **Number of table dimensions**. For information about how to construct multidimensional arrays in MATLAB, see Multidimensional Arrays in MATLAB's online documentation.

Interpolation method

None (flat) or Linear.

Extrapolation method
None (clip) or Linear.

Action for out of range input
None, Warning, Error.

Characteristics

Direct Feedthrough	Yes,
Sample Time	Inherited from driving blocks
Scalar Expansion	Yes
Zero Crossing	No

Purpose   Perform the specified logical operation on the input.

Library   Math
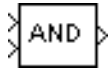
Description   The Logical Operator block performs any of these logical operations on its
inputs: AND, OR, NAND, NOR, XOR, and NOT. The output depends on the
number of inputs, their dimensionality, and the selected operator. The output
is 1 if TRUE and 0 if FALSE. The block icon shows the selected operator. The
following rules apply to the inputs and outputs of the block:

- If the block has more than one input, any nonscalar inputs must have the
  same dimensions. For example, if any input is a 2-by-2 array, all other
  nonscalar inputs must also be 2-by-2 arrays.

- Scalar inputs are expanded to have the same dimensions as the nonscalar
  inputs.

- If the block has more than one input, the output has the same dimensions as
  the inputs (after scalar expansion) and each output element is the result of
  applying the specified logical operation to the corresponding input elements.
  For example, if the specified operation is AND and the inputs are 2-by-2
  arrays, the output is a 2-by-2 array whose top, left element is the result of
  applying AND to the top, left elements of the inputs, etc.

- If the block has a single input and the specified operator is not the NOT
  operator, the input must be vector-like, i.e. a scalar, a 1-D array, or a one-row
  or one-column 2-D array. The output is a scalar value equal to the result of
  applying the operation to the elements of the input.

- If the specified operation is NOT, the block accepts only one input. The
  output has the same dimensions as the input and contains the logical
  complements of the elements of the input.

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**   A Logical Operator block accepts only signals of type bool ean on its input ports,
if Boolean logic signals are enabled (see "Enabling Strict Boolean Type
Checking" on page 4-48). Otherwise, the block also accepts inputs of type
doubl e. A nonzero input of type doubl e 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.

Logical Operator

Parameters and Dialog Box

Block Parameters: Logical Operator

─ Logical Operator ──────────────────────────────

Logical operators. For a single input, operators are applied across the input vector. For multiple inputs, operators are applied across the inputs.

─ Parameters ──────────────────────────────

Operator: AND ▾

Number of input ports:

2

[OK] [Cancel] [Help] [Apply]
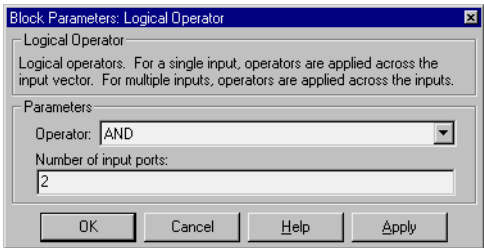
Operator

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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of inputs
Dimensionalized	Yes
Zero Crossing	No

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.
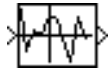
You define the table by specifying (either as row or column vectors) the **Vector of input values** and **Vector of output values** parameters. The block produces an output value by comparing the block input with values in the input vector:

- 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 9-136.

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]
```
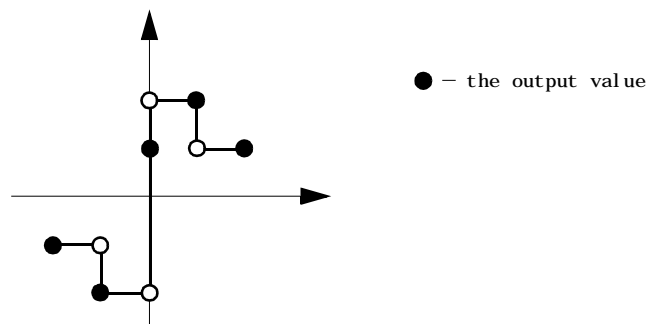

● — the output value

This example has three step discontinuities: at u = -1, 0, and +1.

Look-Up Table

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 doubl e.

Parameters and Dialog Box
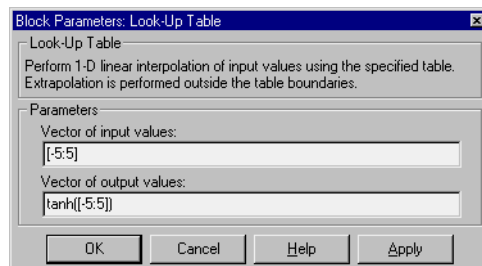
Vector of input values

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.

Vector of output values

The vector of values containing block output values. This vector must be the same size as the input vector.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	No
Dimensionalized	Yes
Zero Crossing	No

Look-Up Table (2-D)

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.
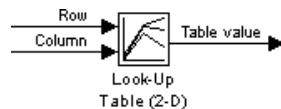
The block generates output based on the input values:

- 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 9-133).

Example
In this example, the block parameters are defined as:

```
Row:       [1 2]
Column:    [3 4]
Table:     [10 20; 30 40]
```

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).
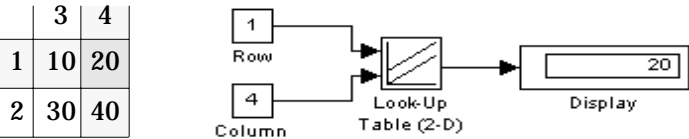
In the second figure, the first input is 1.7 and the second is 3.4. These values cause the block to interpolate between row and column values, as shown in the table at the left. The value at the intersection (28) is the output value.
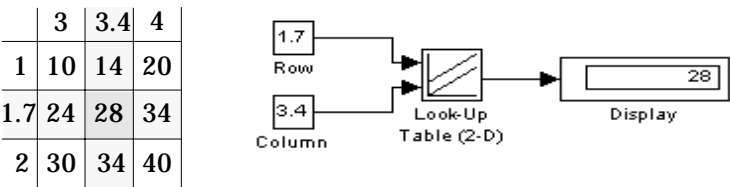
Data Type Support

A Look-Up Table (2-D) block accepts and outputs signals of type doubl e.

Parameters and Dialog Box
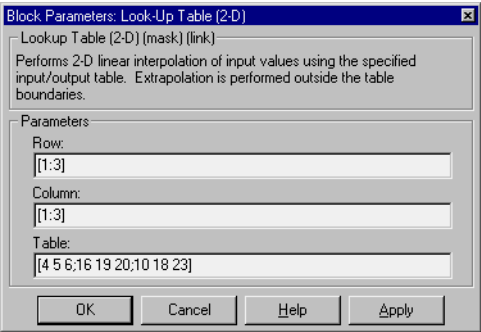
Row

> The row values for the table, entered as a vector. The vector values must increase monotonically.

Look-Up Table (2-D)

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.

Characteristics	Direct Feedthrough	Yes
	Sample Time	Inherited from driving blocks
	Scalar Expansion	Of one input if the other is a vector
	Dimensionalized	Yes
	Zero Crossing	No

Perform constant, linear or spline interpolated mapping of N input values to a sampled representation of a function in N variables.

Library   Functions & Tables

Description   The Look-Up Table (n-D) block evaluates a sampled representation of a function in N variables by interpolating between samples to give an approximate value for $y = F(x1, x2, x3, ..., xn)$, even when the function F is known only empirically. The block efficiently maps the block inputs to the output value using interpolation on a table of values defined by the block's parameters. Interpolation methods supported are:

- Flat (constant)
- Linear
- Natural (cubic) spline

You can apply any of these methods to 1-D, 2-D, 3-D or higher dimensional tables.

You define a set of output values as the **Table data** parameter and the values that correspond to its rows, columns and higher dimensions with the *N*th breakpoint set parameter. The block generates an output value by comparing the block inputs with the breakpoint set parameters. The first input identifies the first dimension (row) breakpoints, the second breakpoint set identifies a column, and so on, as shown by this figure.
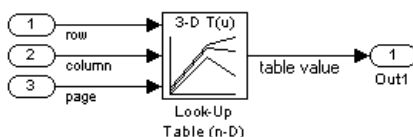
If you are unfamiliar with how to construct N-dimensional arrays in MATLAB, see Multidimensional Arrays in MATLAB's online documentation.

The block generates output based on the input values:

- If the inputs match breakpoint parameter values, the output is the table value at the intersection of the row, column and higher dimensions breakpoints.

Look-Up Table (n-D)

- If the inputs do not match row and column parameter values, the block generates output by interpolating between the appropriate table values. If any of the block inputs are outside the ranges of their respective breakpoint sets, the block will limit the input values to the breakpoint set's range in that dimension. If extrapolation is enabled, it extrapolates linearly or by using a cubic polynomial (if you selected cubic spline extrapolation).

Note As an alternative, you can use the Look-Up Table (n-D) block with the PreLook-Up Index Search block to have more flexibility and potentially much higher performance for linear interpolations in certain circumstances.

For non-interpolated table look-ups, use the Direct Look-Up Table, (n-D) block when the look-up operation is a simple array access, for example, if you have an integer value k and you merely want the k-th element of a table, *y = table(k)*.

Data Type Support

An n-D Interpolated Look-Up Table block accepts signals of types double or single, but for any given n-D Interpolated Look-Up Table block, the inputs must all be of the same type. Table data and Breakpoint set parameters must be of the same type as the inputs. The output data type is also set to the input data type.

**Parameters
and Dialog Box**
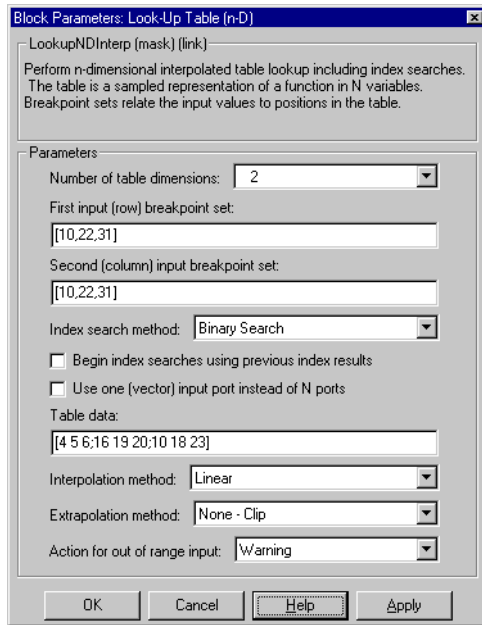
Number of table dimensions

> The number of dimensions that the **Table data** parameter is to have. This
> determines the number of independent variables for the table and hence
> the number of inputs to the block (see descriptions for "Explicit Number of
> dimensions" and "Use one (vector) input port instead of N ports", below).

First input (row) breakpoint set

> The row values represented in the table, entered as a vector. The vector
> values must increase monotonically. This field is always visible.

Second (column) input breakpoint set

> The column values for the table, entered as a vector. The vector values
> must increase monotonically. This field is visible if the **Number of table
> dimensions** popup is 2, 3, 4 or More.

Third ... Nth input breakpoint set

> The values corresponding to the third dimension for the table, entered as a
> vector. The vector values must increase monotonically. This field is visible
> if the **Number of table dimensions** is 3, 4 or More.

Fourth input breakpoint set

The values corresponding to the fourth dimension for the table, entered as a vector. The vector values must increase monotonically. This field is visible if the **Number of table dimensions** is 4 or More.

Fifth..Nth input breakpoint sets (cell array)

The cell array of values corresponding to the third, fourth, or higher dimensions for the table, entered as a 1-D cell array of vectors. For example, {[10: 10: 30], [0: 10: 100]} is a cell array of two vectors that will be used for the fifth and sixth dimensions' breakpoint sets. The vector values must increase monotonically. This field is visible if the **Number of table dimensions** is More.

Explicit number of dimensions

The number of table dimensions when the number is five or more. This is indicated when you set the **Number of table dimensions** field to More.

Index search method

Choose "Evenly Spaced Points", "Linear Search" or "Binary Search" (default). Each search method has speed advantages over the others in different circumstances. A suboptimal choice of index seach method can lead to slow performance in models that rely heavily on look-up tables. If the breakpoint data are evenly spaced, e.g., 10, 20, 30, ..., you can achieve the greatest speed by selecting "Evenly Spaced Points" to directly calculate the indices into the table. For irregularly spaced breakpoint sets, if the input signals do not vary much from one time step to the next, selecting "Linear Search" and "Begin index searches using previous index results" at the same time will produce the best speed performance. For irregularly spaced breakpoint sets with rapidly varying input signals that jump more than one or two table intervals per time step, selecting "Binary Search" will give the best speed performance. Note that the "Evenly Spaced Points" algorithm only makes use of the first two breakpoints in determining the offset and spacing of the rest of the points.

Begin index searches using previous index results

Activating this option will cause the block to initialize index searches using the index found on the previous time step. This is a huge performance improvement for the block when the input signals do not change much with respect to its position in the table from one time step to the next. When this

option is deactivated, the linear search and binary search methods can take significantly longer, especially for large breakpoint data sets.

Use one (vector) input port instead of N ports

Instead of having one input port per independent variable, the block is configured with just one input port that expects a signal that is N elements wide for an N-dimensional table. This may be useful in removing line clutter on a block diagram with large numbers of tables.

Table data

The table of output values. To execute a model with this block, the matrix size must match the dimensions defined by the **N breakpoint set** parameter or by the **Explicit number of dimensions** parameter when the number of dimensions exceeds four. During block diagram ediig, you can leave this field blank since only the Number of table dimensions field is required to set the number of ports on the block.

Interpolation method

None (flat), Linear, or Cubic Spline.

Extrapolation method

None (clip), Linear, or Cubic Spline.

Action for out of range input

None, Warning, or Error. An out of range condition during simulation results in warning messages in the command window if "Warning" is selected, and the simulation halts with an error message if "Error" is selected.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving blocks
Scalar Expansion	No
Dimensionalized	No
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 doubl e. The angle input is assumed to be in radians. The data type of the complex output signal is doubl e.

The inputs may be both signals of equal dimensions, or one input may be an array and the other a scalar. If the block has an array input, the output is an array 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
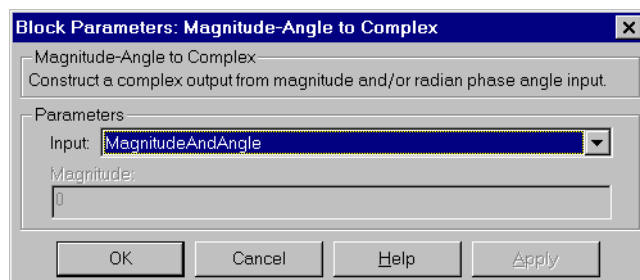
Input

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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs
Dimensionalized	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.
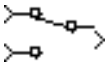
Data Type   A Manual Switch block accepts all input types. Both inputs must be of the same
Support   numeric and data type. The block's output has the same numeric type (real or
                     complex) and data type as its input.

Parameters   None
and Dialog Box

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Zero Crossing	No

Purpose	Perform a mathematical function.

Library	Math

Description

The Math Function block performs numerous common mathematical functions.
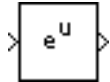
You can select one of these functions from the **Function** list: exp, log, 10^u, log10, magnitude^2, square, sqrt, pow, conj, reciprocal, hypot, rem, mod, transpose, and hermitian. The block output is the result of the function operating on the input or inputs.

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 vector or matrix 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

Block Parameters: Math Function

Math

Mathematical functions including logarithmic, exponential, power, and modulus functions.

Parameters

Function: exp

Output signal type: auto

OK   Cancel   Help   Apply
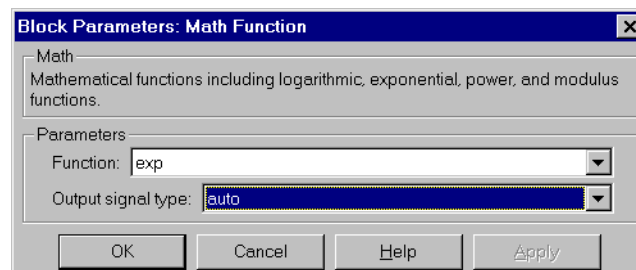
Function

The mathematical 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 Signal	Output Signal Type		
		Auto	Real	Complex
Exp, `log`, `10`^{u}, `log10`, `square`, `sqrt`, `pow`, `reciprocal`, `conjugate`, `transpose`, `hermitian`	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
Dimensionalized	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 output of the function must match the output dimensions of the block or an error occurs.
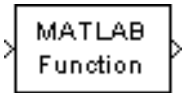
Here are some sample valid expressions for this block.

```
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

9-149
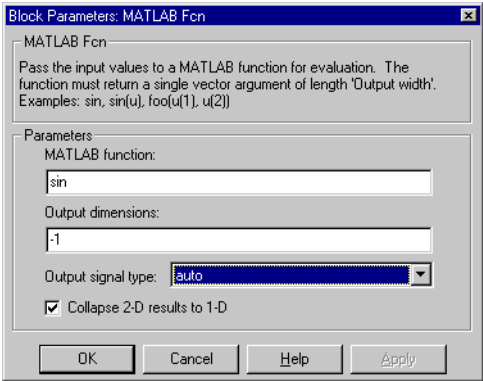
MATLAB Fcn

MATLAB function

The function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

Output dimensions

The output dimensions. If the output dimensions are to be the same as the input dimensions, specify -1. Otherwise, you must specify the correct dimensions 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.

Collapse 2-D results to 1-D

Outputs a 2-D array as a 1-D array containing the 2-D array's elements in column-major order.

Characteristics	Direct Feedthrough	Yes
	Sample Time	Inherited from driving block
	Scalar Expansion	N/A
	Dimensionalized	Yes
	Zero Crossing	No

Purpose   Concatenate inputs horizontally or vertically.

Library   Signals & Systems
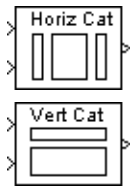
Description   The Matrix Concatenation block concatenates input matrices u_1, u_2, ..., u_n along rows or columns, where n is specified by the **Number of inputs** parameter. The block accepts inputs with any combination of built-in Simulink data types. If all inputs are sample-based, the output is sample-based. Otherwise, the output is frame-based.

Horizontal Matrix Concatenation

When the **Concatenation method** parameter is set to **Horizontal**, the block concatenates the input matrices along *rows*.

```
y = [u1 u2 u3 ... un]          % Equivalent MATLAB code
```

For horizontal concatenation, inputs must all have the same row dimension, M, but may have different column dimensions. The output matrix has dimension M-by-ΣN_i, where N_i is the number of columns in input u_i (i = 1, 2, ..., n).

When some of the inputs are length-M 1-D vectors while others are M-by-N_i matrices, the vector inputs are treated as M-by-1 matrices.

Vertical Matrix Concatenation

When the **Concatenation method** parameter is set to **Vertical**, the block concatenates the input matrices along *columns*.

```
y = [u1; u2; u3; ...; un]          % Equivalent MATLAB code
```

For vertical concatenation, inputs must all have the same column dimension, N, but may have different row dimensions. The output matrix has dimension ΣM_i-by-N, where M_i is the number of rows in input u_i (i = 1, 2, ..., n).

When some of the inputs are length-M_i 1-D vectors while others are M_i-by-1 matrices, the vector inputs are treated as M_i-by-1 matrices. (1-D vector inputs are not accepted for vertical concatenation when the other inputs have column dimension greater than 1.)

Matrix Concatenation

1-D Vector Concatenation

When all inputs to the Matrix Concatenation block are length-M_i 1-D vectors, the output is a ΣM_i-by-1 matrix containing all input elements concatenated in port order: the elements in the vector input to the top port appear as the first elements in the output, and the elements in the vector input to the bottom port appear as the last elements in the output.

Dialog Box
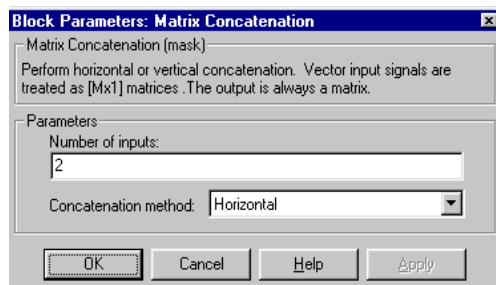
Number of inputs

The number of matrices to concatenate.

Concatenation method

The dimension along which to concatenate the inputs.

Purpose   Multiply the input by a matrix.

Library   Math

Description   The Matrix Gain block is the Gain block with its parameters set to default values appropriate for a matrix gain. See the Gain block for more information.
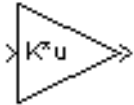
Data Type Support   See the Gain block.

Parameters and Dialog Box
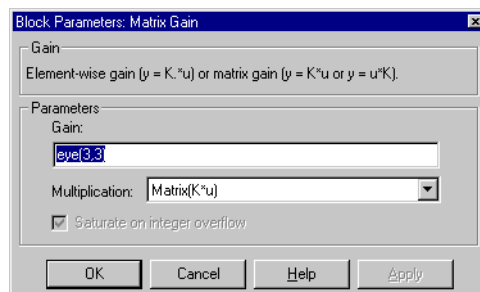
Gain

The gain, specified as a matrix. The default is eye(3, 3). See the Gain block for more information.

Multiplication

Type of multiplication used to multiply the input signal by the gain. The default is set for matrix multiplication. See the Gain block for more information.

Saturate on Integer Overflow

Applies only to element-wise multiplication. See the Gain block for more information.

Matrix Gain

Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	No
States	0
Dimensionalized	Yes
Zero Crossing	No

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.
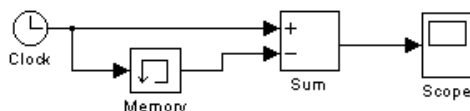
Note Avoid using the Memory block when integrating with ode15s or ode113, unless the input to the block does not change.

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
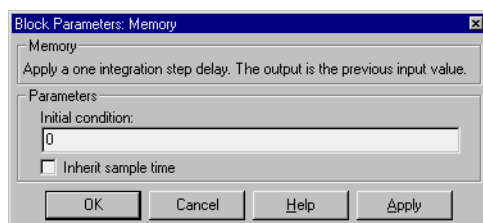
Initial condition

   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.

Memory

Characteristics

Direct Feedthrough	No
Sample Time	Continuous, but inherited if the **Inherit sample time** check box is selected
Scalar Expansion	Of the **Initial condition** parameter
Dimensionalized	Yes
Zero Crossing	No

Purpose   Combine multiple signals into a single signal.

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.

Note Merge blocks facilitate creation of alternately executing subsystems.
See "Creating Alternately Executing Subsystems" on page 7-12 for an
application example.

A Merge block does not accept signals whose elements have been reordered. For
example, in the following diagram,
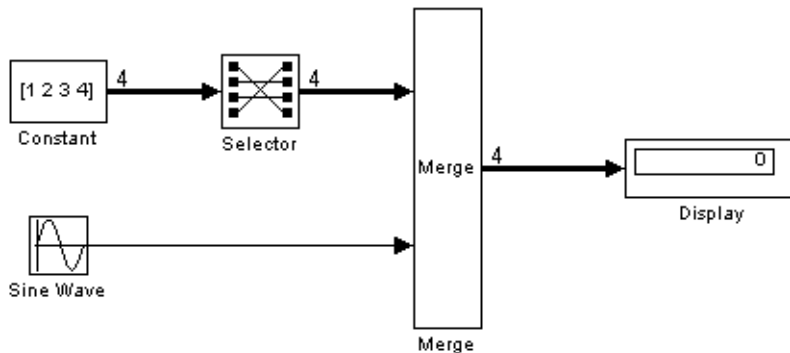
the Merge block does not accept the output of the Selector block because the
Selector block interchanges the first and fourth elements of the vector signal.

If the block's **Allow unequal port widths** option is not selected, the block
accepts only inputs of equal dimensions and outputs a signal of the same
dimensions as the inputs. If the **Allow unequal port widths** option is selected,
the block accepts scalars and vectors (but not matrices) having differing
numbers of elements. Further, the block allows you to specify an offset for each

input signal relative to the beginning of the output signal. The width of the output signal is $\max(w_1+o_1, \ w_2+o_2, \ \ldots \ w_n+o_n)$ where $w_1, \ \ldots \ w_n$ are the widths of the input signals and $o_1, \ \ldots \ o_n$ are the offsets for the input signals. For example, the Merge block in the following diagram
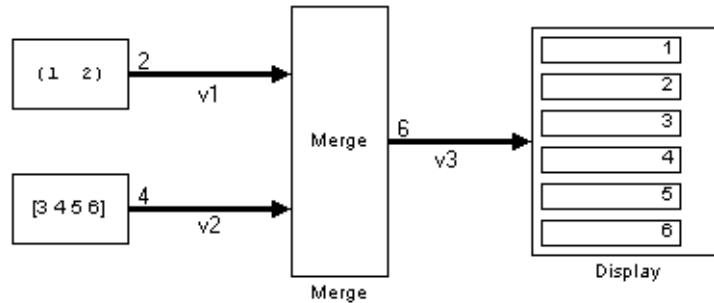
merges signals v1 and v2 to produce signal v3. In this example, the offset of v1 is 0 and the offset of v2 is 2, resulting in an output signal six elements wide. The Merge block maps the elements of v1 to the first two elements of v3 and the elements of v2 to the last four elements of v3.

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.

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
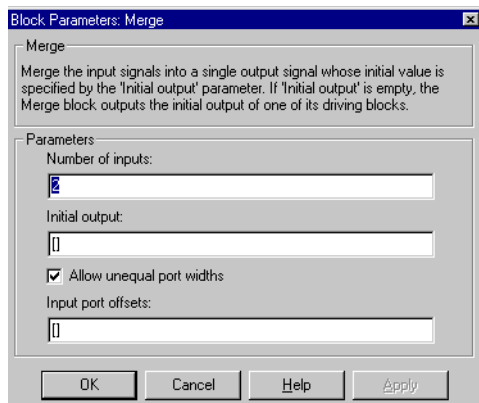
Number of inputs

The number of input ports to merge.

Initial output

Initial value of output. If unspecified, the initial output equals the initial output, if any, of one of the driving blocks.

Allow unequal port widths

Allows the block to accept inputs having different numbers of elements.

Input port offsets

Vector specifying the offset of each input signal relative to the beginning of the output signal.

Characteristics	Sample Time	Inherited from the driving block
	Dimensionalized	Yes
	Scalar Expansion	No

MinMax

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 input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

If the block has multiple input ports, the nonscalar inputs must all have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

Data Type Support

A MinMax block accepts and outputs real-valued signals of any data type.

Parameters and Dialog Box
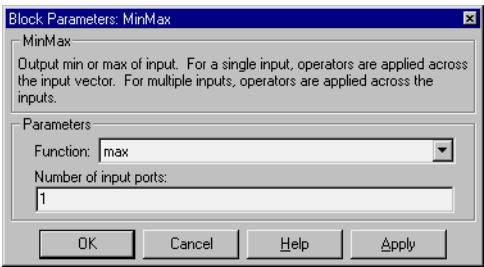
Function

    The function (min or max) to apply to the input.

Number of input ports

    The number of inputs to the block.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from the driving block

Scalar Expansion	Of the inputs
Dimensionalized	Yes
Zero Crossing	Yes, to detect minimum and maximum values

Model Info

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

van der Pol Equation
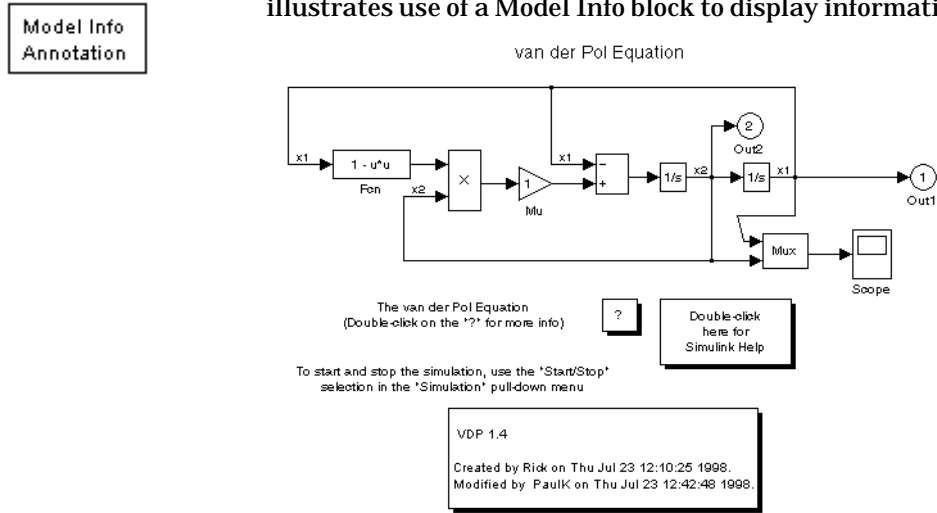
A Model Info block can show revision control information embedded in the
model itself and/or information maintained by an external revision control or
configuration management system. A Model Info block's dialog allows you to
specify the content and format of the text displayed by the block.

Data Type   Not applicable.
Support
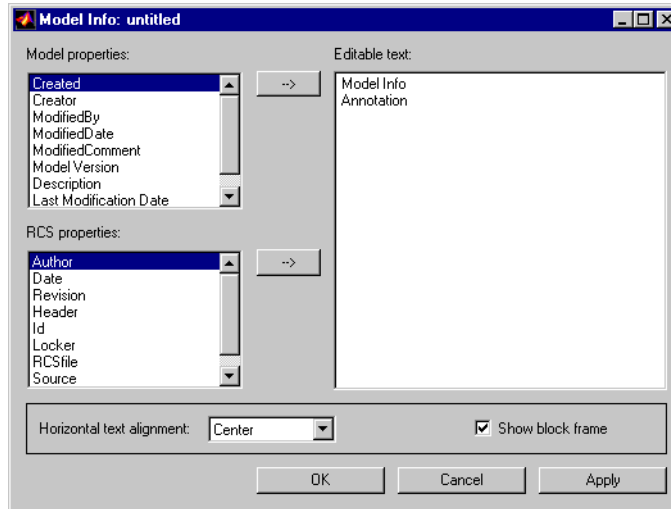
Dialog Box

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

Model Info

Control Properties" on page 4-111 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 4-107). 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.

Purpose   Choose between block inputs.

Library   Nonlinear

Description   The Multiport Switch block chooses between a number of inputs.
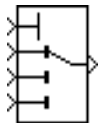
The first (top) input is the control input and the other inputs are data inputs. The value of the control input determines which data input to pass through to the output port.

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.

(Truncated) Control Input	Passes This Data Input
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 an array, the output is an array. Any scalar inputs are expanded to arrays.

• If the block has only one data input, the input must be a scalar or a vector (1-D array). If the input is a vector, the block output is the element of the vector that corresponds to the truncated value of the control input.

Multiport Switch

Data Type Support   The control input of a Multiport Switch block accepts a real-valued signal of any built-in data type except bool ean. 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

Block Parameters: Multiport Switch

Multi-Port Switch
Pass through the input signals corresponding to the rounded value of the first input.

Parameters
Number of inputs:
3

| OK | Cancel | Help | Apply |
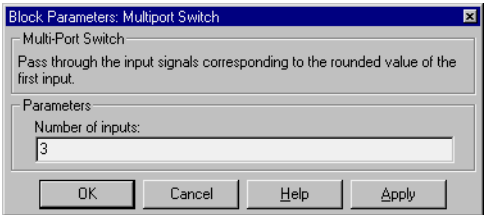
Number of inputs
The number of data inputs to the block.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block(s)
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	No

Purpose   Combine several input signals into a vector or bus output signal.

Library   Signals & Systems

Description   The Mux block combines its inputs into a single output. An input can be a scalar, vector, or matrix signal. Depending on its inputs, the output of a Mux block is a vector or a composite signal, i.e., a signal containing both matrix and vector elements. If all of a Mux block's inputs are vectors or vector-like, the block's output is a vector. A *vector-like* signal is any signal that is a scalar (one-element vector), a vector, or a single-column or single-row matrix. If any input is a nonvector-like matrix signal, the output of the Mux is a bus signal. Bus signals can drive only virtual blocks, e.g., a Demux, Subsystem, or Go To block.

The Mux block's **Number of Inputs** parameter allows you to specify input signal names and dimensionality as well as the number of inputs. You can use any of the following formats to specify this parameter:

- Scalar

  Specifies the number of inputs to the Mux block. When this format is used, the block accepts signals of any dimensionality. Also, Simulink assigns each input the name signalN, where N is the input port number.

- Vector

  The length of the vector specifies the number of inputs. Each element specifies the dimensionality of the corresponding input. A positive value specifies that the corrresponding port can accept only vectors of that size. For example [2 3] specifies two input ports of size 2 and 3, respectively. If an input signal width does not match the expected width, Simulink displays an error message. A value of -1 specifies that the corresponding port can accept vectors or matrices of any dimensionality.

- Cell array

  The length of the cell array specifies the number of inputs. The value of each cell specifies the dimensionality of the corresponding input. A scalar value N specifies a vector of size N. A vector value [M N] specifies an MxN matrix. A value of -1 means the corresponding port can accept signals of any dimensionality.

Mux

- Signal name list

  You can enter a list of signal names separated by commas. Simulink assigns each name to the corresponding port and signal. For example, if you enter position, velocity, the Mux block will have two inputs, named position and velocity.

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
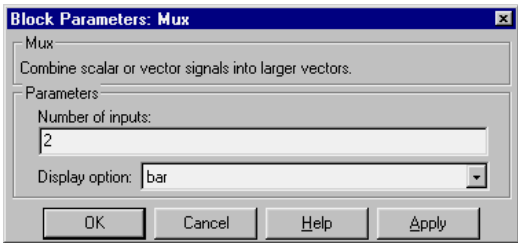
Number of inputs

The number and dimensionality of inputs. You can enter a comma-separated list of signal names for this parameter field.

Display option

The appearance of the block icon in your model.

Display Option	Appearance of Block in Model
none	Mux appears inside block icon
signals	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.

Outport Blocks in a Subsystem

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port on a Subsystem block is the block whose **Port number** parameter matches the relative position of the output port on the Subsystem block. For example, the Outport block whose **Port number** parameter is 1 sends its signal to the block connected to the 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.

Outport

The Outport block name appears in the Subsystem block icon as a port label. To suppress display of the label, select the Outport block and choose **Hide Name** from the **Format** menu.

Outport Blocks in a Conditionally Executed Subsystem

When an Outport block is in an 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** pop-up 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.

Outport Blocks in a Top-Level System

Outport 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 5-22) or the sim command (see sim on page 5-37). For example, if a system has more than one Outport block and the save format is array, the following command

   [t, x, y] = sim(...);

   writes y as a matrix, with each column containing data for a different Outport block. The column order matches the order of the port numbers for the Outport blocks.

   If you specify more than one variable name after the second (state) argument, data from each Outport block is written to a different variable. For example, if the system has two Outport blocks, to save data from Outport block 1 to speed and the data from Outport block 2 to dist, you could specify this command:

   [t, x, speed, dist] = sim(...);

- To provide a means for the linmod and trim analysis functions to obtain output from the system. For more information about using Outport blocks with analysis commands, see Chapter 5.

Numeric and Data Type Support

An Outport 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 array connected to an Outport block can be of differing numeric and data types except in the following circumstance. If the outport is in a conditionally executed subsystem and the initial output is specified, all elements of an input array must be of the same numeric and data type.

Simulink's data type conversion rules apply to an outport'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 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
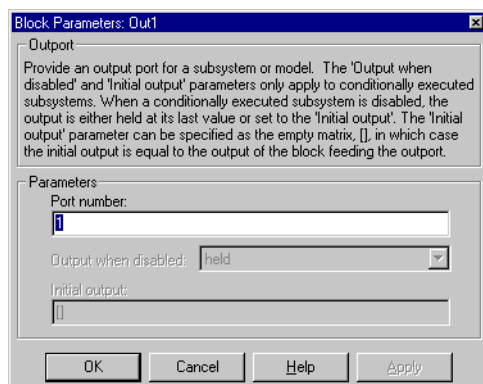
Port number

The port number of the Outport 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.

Outport

Characteristics   Sample Time   Inherited from driving block

Dimensionalized   Yes

Purpose   Perform evaluation of polynomial coefficients on input values.

Library   Functions & Tables

Description   The Polynomial block uses a coefficients parameter to evaluate a real
                    polynomial for the input value.
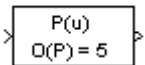
You define a set of polynomial coefficients in the form accepted by MATLAB's
polyval command. The block will then calculate *P(u)* at each time step for the
input *u*. Inputs and coefficients must be non-complex.

Data Type   The Polynomial block accepts real signals of types double or single.The
Support   **Polynomial coefficients** parameter must be of the same type as the inputs.
                    The output data type is set to the input data type.

**Parameters
and Dialog Box**
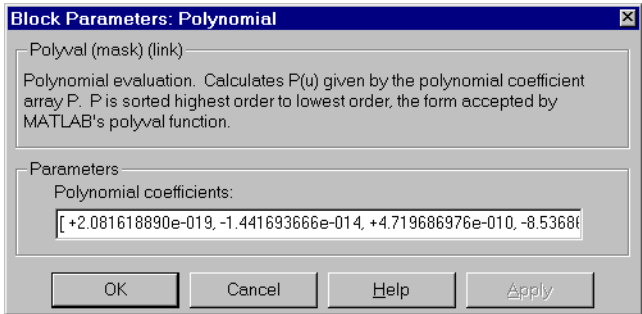
Polynomial coefficients

Values are in coefficients of a polynomial in MATLAB polyval form, with the
first coefficient representing x^N, then decreasing in order until the last
cofficient, which represents the constant for the polynomial. See polyval for
more information.

Characteristics | Direct Feedthrough | Yes |
| --- | --- |
| Sample Time | Inherited from driving block |
| Scalar Expansion | No |

Polynomial

Dimensionalized	Yes
Zero Crossing	No

Purpose   First stage of high performance constant or linear interpolation that performs index search and interval fraction calculation for input on a breakpoint set.

Library   Functions & Tables

Description   The PreLook-Up Index Search block calculates the indices and interval fractions for the input value in the **Breakpoint data** parameter.   By using this combination of blocks, multiple Interpolation (n-D) blocks can be ed by one set of PreLook-Up Index Search blocks. In models that have many interpolation blocks simulation performance be greatly increased.

To use this block, you must define a set of breakpoint values. In normal use, this breakpoint data set corresponds to one dimension of a **Table data** parameter in an Interpolation (n-D) using PreLook-Up block. The block generates a pair of outputs for each input value by calculating the index of the breakpoint set element that is less than or equal to the input value and the resulting fractional value that is a number $0 \le f < 1$ that represent's the input value's normalized position between the index and the next index value.

For example, if the breakpoint data is:

   [0 5 10 20 50 100]

and the input value u is 55, the (index, fraction) pair will be (4, 0.1), denoted as k and f on the block icon. Note that the index value is zero-based.

Data Type Support   A PreLook-Up Index Search block accepts signals of types double or single, but for any given block, the inputs must all be of the same type. The **Breakpoint data** parameter must be of the same type as the inputs. The output data type is set to the input data type.

Prelook-Up Index Search

**Parameters
and Dialog Box**

```
Block Parameters: PreLook-Up Index Search                    ⊠

─LookupIdxSearch (mask) (link)────────────────────────────
 Locate the input value's relative position within a range of numbers (the
 "breakpoint data" set). Returns an array of the interval index "k" and distance
 fraction "f" that the input "u" reaches into the kth interval.

─Parameters──────────────────────────────────────────────
  Breakpoint data:
  [0:10:100]

  Index search method:  Binary Search                    ▼

  ☐ Begin index search using previous index result

  ☐ Output only the index

  Process out of range input:  Clip to Range              ▼

  Action for out of range input:  Warning                ▼

      OK         Cancel        Help         Apply
```

Breakpoint data
The set of numbers to search.

Index search method
Binary search, evenly spaced points, or linear search. Use linear search in combination with **Begin index search using previous index result** for higher performance than a binary search when the input values do not change much from one time step to the next. For large breakpoint sets ,a linear search can be very slow if the input value changes by more than a few intervals from one time step to the next.

Begin index search using previous index result
Check this option if you want the block to start its search using the index that was found on the previous time step. For inputs that change slowly with respect to the interval size, you may realize a large performance gain.

Output only the index
If this block is not being used to feed an Interpolation (n-D) using PreLook-Up block, the interval fraction output can be dropped and the resulting index value output is a $uint32$ instead.

Process out of range input
Clip to Range or Linear Extrapolation.

Action for out of range input
None, Warning, Error.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving blocks
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	No

Product

Purpose   Generate the element-wise product, quotient, matrix product, or inverse of block inputs.

Library   Math

Description The Product block outputs the element-wise or matrix product of its inputs, depending on the values of the **Multiplication** and **Number of inputs** parameters:
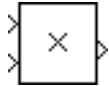
- If the value of the **Number of inputs** parameter is a combination of * and / symbols, the number of block inputs is equal to the number of symbols. The block icon shows the appropriate symbol adjacent to each input port.

   For example, entering */ as the parameter value results in the block icon
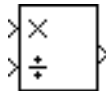
   when the the **Multiplication** parameter is element-wise.

   If the value of the **Multiplication** parameter is element-wise, the block output is the element-by-element product of all inputs marked * divided by all inputs marked /. For example, if the inputs are vectors of size n, the output is a vector of size n each of whose elements equals

   $$y_i = u1_i \times u2_i \times \ldots \times un_i$$

   (To create the dot-product of input vectors, use the Dot Product block.

   If any input is a matrix, all inputs must be a matrix or a scalar where a scalar is defined as a 1-by-1 matrix or a 1-element vector. If any input is a vector, all inputs must be vector-like. A vector-like input is any input that is either a scalar, a vector, or a column matrix or a row matrix. All nonscalar inputs must have the same dimension. The inputs cannot include both column and row matrices.

   If the value of the **Multiplication** parameter is matrix, the block output is the matrix product of inputs marked * multiplied by the matrix inverse of each input marked /. The order of operations is the same as the order specified by the **Number of Inputs** field, for example, a value of */* results in the matrix product $AB^{-1}C$, where A, B, C are the first, second, and third

inputs signals, respectively. The dimensions of the matrices must be such that the matrix product is defined.

If all inputs are scalars, the output of the block is a scalar. Otherwise, the output is a matrix or a vector depending on whether the inputs are matrices or vectors.

- If the value of the **Number of inputs** parameter is *, the value of the **Multiplication** parameter is element-wise, and the input is vector-like, i.e., a 1-D array or a one-column or one-row 2-D array, the block outputs the scalar product of the elements of the input.

$$y = \Pi u_i$$

In this case, the block icon appears as follows.

If the input is a matrix and the the **Multiplication** parameter is element-wise, Simulink signals an error. If the value of the **Multiplication** parameter is matrix, the block outputs the input unchanged.

- If the value of the **Number of inputs** parameter is /, the value of the **Multiplication** parameter is element-wise, and the input is vector-like, the block outputs the inverse of the scalar product of the input elements. If the input is a matrix and the the **Multiplication** parameter is element-wise, Simulink signals an error. If the value of the **Multiplication** parameter is matrix, the block outputs the matrix inverse of the input.

- Entering a scalar value as the **Number of inputs** parameter is equivalent to entering a string of * characters where the length of the string is the scalar value.

- If the block has a single input, it must be a scalar or vector-like.

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 for element-wise multiplication. All input signals must be of the same data type. The output signal data type is the same as the input's. The inputs must be real or complex signals of type single or double for matrix multiplication.

Product

Parameters and Dialog Box
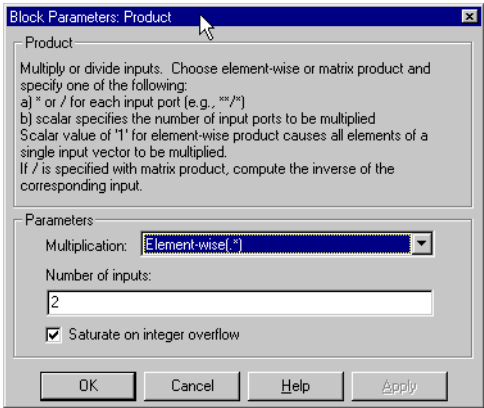
Multiplication

Specifies whether to use element-wise or matrix multiplication to create the product of the inputs.

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

This option is enabled only for element-wise multiplication. 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 option on the **Diagnostics** page of the **Simulation Parameters** dialog (see "The Diagnostics Pane" on page 5-26).

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	No

Purpose   Probe a line for its width, dimensionality, 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, dimensionality, 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 dimensionality, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port. The block accepts real or complex-valued signals of any built-in data type. It outputs signals of type doubl e. During simulation, the block's icon displays the probed data.

>:0, Ts:[0 0], C:
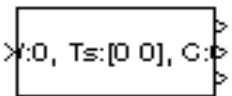
Data Type Support   A Probe block accepts and outputs any built-in data type.

Parameters and Dialog Box

Block Parameters: Probe

┌ Probe ──────────────────────────────────────
Probe a line for its width, sample time, or complex signal flag.

┌ Parameters ─────────────────────────────────
☑ Probe width

☑ Probe sample time

☑ Probe complex signal

☑ Probe signal dimensions

[OK]   [Cancel]   [Help]   [Apply]
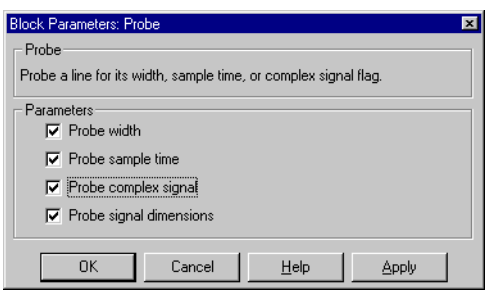
Probe width
    If checked, output width (number of elements) of probed signal.

Probe sample time
    If checked, output sample time of probed signal.

Probe complex signal
    If checked, output 1 if probed signal is complex; otherwise, 0.

Probe signal dimensions
    If checked, output the dimensions of the probed signal.

Probe

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	No

Purpose   Generate pulses at regular intervals.

Library   Sources

Description   The Pulse Generator block generates a series of scalar, vector, or matrix pulses
at regular intervals. The block's **Amplitude**, **Period**, **Duty cycle**, and **Start
time** parameters determines the characteristics of the output signal. All must
have the same dimensions after scalar expansion and must be of the same data
and numeric (complex or real) type.

Use the Pulse Generator block for continuous systems. To generate discrete
signals, use the Discrete Pulse Generator block.

**Data Type
Support**   A Pulse Generator block outputs real or complex signals of any data type. The
data and numeric (real or complex) type of the output signal is the same as that
of the **Amplitude** parameter.

**Parameters
and Dialog Box**
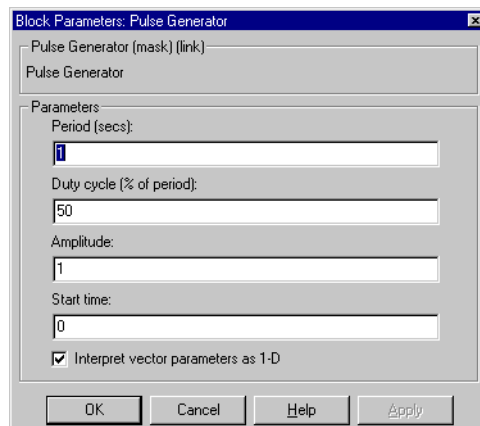
Period

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.

Pulse Generator

Amplitude

The pulse amplitude. The default is 1.

Start time

The delay before the pulse is generated, in seconds. The default is 0 seconds.

Interpret vector parameters as 1-D

If this option is checked and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters.

Characteristics		
	Sample Time	Inherited
	Scalar Expansion	Of parameters
	Dimensionalized	Yes
	Zero Crossing	No

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
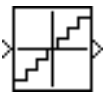
$$y = q * round(u/q)$$

where y is the output, u the input, and q the **Quantization interval** parameter.

Data Type Support

A Quantizer block accepts and outputs real or complex signals of type single or double.

Parameters and Dialog Box

Block Parameters: Quantizer

─ Quantizer ─
Discretize input at given interval.

─ Parameters ─
Quantization interval:

0.5

☑ Treat as gain when linearizing

| OK | Cancel | Help | Apply |
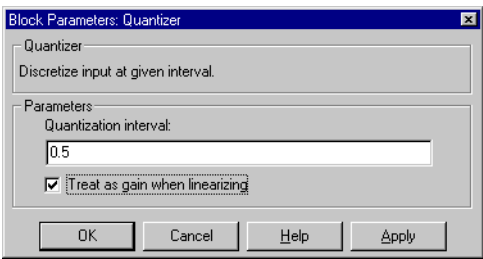
Quantization interval

The interval around which the output is quantized. Permissible output values for the Quantizer block are n*q, where n is an integer and q the **Quantization interval**. The default is 0.5.

Treat as gain when linearizing

Simulink by default treats the Quantizer block as unity gain when linearizing. This is the large signal linearization case. If you uncheck this box, the linearization routines assume the small signal case and set the gain to zero.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block

Quantizer

Scalar Expansion	Of parameter
Dimensionalized	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. The block's **Slope**, **Start time**, **Duty Cycle**, and **Initial output** parameters determines the characteristics of the output signal. All must have the same dimensions after scalar expansion.

Data Type Support   A Ramp block outputs signals of type doubl e.

Parameters and Dialog Box
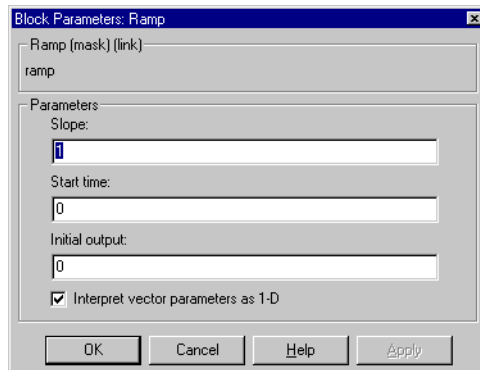
Slope
The rate of change of the generated signal. The default is 1.

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.

Interpret vector parameters as 1-D
If this option is checked and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters.

Ramp

Characteristics

Sample Time	Inherited from driven block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	Yes

9-188

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.

Avoid integrating a random signal because solvers are meant to integrate
relatively smooth signals. Instead, use the Band-Limited White Noise block.

All the blocks numeric parameters must be of the same dimension after scalar
expansion.

Data Type Support   A Random Number block accepts and outputs signals of type double.

Parameters and Dialog Box
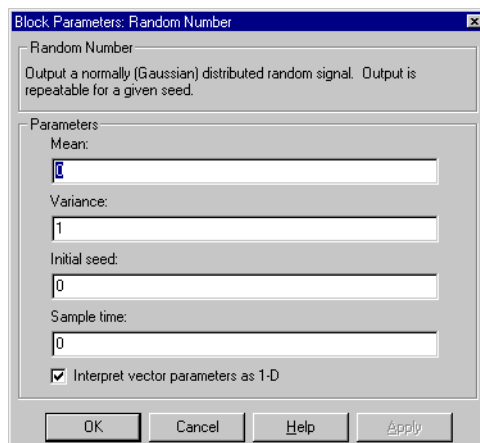
Random Number

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.

Interpret vector parameters as 1-D

If this option is checked and the other parameters are one-row or one-column matrices, after scalar expansion, the block outputs a 1-D signal (vector). Otherwise the output dimensionality is the same as that of the other parameters.

Characteristics

Sample Time	Continuous or discrete
Scalar Expansion	Of parameters
Dimensionalized	Yes
Zero Crossing	No

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.
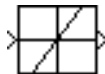
$$rate = \frac{u(i) - y(i-1)}{t(i) - t(i-1)}$$

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   A Rate Limiter block accepts and outputs signals of type doubl e.
Support**

**Parameters
and Dialog Box**
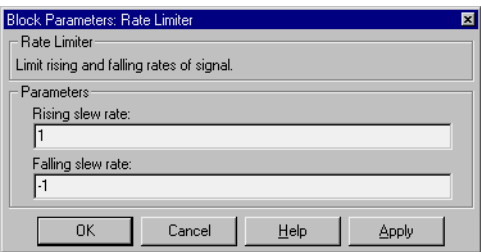
Rate Limiter

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	Continuous
Scalar Expansion	Of input and parameters
Dimensionalized	Yes
Zero Crossing	No

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.
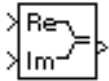
The inputs may be both arrays (vectors or matrices) of equal dimensions, or one input may be an array and the other a scalar. If the block has an array input, the output is a complex array of the same dimensions. The elements of the real input are mapped to real parts of the corresponding complex output elements. The imaginary input 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.

The input signals and real or imaginary output parameter can be of any data type. The output is of the same type as the input or parameter that determines the output.

Data Type Support   See description above.

Parameters and Dialog Box
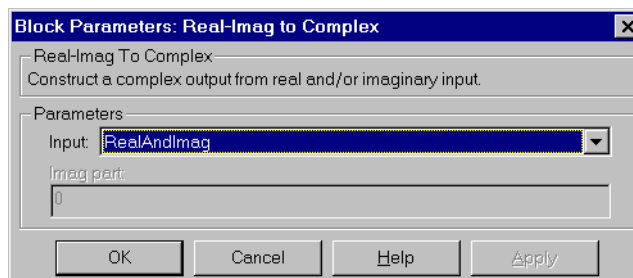
Input

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.

Real-Imag to Complex

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs
Dimensionalized	Yes
Zero Crossing	No

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, arrays, or a combination of a scalar and an array:

- For scalar inputs, the output is a scalar.
- For array inputs, the output is an array of the same dimensions, where each element is the result of an element-by-element comparison of the input arrays.
- For mixed scalar/array inputs, the output is an array, where each element is the result of a comparison between the scalar and the corresponding array element.

The block icon displays the selected operator.

Data and Numeric Type Support A Relational Operator block accepts real or complex signals of any data type. Both inputs must be of the same data type. One input may be real and the other complex, if the operator is == or ! =. The block outputs a signal of type bool ean,

Relational Operator

if Boolean logic signals are enabled (see "Enabling Strict Boolean Type Checking" on page 4-48). Otherwise, the block outputs a signal of type doubl e.

Parameters and Dialog Box
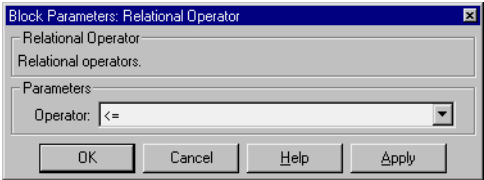
Operator

The relational operator to be applied to the block inputs.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of inputs
Dimensionalized	Yes
Zero Crossing	Yes, to detect when the output changes

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.

The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** value models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

Data Type Support

A Relay block accepts and outputs real signals of type double.

Parameters and Dialog Box
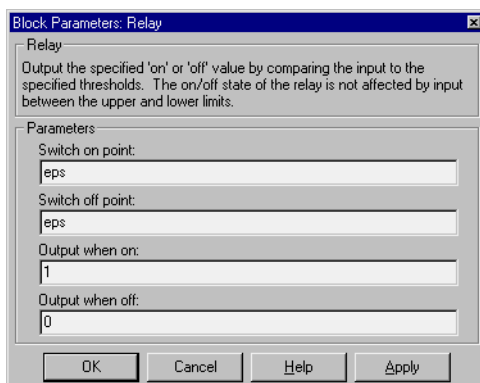
Switch on point

> 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.

Relay

Output when off

The output when the relay is off. The default is 0.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Yes
Dimensionalized	Yes
Zero Crossing	Yes, to detect switch on and switch off points

Purpose   Generate an arbitrarily shaped periodic signal.

Library   Sources

Description   The Repeating Sequence block outputs a periodic scalar signal having a
                     waveform that you specify. You can specify any waveform, using the block
                     dialog's **Time values** and **Output values** parameters. The **Times value**
                     parameter specifies a vector of sample times. The **Output values** parameter
                     specifies a vector of signal amplitudes at the corresponding sample times.
                     Together, the two parameters specify a sampling of the output waveform at
                     points measured from the beginning of the interval over which the waveform
                     repeats (i.e., the signal's period). For example, by default, the **Time values** and
                     Output values parameters are both set to [0 2]. This default setting specifies
                     a sawtooth waveform that repeats every 2 seconds from the start of the
                     simulation and has a maximum amplitude of 2. The Repeating Sequence block
                     uses linear interpolation to compute the value of the waveform between the
                     specified sample points.

Data Type   A Repeating Sequence block outputs real signals of type doubl e.
Support

**Parameters
and Dialog Box**
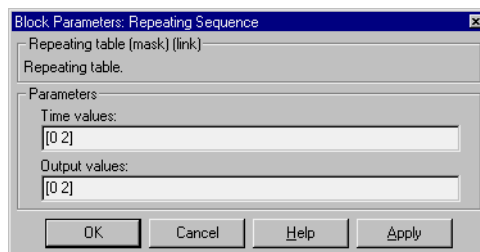
Time values

    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].

Repeating Sequence

Characteristics   Sample Time   Continuous

Scalar Expansion   No

Dimensionalized   No

Zero Crossing   No

Purpose   Change the dimensionality of a signal.

Library   Signals & Systems

Description   The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the block's **Output dimensionality** parameter. For example, you can use the block to change an N-element vector to a 1-by-N or N-by-1 matrix signal, and vice versa.
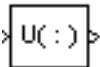
The **Output dimensionality** parameter lets you select any of the following output options.

Output Dimensionality	Description
1-D array	Converts a matrix (2-D array) to a vector (1-D array) array signal. The output vector consists of the first column of the input matrix followed by the second column, etc. (This option leaves a vector input unchanged.)
Column vector	Converts a vector or matrix input signal to a column matrix, i.e., an M-by-1 matrix, where M is the number of elements in the input signal. For matrices, the conversion is done in column-major order.

Reshape

Output Dimensionality	Description
Row vector	Converts a vector or matrix input signal to a row matrix, i.e., a 1-by-N matrix where N is the number of elements in the input signal. For matrices, the conversion is done in column-major order.
Customize	Converts the input signal to an output signal whose dimensions you specify, using the **Output dimensions** parameter. The value of the **Output dimensions** parameter can be a one- or two-element vector. A value of [N] outputs a vector of size N. A value of [M N] outputs an M-by-N matrix. The number of elements of the input signal must match the number of elements specified by the **Output dimensions** parameter. For matrices, the conversion is done in column-major order.

Data Type Support

The Reshape block accepts and outputs signals of any type.

Parameters and Dialog Box
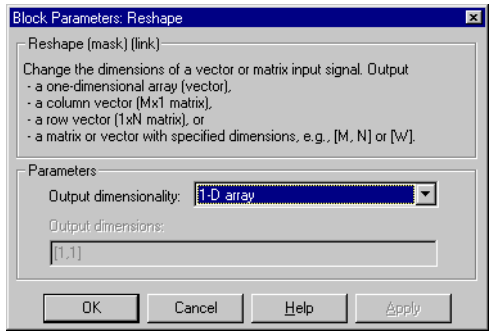
Output dimensionality

The dimensionality of the output signal.

Output dimensions

Specifies a custom output dimensionality. This option is enabled only if you select Customize as the value of the **Output dimensionality** parameter.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Zero Crossing	No

Rounding Function

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 Dimensionalized 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
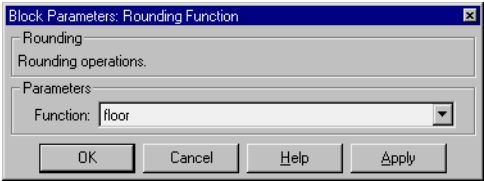
Function
    The rounding function.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	Yes
Zero Crossing	No

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.
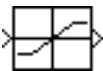
When the parameters are set to the same value, the block outputs that value.

Data Type Support   A Saturation block accepts and outputs real signals of any data type.

Parameters and Dialog Box
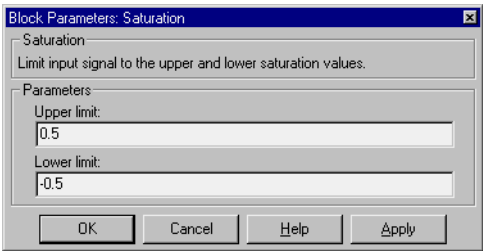
Upper limit

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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of parameters and input
Dimensionalized	Yes
Zero Crossing	Yes, to detect when the signal reaches a limit, and when it leaves the limit

Scope

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 stairstep 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.
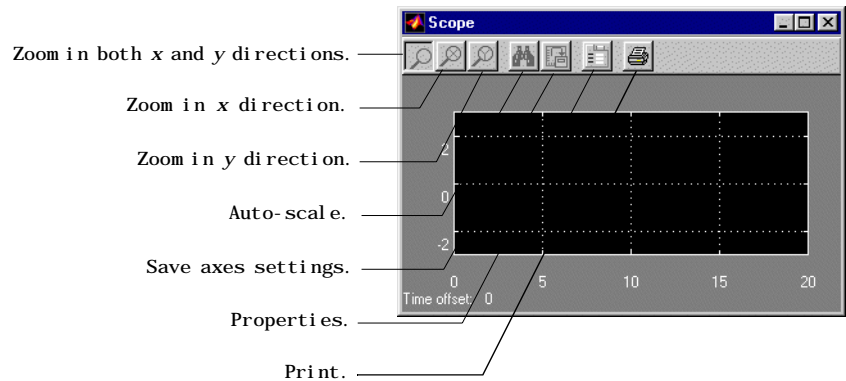
Note Do not use Scope blocks inside of library blocks that you create. Instead, provide the library blocks with output ports to which scopes can be connected to display internal data.

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.
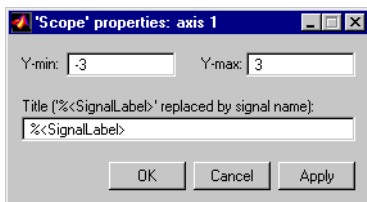
Y-min

> 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 %<Si gnal Label > as part of the title string (%<Si gnal Label > is replaced by the signal label).

Scope

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.
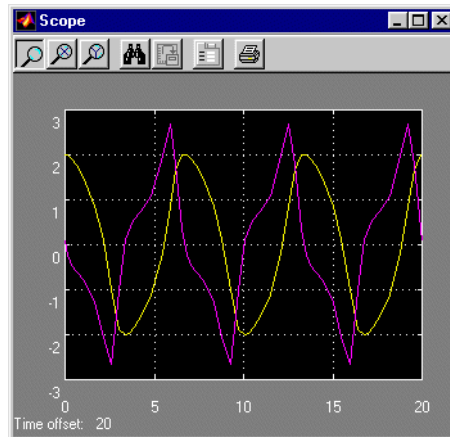
Auto-Scaling the Scope Axes

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.
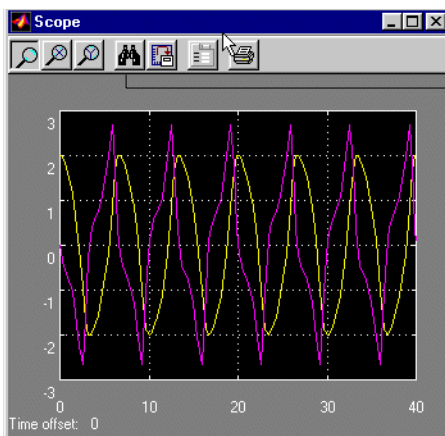
The Auto-scale button

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.
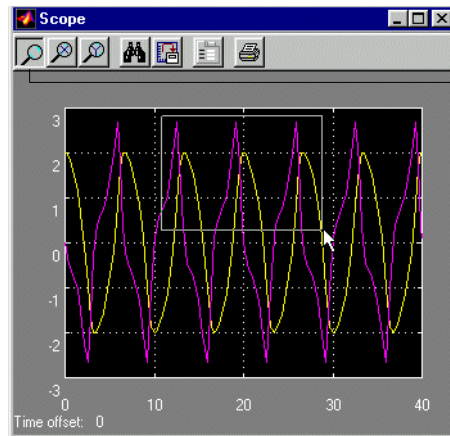
Zoom in both directions

This figure shows the zoomed region, which appears after you release the mouse button.
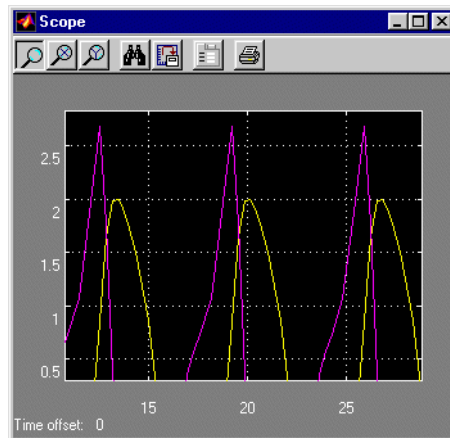
To zoom in on data in just the *x* direction, click on the middle **Zoom** toolbar button. Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the mouse button, then moving the pointer

to the other end of the region. This figure shows the Scope after defining the zoom region but before releasing the mouse button.
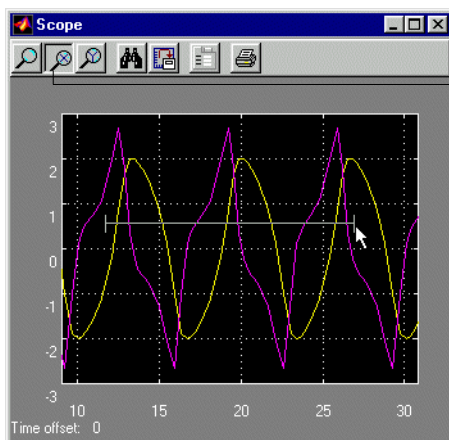
Zoom in x direction

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

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.

 ———— Properties button

When you click on the **Properties** button, this dialog box appears.
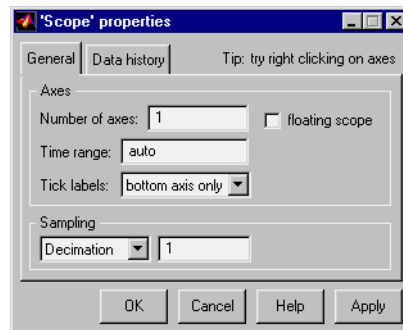
The dialog box has two tabs: **General** and **Data history**.

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. Or you can use the floating Scope's Signal Selector (see "Signal Selector" on page 9-215) to select signals for display. The Signal Selector allows you to select signals anywhere in your model, including unopened subsystems.

You can have more than one floating scope in a model, but only one axes set in one scope can be active at a given time. Active floating scopes show the active axes by making them blue. Selecting or deselecting lines will affect that Scope block only. Other floating Scopes will continue to display the signals that you selected when they were active. In other words, nonactive floating scopes are locked in that their signal displays cannot change.

If you plan to use a floating scope during a simulation, you should disable signal storage reuse. See "Signal storage reuse" on page 5-31 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.

Scope

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.
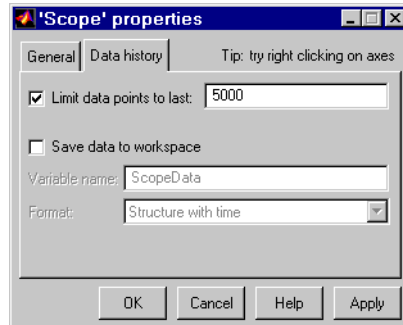
You can also choose to save data to the workspace in this tab. You apply the current parameters and options by clicking on the **Apply** or **OK** button. The values that appear in these fields are the values that will be used in the next simulation.

Limit data points to last

You can limit the number of data points saved to the workspace by checking the **Limit data points 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 data points is limited to 1,000 and the simulation generates 2,000 data points, only the last 1,000 are available for regenerating the display.

Save 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 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: `Array`, `Structure`, or `Structure with time`. Use `Array` 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.

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.

Print icon

Signal Selector

The Signal Selector allows you to select the signals to be displayed in the floating scope. You can use it to select any signal in you model, including signals in unopened subsystems. To display the Signal Selector, first start simulation of your model with the floating scope open. Then right click your mouse in the floating scope and select **Signal Selection** from the popup menu that appears. The Signal Selector appears.
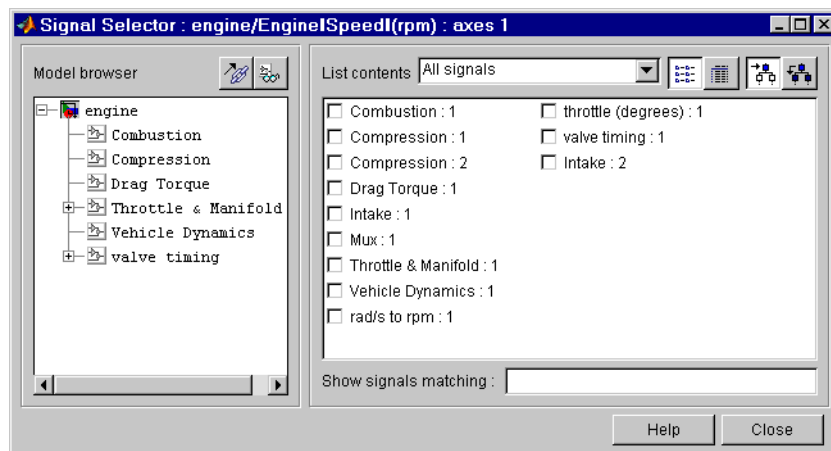
Scope

The Signal Selector contains contains two panes. The left pane allows you to display signals of any subsystem in your model. The signals appear in the right pane. The right pane allows you to select which signals to display in the floating scope.

To select a subsystem for viewing, click its entry in the **Model hierarchy** tree or use the up or down arrows on move the selection highlight to the entry, using the up and down arrows on your keyboard. To show or hide the subsystems contained by the currently selected subsystem, click the +/- button next to the subsystem's name or press the forward or backward arrow keys on your keyboard. To view subsystems included as library links in your model, click the **Library Links** button at the top of the **Model hierarchy** pane. To view the subsystems contained by masked subsystems, click the **Look Under Masks** button at the top of the pane.

The **Signals** pane shows all the signals in the currently selected subsystem by default. To show named signals only, select Named signals only from the **List contents** control at the top of the pane. To show test point signals only, select Test point signals only from the **List contents** control. To show only signals whose signals match a specified string of characters, enter the characters in thes **Show signals matching** control at the bottom of the **Signals** pane and press the **Enter** key.To show the selected types of signals for all subsystems below the currently selected subsystem in the model hierarchy, select the **Current and Below** button at the top of the **Signals** pane.

The **Signals** pane by default shows the name of each signal and the number of the port that emits the signal. To display more information on each signal, select the **Table view** button at the top of the pane. The table view shows the path and data type of each signal and whether the signal is a test point. To select or deselect a signal in the **Signals** pane, click its entry or use the arrow keys to move the selection highlight to the signal entry and press the **Enter** key. You can also move the selection highlight to a signal entry by typing the first few characters of its name (enough to uniquely identify it).

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 from a vector or matrix signal.

Library   Signals & Systems

Description   The Selector block generates as output selected elements of an input vector or matrix.
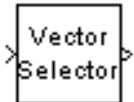
A Selector block accepts either vector or matrix signals as input. Set the **Input Type** parameter to the type of signal (vector or matrix) that the block should accept in your model. The parameter dialog box and the block icon change to reflect the type of input that you select. The way the block determines the elements to select differs slightly, depending on the type of input.

Vector Input

If the input type is vector, a Selector block outputs a vector of selected elements. The block determines the indices of the elements to select either from the block's **Elements** parameter or from an external signal. Set the **Source of element indices** parameter to the source (internal, i.e., parameter value, or external) that you prefer. If you select external, the block adds an input port for the external index signal.

In either case, the elements to be selected 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].
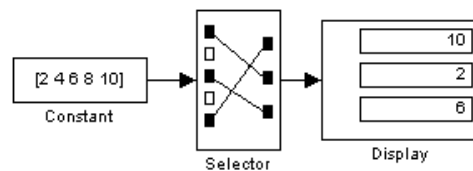
The block icon displays the ordering of input vector elements graphically if the block icon is large enough.

If you select external as the source for element indices, the block adds an input port for the element indices signal. The signal should specify the elements to be selected in the same way they are specified, using the **Elements** parameter.

If the input type is vector, you must specify the width of the input signal or -1, using the **Input port width** parameter. If you specify a width greater than 0,

the width of the input signal must equal the specified width. Otherwise, the block reports an error. If you specify a width of -1, the block accepts a vector signal of any width.

Matrix Input

If the input type is matrix, the Selector block outputs a matrix of elements selected from the input matrix. The block determines the row and column indices of the elements to select either from its **Rows** and **Columns** parameters or from external signals. Set the block's **Source of row indices** and **Source of column indices** to the source that you prefer (internal or external). If you set either source to external, the block adds an input port for the external indices signal. If you set both sources to external, the block adds two input ports.

In either case, the indices of the row and columns to be selected must be specified as vectors (or a scalar if only one row or colum is to be selected). For example, the **Rows** expression [2 1] and the **Columns** expression [1 3] specifies output of a 2x2 matrix whose first row contains the first and third elements of the input matrix's second row and whose second row contains the first and third elements of the input matrix's first row.
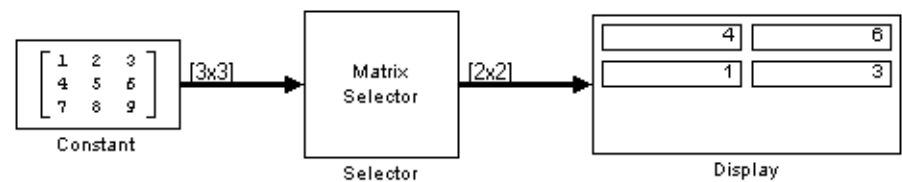
Data Type Support

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.

Parameters and Dialog Box

The parameter dialog box appears as follows when vector input mode is selected.
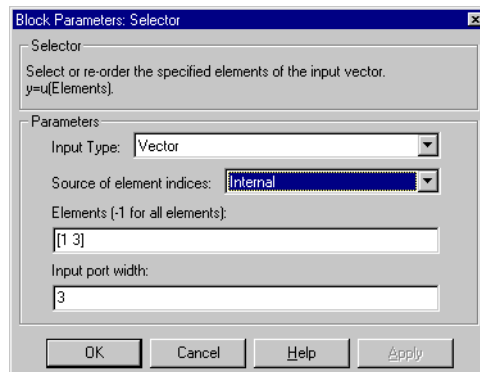
Input Type

The type of the input signal: vector or matrix.

Source of element indices

The source of the indices specifying the elements to select, either internal, i.e., the **Elements** parameter, or external, i.e., an input signal.

Elements

The elements to be included in the output vector.

Input port width

The number of elements in the input vector.

The dialog box appears as follows when matrix input mode is selected.
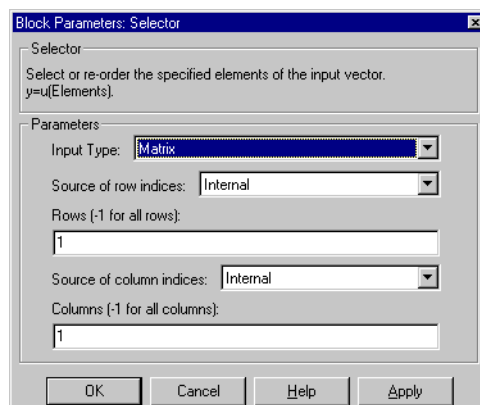
Selector

Input Type

The type of the input signal: vector or matrix.

Source of row indices

The source of the indices specifying the rows to select from the input matrix, either internal, i.e., the **Rows** parameter, or external, i.e., an input signal.

Rows

Indices of the rows from which to select elements to be included in the output matrix.

Source of column indices

The source of the indices specifying the columns to select from the input matrix, either internal, i.e., the **Columns** parameter, or external, i.e., an input signal.

Columns

Indices of the columns from which to select elements to be included in the output matrix.

Characteristics	Sample Time	Inherited from driving block
	Dimensionalized	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.
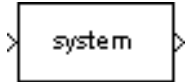
The S-Function block allows additional parameters to be passed directly to the
named S-function. The function parameters can be specified as MATLAB
expressions or as variables separated by commas. For example,

    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**
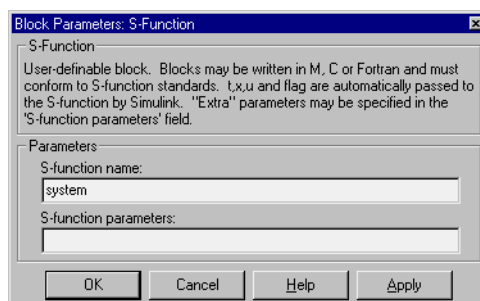
S-Function

S-function name

The S-function name.

S-function parameters

Additional S-function parameters. See the preceeding block description for information on how to specify the parameters.

Characteristics	Direct Feedthrough	Depends on contents of S-function
	Sample Time	Depends on contents of S-function
	Scalar Expansion	Depends on contents of S-function
	Dimensionalized	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:
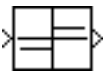
- The output is 1 when the input is greater than zero.
- The output is 0 when the input is equal to zero.
- The output is -1 when the input is less than zero.

Data Type Support A Sign block accepts and outputs real signals of type doubl e.

Dialog Box

```
Block Parameters: Sign                          ☒
┌─ Signum ──────────────────────────────────────
│ Output 1 for positive input, -1 for negative input, and 0 for 0 input.  y =
│ signum(u)
└───────────────────────────────────────────────
        OK          Cancel         Help          Apply
```

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	N/A
Dimensionalized	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.
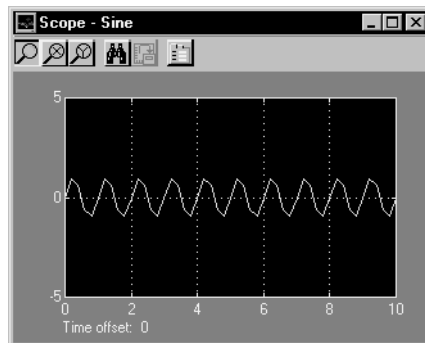
Sine Wave
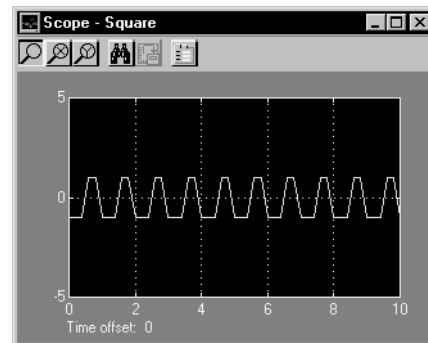
Square Wave
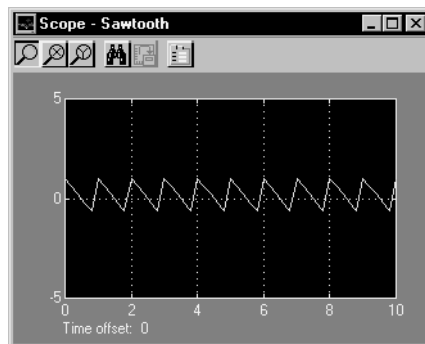
Sawtooth Wave

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.

The block's **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions as the **Amplitude** and **Frequency** parameters (after scalar expansion). If the **Interpret vector parameters as 1-D** option is on, the block outputs a vector (1-D) signal if the **Amplitude** and **Frequency** parameters are row or column vectors, i.e. single row or column 2-D arrays. Otherwise, the block outputs a signal of the same dimensions as the parameters.

Data Type Support

A Signal Generator block outputs a scalar or array of real signals of type doubl e.

Parameters and Dialog Box
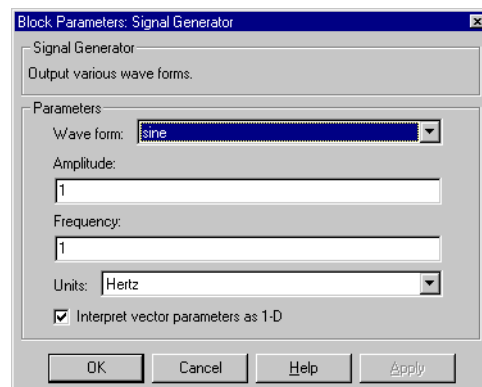
Wave form

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.

Signal Generator

Units

The signal units, hertz or radians/sec. The default is hertz.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the **Amplitude** and **Frequency** parameters result in a vector output signal.

Characteristics	Sample Time	Continuous
	Scalar Expansion	Of parameters
	Dimensionalized	Yes
	Zero Crossing	No

Purpose   Verify that the input signal has specified dimensions, sample time, data type, and numeric type of a signal.

Library   Signals & Systems

Description   This block checks that the input signal has specified attributes. If so, the block outputs the input signal unchanged. Otherwise, it halts the simulation and displays an error message.
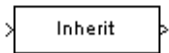
The Signal Specification block can be uses as an assert mechanism to ensure that the attributes of a signal meet the desired attributes for certain sections of your model. For example, consider two people working on different parts of a model, the signal specification block is useful for indicating what attributes various signals are needed by the different sections of the model. If there is a miscommunication and say data types are changed unexpectedly, the attributes will not match up and Simulink will report an appropriate error. Using the signal specification block will help ensure you don't introduce unexpected problems in your models. If you are familiar with the assert mechanism in languages such as C, you will see that the signal specification block servers a similar purpose.

The Signal Specification block can also be used to assure correct propagation of signal attributes throughout a model. Simulink's capability of allowing many attributes to propagate from block to block is very powerful. However, it is possible to create models (when using user written S-functions) that don't have enough information to correctly propagate attributes around the model. For these cases, the signal specification block is a good way of providing the information Simulink needs when propagating attributes from block to block. The use of the signal specification block also helps speed up model compilation (update diagram) when blocks are missing signal attributes.

Data Type Support   Accepts signals of any data type that matches the data type specified by the **Data Type** parameter

Signal Specification

Parameters and Dialog Box
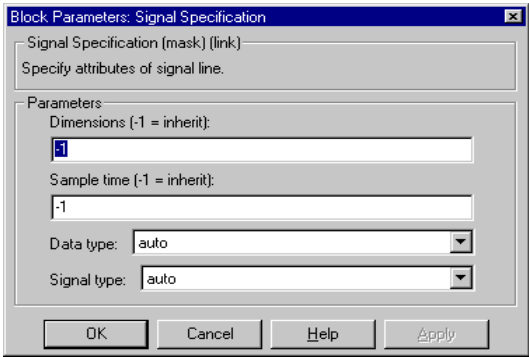
Dimensions

Dimensions that input signal must match. Valid values are -1 (don't care), n (vector signal of width n), [m n] (matrix signal having m rows and n columns.

Sample Time

Sample time that input signal must match. Valid values are -1 (don't care), period >= 0, [offset, period], [0, -1], [-1, -1], where period is the sample rate and offset is the offset of the sample period from time zero (see "Sample Time" on page 3-23).

Data Type

Data type that input signal must match. Choices include auto (don't care), double, single, int8, uint8, int16, uint16, int32, uint32, and boolean.

Signal Type

Numeric type that input signal must match. Choices include auto (don't care), real, or complex.

Characteristics

Direct Feedthrough	Yes
Sample Time	Continuous
Scalar Expansion	No
States	0
Dimensionalized	Yes
Zero Crossing	No

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

$$y = Amplitude \times \sin(frequency \times time + phase)$$

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.

Sine Wave

$$\begin{bmatrix} \sin(t+\Delta t) \\ \cos(t+\Delta t) \end{bmatrix} = \begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \end{bmatrix}$$

Since Δt is constant, the following expression is a constant.

$$\begin{bmatrix} \cos(\Delta t) & \sin(\Delta t) \\ -\sin(\Delta t) & \cos(\Delta t) \end{bmatrix}$$

Therefore the problem becomes one of a matrix multiply of the value of $\sin(t)$ by a constant matrix to obtain $\sin(t+\Delta t)$.

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.

The block's numeric parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions and dimensionlity as the parameters. If the **Interpret vector parameters as 1-D** option is on and the numeric parameters are row or column vectors (i.e., single row or column 2-D arrays), the block outputs a vector (1-D array) signal; otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support   A Sine Wave block accepts and outputs real signals of type doubl e.

Parameters and Dialog Box
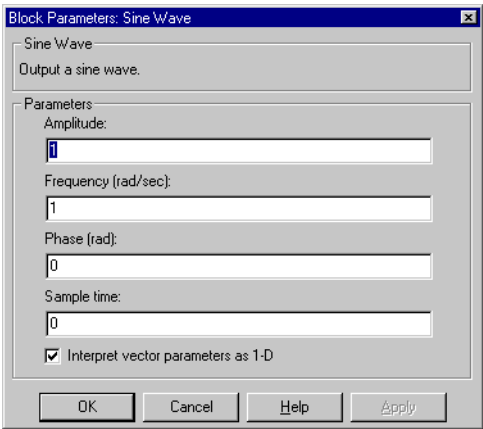
Amplitude

The amplitude of the signal. The default is 1.

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.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the Sine Wave block's numeric parameters result in a vector output signal; otherwise, the block outputs a signal of the same dimensionality as the parameters. If this option is not selected, the block always outputs a signal of the same dimensionality as the block's numeric parameters.

Characteristics		
Sample Time	Continuous, discrete, or inherited	
Scalar Expansion	Of parameters	
Dimensionalized	Yes	
Zero Crossing	No	

Slider Gain

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.
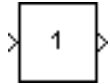
Data Type Support
Data type support for the Slider Gain block is the same as that for the Gain block (see "Gain" on page 9-108).

Dialog Box
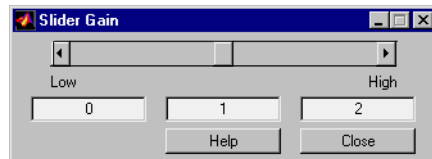
Low
> The lower limit of the slider range. The default is 0.

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 or matrix gain to the block input, consider using the Gain block.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the gain
States	0
Dimensionalized	Yes
Zero Crossing	No

State-Space

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.
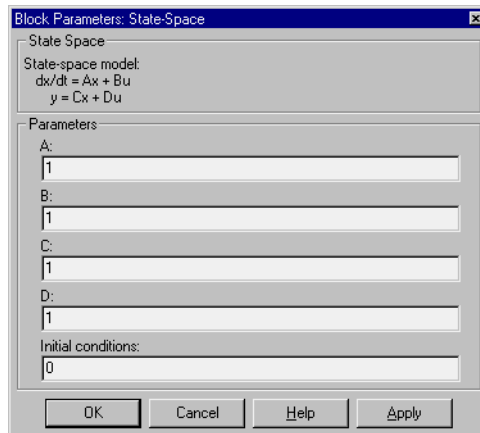
Data Type Support   A State-Space block accepts and outputs real signals of type doubl e.

Parameters and Dialog Box

```
Block Parameters: State-Space              ☒
─ State Space ─────────────────────────────
 State-space model:
   dx/dt = Ax + Bu
     y = Cx + Du
─ Parameters ──────────────────────────────
 A:
 ┌───────────────────────────────────────┐
 │1                                       │
 └───────────────────────────────────────┘
 B:
 ┌───────────────────────────────────────┐
 │1                                       │
 └───────────────────────────────────────┘
 C:
 ┌───────────────────────────────────────┐
 │1                                       │
 └───────────────────────────────────────┘
 D:
 ┌───────────────────────────────────────┐
 │1                                       │
 └───────────────────────────────────────┘
 Initial conditions:
 ┌───────────────────────────────────────┐
 │0                                       │
 └───────────────────────────────────────┘

  ┌────────┐ ┌────────┐ ┌────────┐ ┌────────┐
  │   OK   │ │ Cancel │ │  Help  │ │ Apply  │
  └────────┘ └────────┘ └────────┘ └────────┘
```

A, B, C, D

The matrix coefficients.

Initial conditions

The initial state vector.

Characteristics

Direct Feedthrough	Only if $D \neq 0$
Sample Time	Continuous
Scalar Expansion	Of the initial conditions
States	Depends on the size of A
Dimensionalized	Yes
Zero Crossing	No

Step

Purpose	Generate a step function.
Library	Sources
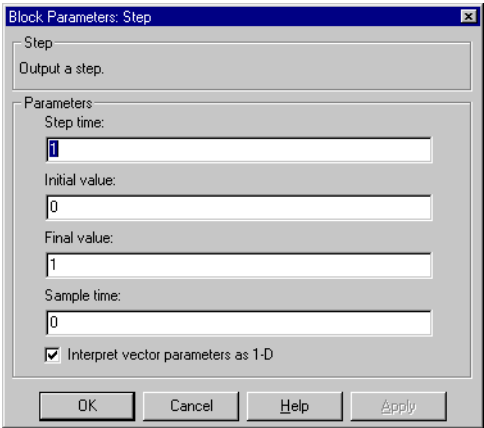
Description

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 block's numeric parameters must be of the same dimensions after scalar expansion. If the **Interpret vector parameters as 1-D** option is off, the block outputs a signal of the same dimensions and dimensionlity as the parameters. If the **Interpret vector parameters as 1-D** option is on and the numeric parameters are row or column vectors (i.e., single row or column 2-D arrays), the block outputs a vector (1-D array) signal; otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Data Type Support

A Step block outputs real signals of type doubl e.

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.

Sample time

Sample rate of step.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the Step block's numeric parameters result in a vector output signal; otherwise, the block outputs a signal of the same dimensionality as the parameters. If this option is not selected, the block always outputs a signal of the same dimensionality as the block's numeric parameters.

Characteristics		
	Sample Time	Inherited from driven block
	Scalar Expansion	Of parameters
	Dimensionalized	Yes
	Zero Crossing	Yes, to detect step times

Stop Simulation

Purpose   Stop the simulation when the input is nonzero.

Library   Sinks

Description   The Stop Simulation block stops the simulation when the input is nonzero.
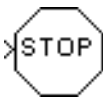
The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

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.
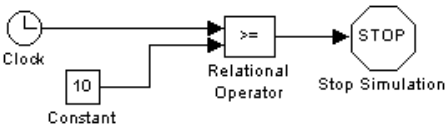
Data Type Support   A Stop Simulation block accepts real signals of type doubl e or bool ean.

Dialog Box
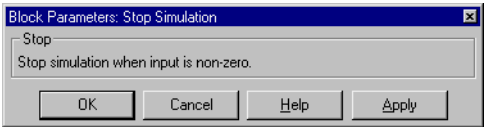
Characteristics   Sample Time   Inherited from driving block

                       Dimensionalized   Yes

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 Signals & Systems 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.

For more information about subsystems, see "Creating Subsystems" in
Chapter 3.

**Data Type
Support**   A subsystem's enable and trigger ports accept any data type. See "Inport" on
                     page 9-119 for information on the data types accepted by a subsystem's input
                     ports. See "Outport" on page 9-169 for information on the data types ouput by
                     a subsystem's output ports.

Subsystem

Parameters and Dialog Box
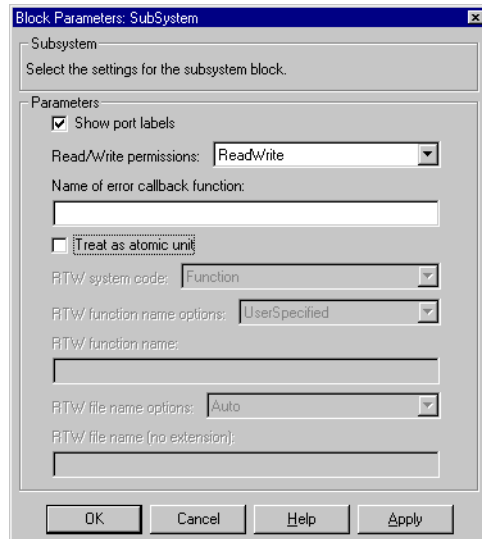
Show port labels

Causes Simulink to display the labels of the subsystem's ports in the subsystem's icon.

Treat as atomic unit

Causes Simulink to treat the subsystem as a unit when determining block execution order. When it comes time to execute the subsystem, Simulink executes all blocks within the subsystem before executing any other block at the same level as the subsystem block. If this option is not selected, Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block execution order. This can cause execution of blocks within the subsystem to be interleaved with execution of blocks outside the subsystem. See "Atomic Versus Virtual Subsystems" on page 3-13 for more information.

Access

Controls user access to the contents of the subsystem. You can select any of the following values.

Access	Description
ReadWrite	User can open and modify the contents of the subsystem.
ReadOnly	User can open but not modify the subsystem. If the subsystem resides in a block library, a user can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.
NoReadOrWrite	User cannot open or modify the subsystem. If the subsystem resides in a library, a user can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.
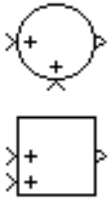
Name of error callback function

Name of a function to be called if an error occurs while executing the subsystem. Simulink passes two arguments to the function: the handle of the subsystem and a string that specifies the error type. If no function is specified, Simulink displays a generic error message if executing the subsystem causes an error.

Note Parameters whose names begin with RTW are used by the Real-Time Workshop for code generation. See the Real-Time Workshop documentation for more information.

Subsystem

Characteristics   Sample Time   Depends on the blocks in the subsystem

Dimensionalized   Depends on the blocks in the subsystem

Zero Crossing   Yes, for enable and trigger ports if present

Purpose Output the sum of inputs.

Library Math

Description The Sum block adds scalar, vector, or matrix inputs or the elements of a single vector input. The following rules determine the block's output:

- If the block has more than one input, all nonscalar inputs must be of the same dimensionality and dimensions, that is, either all vectors or all matrices of the same dimensions. For example, if any input is a 2-by-2 matrix, any other input must be a 2-by-2 matrix or a scalar.

- If any input is a scalar, it is expanded to have the same dimensions as the nonscalar inputs. For example, if the nonscalar inputs are 2-by-2 matrices, the scalar inputs are expanded to be 2-by-2 matrices.

- The output has the same dimensions as the inputs (after scalar expansion) and each element is the sum of the corresponding elements of the inputs. In other words, the output is the element-wise sum of the inputs.

- If the block has only one input, it must be either a scalar or a vector. If the input is a vector, the output is a scalar equal to the sum of the elements of the input 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 Pane" on page 5-26).

Characteristics		
	Direct Feedthrough	Yes
	Sample Time	Inherited from driving blocks
	Scalar Expansion	Yes
	States	0
	Dimensionalized	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.
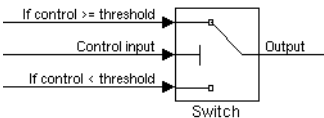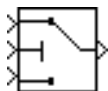
                   To drive the switch with a logic input (i.e., 0 or 1), set the threshold to 0.5.

Data Type   A Switch block accepts real- or complex-valued signals of any data type as
Support   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 threshhold input must be bool ean or doubl e.

Parameters
and Dialog Box
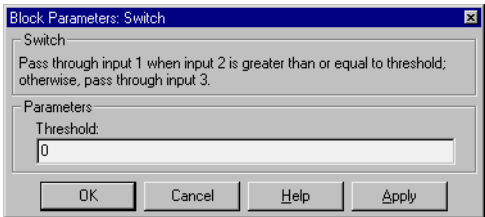
                   Threshold
                        The value of the control (the second input) at which the switch flips to its
                        other state. You can specify this parameter as either a scalar or a vector
                        equal in width to the input vectors.

Characteristics   Direct Feedthrough   Yes

                      Sample Time   Inherited from driving block

                      Scalar Expansion   Yes

Dimensionalized   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
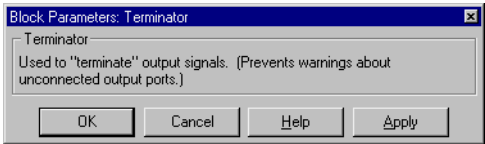
Block Parameters: Terminator

┌ Terminator ─────────────────────────────
Used to "terminate" output signals. (Prevents warnings about unconnected output ports.)

| OK | Cancel | Help | Apply |

Characteristics

Sample Time	Inherited from driving block
Dimensionalized	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.

>untitled.mat

$$\begin{bmatrix} t_1 & t_2 & \ldots t_{final} \\ u1_1 & u1_2 & \ldots u1_{final} \\ .. & & \\ un_1 & un_2 & \ldots un_{final} \end{bmatrix}$$

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 nth 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
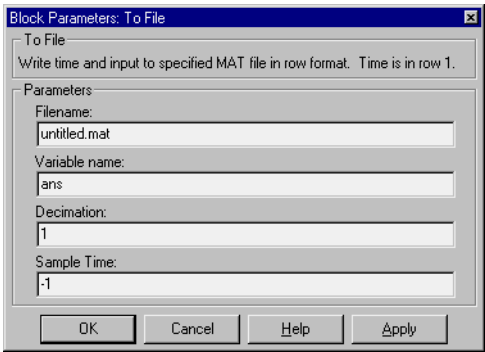
Filename

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
	Dimensionalized	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 an array or structure that has the name specified by the block's **Variable name** parameter. The **Save format** parameter determines the output format.

simout
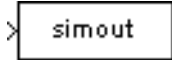
Array

Selecting this option causes the To Workspace block to save the input as an N-dimensional array where N is one more than the number of dimensions of the input signal. For example, if the input signal is a 1-D array (i.e., a vector), the resulting workspace array is two-dimensional. If the input signal is a 2-D array (i.e., a matrix), the array is three-dimensional.

The way samples are stored in the array depends on whether the input signal is a scalar or vector or a matrix. If the input is a scalar or a vector, each input sample is output as a row of the array. For example, suppose that the name of the output array is `simout`. Then, `simout(1,:)` corresponds to the first sample, `simout(2,:)` corresponds to the second sample, etc. If the input signal is a matrix, the third dimension of the workspace array corresponds to the values of the input signal at specified sampling point. For example, suppose again that `simout` is the name of the resulting workspace array. Then, `simout(:,:,1)` is the value of the input signal at the first sample point; `simout(:,:,2)` is the value of the input signal at the second sample point; etc.

The amount of data written and the time steps at which the data is written are determined by block parameters:

- The **Limit data points to last** parameter indicates how many sample points to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. To capture all the data, set this value to `inf`.

- The **Decimation** parameter allows you to write data at every nth 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 array to which the data is written.

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 three fields: values, dimensions, and label. The values field contains the array of signal values. The dimensions field specifies the dimensions of the values array. The label field contains the label of the input line.

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 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, use the Structure with Time format to save the data.

Examples

In a simulation where the start time is 0, the **Maximum number of sample points** 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 sample points** 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 **Limit data points to last** is 3. In this case, only the last three sample points collected are written to the workspace. If the simulation stop time is 100, data corresponds to times 99.0, 99.5, and 100.0 seconds (three points).

Data Type Support

A To Workspace block can save input of any real or complex data type to the MATLAB workspace.

Parameters and Dialog Box
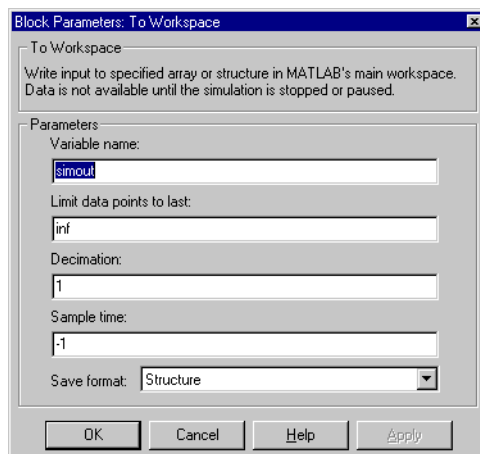
Variable name

The name of the array that holds the data.

Limit data points to last

The maximum number of input samples to be saved. The default is 1000 samples.

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 structure.

To Workspace

Characteristics   Sample Time   Inherited

                                     Dimensionalized   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
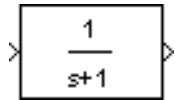
$$H(s) = \frac{y(s)}{u(s)} = \frac{num(s)}{den(s)} = \frac{num(1)s^{nn-1} + num(2)s^{nn-2} + \dots + num(nn)}{den(1)s^{nd-1} + den(2)s^{nd-2} + \dots + den(nd)}$$

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 documentation.

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

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:

$$\frac{num(s)}{den(s)}$$

Data Type Support

A Transfer Fcn block accepts and outputs signals of type double.
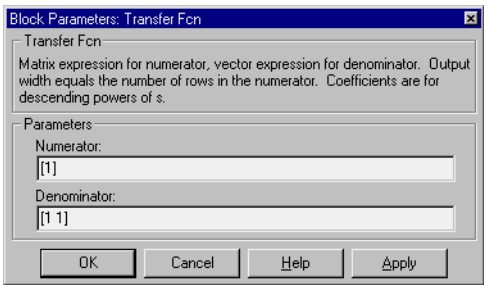
Parameters and Dialog Box

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 **Numerator** and **Denominator** parameters are equal
	Sample Time	Continuous
	Scalar Expansion	No
	States	Length of **Denominator** -1

Dimensionalized	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 dimensionalized 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 - tdelay*.

This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Using l i nmod 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

```
Block Parameters: Transport Delay                              ⊠
┌─ Transport Delay ──────────────────────────────────────────┐
│ Apply specified delay to the input signal.  Best accuracy is achieved
│ when the delay is larger than the simulation step size.     │
│ ┌─ Parameters ──────────────────────────────────────────┐  │
│ │ Time delay:                                            │  │
│ │ [1                                                  ]  │  │
│ │ Initial input:                                         │  │
│ │ [0                                                  ]  │  │
│ │ Initial buffer size:                                   │  │
│ │ [1024                                               ]  │  │
│ │ Pade order (for linearization):                        │  │
│ │ [0                                                  ]  │  │
│ └──────────────────────────────────────────────────────┘  │
│      [  OK  ]   [ Cancel ]   [  Help  ]   [  Apply  ]       │
└────────────────────────────────────────────────────────────┘
```
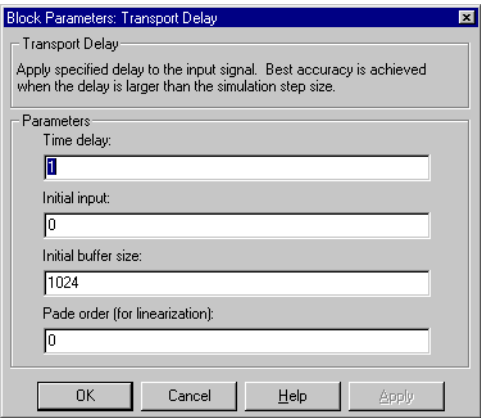
Time delay

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.

Pade order (for linearization)

The order of the Pade approximation for linearization routines. The default value is 0, which results in a unity gain with no dynamic states. Setting the order to a positive integer n adds n states states to your model, but results in a more accurate linear model of the transport delay.

Characteristics

Direct Feedthrough	No
Sample Time	Continuous
Scalar Expansion	Of input and all parameters except **Initial buffer size**

Transport Delay

Dimensionalized	Yes
Zero Crossing	No

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 any data type.

Trigger

Parameters and Dialog Box

Block Parameters: Trigger
┌─ Trigger Port ──┐
Place this block in a subsystem to create a triggered subsystem.
└───┘
┌─ Parameters ──┐
Trigger type: [rising   ▼]
☐ Show output port
Output data type: [auto   ▼]
└───┘

[OK]   [Cancel]   [Help]   [Apply]
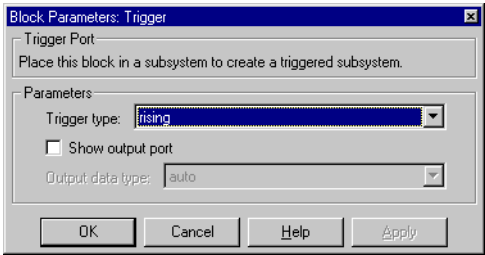
Trigger type

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
	Dimensionalized	Yes

Purpose   Perform a trigonometric function.

Library   Math

Description   The Trigonometric Function block performs numerous common trigonometric functions.
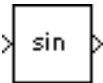
You can select one of these functions from the **Function** list: sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, and tanh. The block output is the result of the function operating on the input or inputs.

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 dimensionalized 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
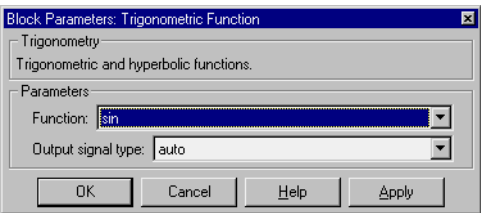
Function
   The trigonometric function.

Output signal type
   Type of signal (complex or real) to output.

Characteristics

Direct Feedthrough	Yes
Sample Time	Inherited from driving block
Scalar Expansion	Of the input when the function requires two inputs

Trigonometric Function

Dimensionalized	Yes
Zero Crossing	No

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 normally distributed random numbers, use
the Random Number block.

Avoid integrating a random signal because solvers are meant to integrate
relatively smooth signals. Instead, use the Band-Limited White Noise block.

The block's numeric parameters must be of the same dimensions after scalar
expansion. If the **Interpret vector parameters as 1-D** option is off, the block
outputs a signal of the same dimensions and dimensionlity as the parameters.
If the **Interpret vector parameters as 1-D** option is on and the numeric
parameters are row or column vectors (i.e., single row or column 2-D arrays),
the block outputs a vector (1-D array) signal; otherwise, the block outputs a
signal of the same dimensions as the parameters.

**Data Type
Support**   A Uniform Random Number block outputs a real signal of type double.

**Parameters
and Dialog Box**

9-265

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.

Interpret vector parameters as 1-D

If selected, column or row matrix values for the Step block's numeric parameters result in a vector output signal; otherwise, the block outputs a signal of the same dimensionality as the parameters. If this option is not selected, the block always outputs a signal of the same dimensionality as the block's numeric parameters.

Characteristics	Sample Time	Continuous, discrete, or inherited
	Scalar Expansion	No
	Dimensionalized	Yes
	Zero Crossing	No

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.
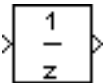
If an undelayed sample-and-hold function is desired, use a Zero-Order Hold block, or if a delay of greater than one unit is desired, use a Discrete Transfer Fcn block. (See the description of the Transport Delay block for an example that uses the Unit Delay block.)

Data Type Support   A Unit block accepts real or complex signals of any 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
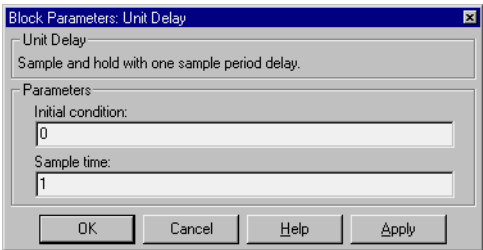
Initial condition

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.

Characteristics

Direct Feedthrough	No
Sample Time	Discrete
Scalar Expansion	Of the **Initial condition** parameter or the input

Unit Delay

States	Inherited from driving block or parameters
Dimensionalized	Yes
Zero Crossing	No

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.
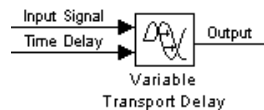
The **Maximum delay** parameter defines the largest value the time delay input
can have. The block clips values of the delay that exceed this value. The
Maximum delay must be greater than or equal to zero. If the time delay
becomes negative, the block clips it to zero and issues a warning message.

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 - tdelay*.

Variable Transport Delay

Data Type Support	A Variable Transport Delay block accepts and outputs real signals of type doubl e.

Parameters and Dialog Box
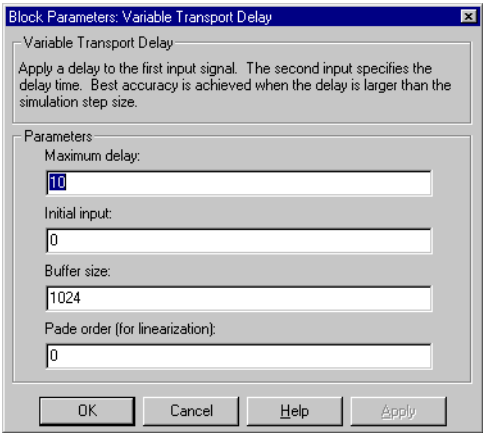
Maximum delay

> 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.

Pade order (for linearization)

The order of the Pade approximation for linearization routines. The default value is 0, which results in a unity gain with no dynamic states. Setting the order to a positive integer n adds n states states to your model, but results in a more accurate linear model of the transport delay.

Characteristics	Direct Feedthrough	Yes, of the time delay (second) input
	Sample Time	Continuous
	Scalar Expansion	Of input and all parameters except **Buffer size**

Dimensionalized   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.
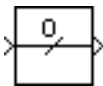
Data Type Support　　The Width block accepts real- or complex-valued signals of any data type, including mixed-type signal vectors. A Width block outputs real signals of type doubl e.

Parameters and Dialog Box
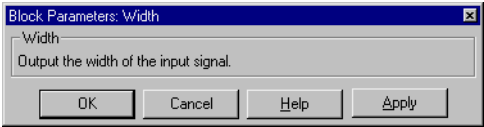
Characteristics

Sample Time	Constant
Dimensionalized	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 lorenzs in the
command window.

**Data Type
Support**   An XY Graph block accepts real signals of type double.

**Parameters
and Dialog Box**
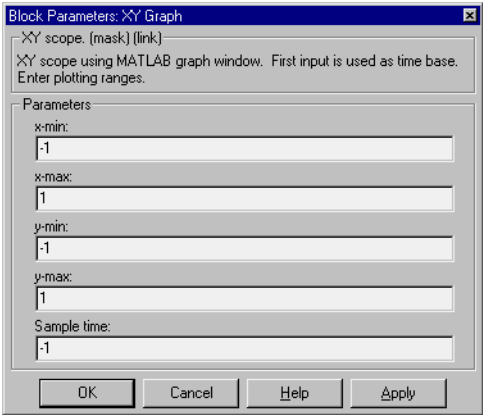
x-min

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..
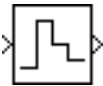
This block provides a mechanism for discretizing one or more signals or resampling the signal at a different rate. You can use it in instances where you need to model sampling without requiring one of the other more complex discrete function blocks. For example, it could be used in conjunction with a 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.

Characteristics

Direct Feedthrough	Yes
Sample Time	Discrete
Scalar Expansion	Yes
States	0
Dimensionalized	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*.
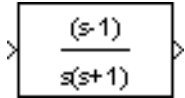
A transfer function can be expressed in factored or zero-pole-gain form, which, for a single-input single-output system in MATLAB, is

$$H(s) = K\frac{Z(s)}{P(x)} = K\frac{(s - Z(1))(s - Z(2))...(s - Z(m))}{(s - P(1))(s - P(2))...(s - P(n))}$$

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:
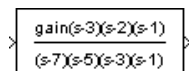
- If each is specified as a variable, the icon shows the variable name followed by "(s)" if appropriate. For example, if you specify **Zeros** as zeros, **Poles** as poles, and **Gain** as gain, the icon looks like this.
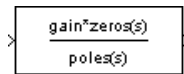
Data Type Support

A Zero-Pole block accepts real signals of type double.

Parameters and Dialog Box
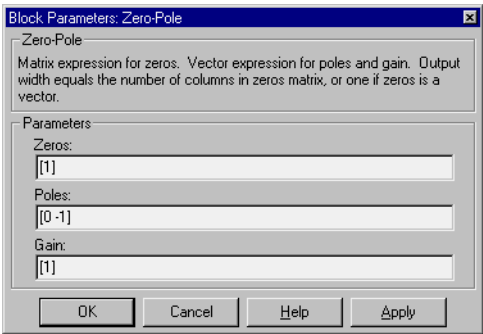
Zeros

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].

Characteristics	Direct Feedthrough	Only if the lengths of the **Poles** and **Zeros** parameters are equal
	Sample Time	Continuous
	Scalar Expansion	No
	States	Length of **Poles** vector

Zero-Pole

Dimensionalized	No
Zero Crossing	No

10

Model Construction Commands

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.

Task	Command
Create a new Simulink system.	new_system
Open an existing system.	open_system
Close a system window.	close_system, bdclose
Save a system.	save_system
Find a system, block, line, or annotation.	find_system
Add a new block to a system.	add_block
Delete a block from a system.	delete_block
Replace a block in a system.	replace_block
Add a line to a system.	add_line
Delete a line from a system.	delete_line
Get a parameter value.	get_param
Set parameter values.	set_param
Get the pathname of the current block.	gcb
Get the pathname of the current system.	gcs
Get the handle of the current block.	gcbh
Get the name of the root-level system.	bdroot
Open the Simulink block library.	simulink

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, "Model and Block Parameters" 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/subsystem$_1$/.../subsystem

- To identify a block, specify the path of the system that contains the block and specify the block name.

    system/subsystem$_1$/.../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

Purpose   Add a line to a Simulink system.

Syntax   h = add_line('sys', 'oport', 'iport')
                  h = add_line('sys', points)

Description   The add_line command adds a line to the specified system and returns a
                  handle to the new line. The line can be defined in two ways:

                  • By naming the block ports that are to be connected by the line
                  • By specifying the location of the points that define the line segments

                  add_line('sys', 'oport', 'iport') adds a straight line to a system from the
                  specified block output port 'oport' to the specified block input port 'iport'.
                  'oport' and 'iport' are strings consisting of a block name and a port
                  identifier in the form 'block/port'. Most block ports are identified by
                  numbering the ports from top to bottom or from left to right, such as 'Gain/1'
                  or 'Sum/2'. Enable, Trigger, and State ports are identified by name, such as
                  'subsystem_name/Enable', 'subsystem_name/Trigger', or
                  'Integrator/State'.

                  add_line(system, points) adds a segmented line to a system. Each row of the
                  points array specifies the *x* and *y* coordinates of a point on a line segment. The
                  origin is the top left corner of the window. The signal flows from the point
                  defined in the first row to the point defined in the last row. If the start of the
                  new line is close to the output of an existing block or line, a connection is made.
                  Likewise, if the end of the line is close to an existing input, a connection is
                  made.

Examples   This command adds a line to the mymodel system connecting the output of the
                  Sine Wave block to the first input of the Mux block.

                      add_line('mymodel','Sine Wave/1','Mux/1')

                  This command adds a line to the mymodel system extending from (20, 55) to
                  (40, 10) to (60, 60).

                      add_line('mymodel',[20 55; 40 10; 60 60])

See Also   delete_line

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.

                   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 saves the mymdl12 system under the new 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, ports, and annotations.
Syntax	find_system(sys, 'c1', cv1, 'c2', cv2,...'p1', v1, 'p2', v2,...)

Description   find_system(sys, 'c1', cv1, 'c2', cv2,...'p1', v1, 'p2', v2,...)
searches the system(s) or subsystems specified by sys, using the constraint(s)
specified by c1, c2, etc., 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 and returns a cell array of pathnames.

Case is ignored for parameter names. Value strings are case sensitive by
default (see the 'CaseSensitive' search constraint for more information). Any
parameters that correspond to dialog box entries have string values. See
Appendix A, "Model and Block Parameters," for a list of model and block
parameters.

You can specify any of the following 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'	'none'	Search skips masked blocks.
	{'graphical'}	Search includes masked blocks that have no workspace and no dialog. This is the default

Name	Value Type	Description
	'functional'	Search includes masked blocks that do not have a dialog.
	'all'	Search includes all masked blocks.
'FollowLinks'	'on' | {'off'}	If 'on', search follows links into library blocks. Default is 'off'.
'FindAll'	'on' | {'off'}	If 'on', search extends to lines, ports, and annotations within systems. Default is 'off'. Note that find_system returns a vector of handles when this option is 'on', regardless of the array type of sys.
'CaseSensitive'	{'on'} | 'off'	If 'on', search considers case when matching search strings. Default is 'on'.
'RegExp'	'on' | {'off'}	If 'on', search treats search expressions as regular expressions. Default is 'off'.

The table encloses default constraint values in brackets. If a 'constraint' is omitted, find_system uses the default constraint value.

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');
```

Searching with Regular Expressions

If you specify the 'RegExp' constraint as 'on', find_system treats search value strings as regular expressions. A regular expression is a string of characters in which some characters have special pattern-matching significance. For example, a period (.) in a regular expression matches not only itself but any other character.

Regular expressions greatly expand the types of searches you can perform with find_system. For example, regular expressions allow you to do partial word searches. You can search for all objects that have a specified parameter that contains or begins or ends with a specified string of characters.

To use regular expressions effectively, you need to learn the meanings of the special characters that regular expressions can contain. The following table lists the special characters supported by find_subsystem and explains their usage.

Expression	Usage
.	Matches any character. For example, the string 'a.' matches 'aa', 'ab', 'ac', etc.
*	Matches zero or more of preceding character. For example, 'ab*' matches 'a', 'ab', 'abb', etc. The expression '.*' matches any string, including the empty string.
+	Matches one or more of preceding character. For example, 'ab+' matches 'ab', 'abb', etc.
^	Matches start of string. For example, '^a.*' matches any string that starts with 'a'.
$	Matches end of string. For example, '.*a$' matches any string that ends with 'a'.
\	Causes the next character to be treated as an ordinary character. This "escape" character lets regular expressions match expressions that contain special characters. For example, the search string '\\' matches any string containing a \ character.
[]	Matches any one of a specified set of characters. For example, 'f[oa]r' matches 'for' and 'far'. Some characters have special meaning within brackets. A hyphen (-) indicates a range of characters to match. For example, '[a-zA-Z1-9]' matches any alphanumeric character. A circumflex (^) indicates characters that should not produce a match. For example, 'f[^i]r' matches 'far' and 'for' but not 'fir'.
\w	Matches a word character. (This is a shorthand expression for [a-z_A-Z0-9].) For example, '^\w' matches 'mu' but not '&mu'.
\d	Matches any digit (shorthand for [0-9]). For example, '\d+' matches any integer.

find_system

Expression	Usage
\D	Matches any non-digit (shorthand for [^0-9]).
\s	Matches a white space (shorthand for [\t\r\n\f]).
\S	Matches a non-white-space (shorthand for[^ \t\r\n\f]).
\<WORD\>	Matches WORD exactly, where WORD is a string of characters separated by white space from other words. For example, '\<to\>' matches 'to' but not 'today'.

To use regular expressions to search Simulink systems, specify the 'regexp' search constraint as 'on' in a find_system command and use a regular expression anywhere you would use an ordinary search value string.

For example, the following command finds all the inport and outport blocks in the clutch model demo provided with Simulink.

```
find_system('clutch', 'regexp', 'on', 'blocktype', 'port')
```

See Also   get_param, set_param

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

gcbh

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

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.

The current system is always the current model or a subsystem of the current model. Use bdroot to get the current model.

Examples   This example returns the path of the system that contains the most recently selected block.

```
gcs
ans =
    clutch/Locked
```

See Also   gcb, bdroot

get_param

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, "Model and Block Parameters," 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)
```

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 7–3.

```
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

new_system

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.
	See Appendix A, "Model and Block Parameters," for a list of the default parameter values for the new system.
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

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

replace_block

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 name/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')
```

See Also   find_system, set_param

save_system

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

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')

set_param

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 4–70.

```
set_param('mymodel/Compute', 'OpenFcn', 'my_open_fcn')
```

See Also   `get_param`, `find_system`

Purpose   Open the Simulink block library.

Syntax   simulink

Description On Microsoft Windows, the simulink command opens (or activates) the Simulink block library browser. On UNIX, the command opens the Simulink library window.

11

Simulink Debugger

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 Simulink debugger has both a graphical and a command-line user interface. The graphical interface allows you to access the debugger's most commonly used features. The command-line interface gives you access to all the debugger's capabilities. Wherever you can use either interface to perform a task, the documentation shows you first how to use the graphical interface and then the command-line interface to perform the task.

Starting the Debugger

To start the debugger, open the model you want to debug and select **Debugger** from the Simulink **Tools** menu. The debugger window appears.
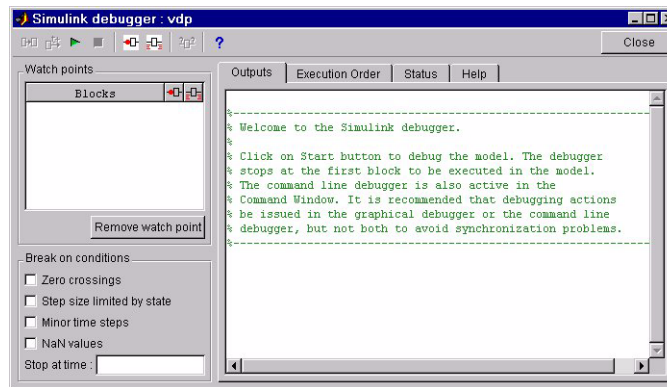
You can also start the debugger from the MATLAB command line, using the sldebug command or the debug option of the sim command to start a model under debugger control. (See sim on page 5-37 for information on specifying sim options.) For example, either the command

    sim('vdp', [0, 10], simset('debug', 'on'))

or the command

    sldebug 'vdp'

loads the Simulink demo model, vdp, into memory, starts the simulation, and stops the simulation at the first block in the model's execution list.

Note When running the debugger in Graphical User Interface (GUI) mode, you must explicitly start the simulation. See "Starting the Simulation" on page 11–4 for more information.

Starting the Simulation

To start the simulation, select the **Start/Continue** button in the debugger's toolbar.

Start/Continue button
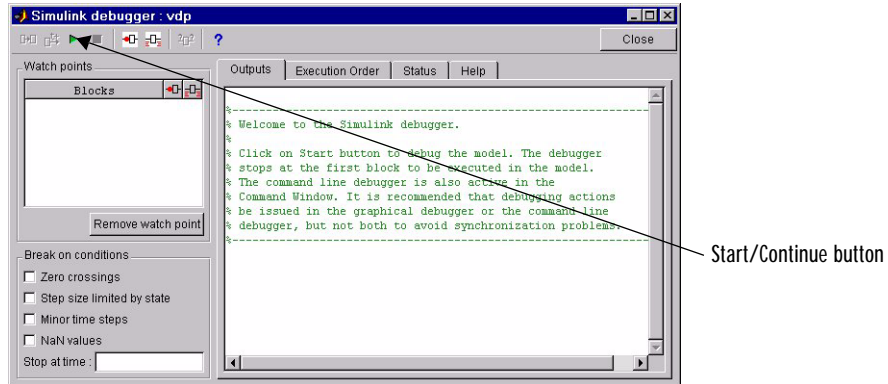
The simulation starts and stops at the first block to be executed. The debugger opens the model window's browser pane and highlights the block at which model execution has stopped.

First block to be executed.
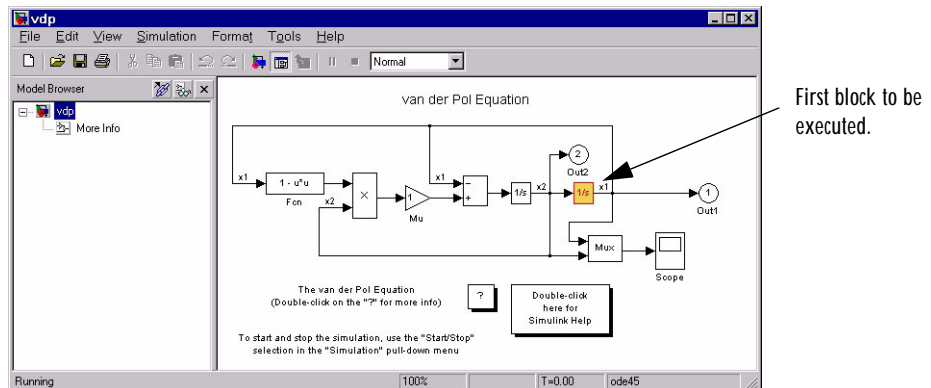
The debugger displays the simulation start time and a debug command prompt in the MATLAB command window when the debugger is running in

command-line mode or in the debugger's output pane when the debugger is running in GUI mode.
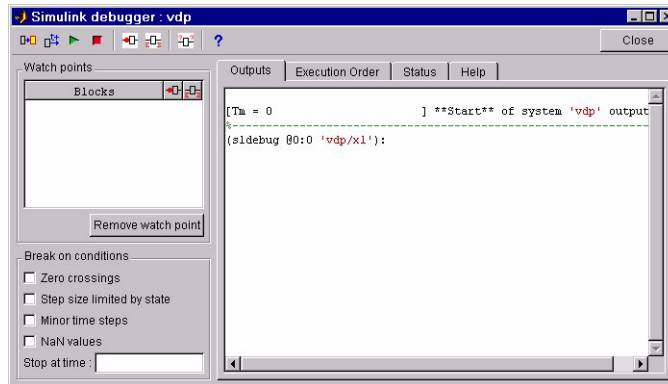
The command prompt displays the block index (see "About Block Indexes" on page 11–6) and name of the first block to be executed.

Note When you start the debugger in GUI mode, the debugger's command-line interface is also active in the MATLAB command window. However, you should avoid using the command-line interface to prevent synchronization errors between the graphical and command line interfaces.

At this point, you can set breakpoints, run the simulation step-by-step, continue the simulation to the next breakpoint or end, examine data, or perform other debugging tasks. The following sections explain how to use the debugger's graphical controls to perform these debugging tasks.

Using the Debugger's Command-Line Interface

In command-line mode, you control the debugger by entering commands at the debugger command line in the MATLAB command window. The debugger accepts abbreviations for debugger commands. See "Debugger Command Reference" on page 11-23 for a list of command abbreviations and repeatable commands. You can repeat some commands by entering an empty command (i.e., by pressing the **Return** key) at the MATLAB command line.

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 11-41).

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 sldebug command, for example, s, which is a partial completion for the step command. Typing an s at the sldebug prompt steps the model However,

    (sldebug...) eval('s')

displays the value of the variable s.

Getting Online Help

You can get online help on using the debugger's by selecting the **Help** button on the debugger's toolbar or by pressing the F1 key when the text cursor is in a debugger panel or text field. Pressing the **Help** button

Help button
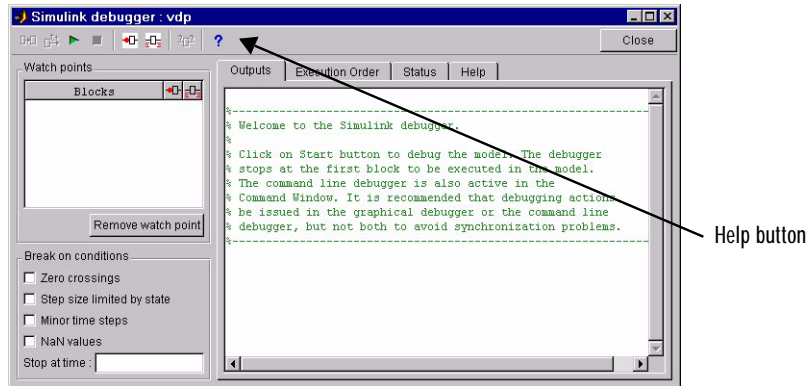
displays help for the debugger in the MATLAB Help browser. Pressing the F1 key displays help for the debugger panel or text field that currently has the keyboard input focus. In command-line mode, you can get a brief description of the debugger commands by typing help at the debug prompt.

Running a Simulation

The Simulink debugger lets you run a simulation from the point at which it is currently suspended to the:

- End of the simulation
- Next breakpoint (see "Setting Breakpoints" on page 11–11)
- Next block
- Next time step

You select the amount to advance by selecting the appropriate button on the debugger toolbar in GUI mode

Next Block   Next Time Step

Start/Continue   Stop
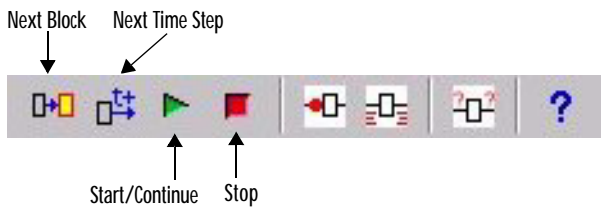
or by entering the appropriate debugger command in command-line mode.

Command	Advances a Simulation…
step	One block
next	One time step
continue	To next breakpoint
run	To end of simulation, ignoring breakpoints

Continuing a Simulation

In GUI mode, the debugger colors the **Run/Continue** button red when it has suspended the simulation for any reason. To continue the simulation, select the **Run/Continue** button. In command-line mode, enter continue to continue the simulation. The debugger continues the simulation to the next breakpoint (see "Setting Breakpoints" on page 11–11) 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.

Advancing to the Next Block

To advance a simulation one block, click ⬛ on the debugger toolbar or, if the debugger is running in command-line mode, 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 11-19). For example, the following figure shows the vdp block diagram after execution of the model's first block.
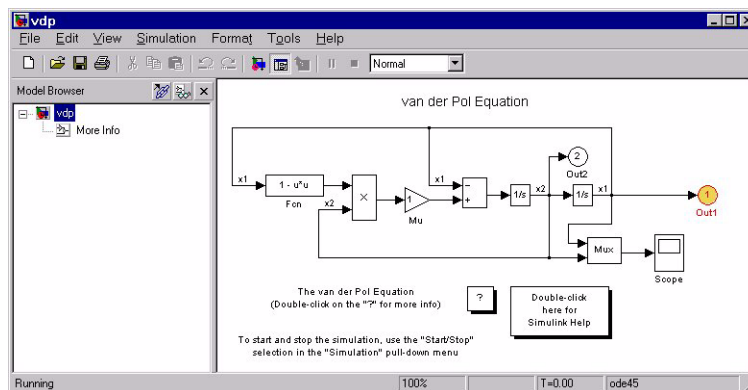
If the next block to be executed occurs in a subsystem block, the debugger opens the subsystem's block diagram and highlights the next block.

After executing a block, the debugger prints the block's inputs (U) and outputs (Y) and redisplays the debug command prompt in the debugger output panel (in GUI mode) or in the MATLAB command window (in command-line mode). The debugger prompt shows the next block to be evaluated.

```
(sldebug @0:0 'vdp/Integrator1'): step
U1 = [0]
Y1 = [2]
(sldebug @0:1 'vdp/Out1'):
```

11-9

Crossing a Time Step Boundary

After executing the last block in the model's block execution list, the debugger advances the simulation to the next time step and halts the simulation. To signal that you have crossed a time step boundary, the debugger prints the current time in the debugger output panel in GUI mode or in the MATLAB command window in command-line mode. For example, stepping through the last block of the first time step of the vdp model results in the following output in the debugger output panel or the MATLAB command window.

```
(sldebug @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, check the **Minor time steps** option on the debugger's **Break on conditions** panel or enter minor at the debugger command prompt.

Advancing to the Next Time Step

To advance to the next time step, click [icon] or enter the next command at the debugger command line. The debugger executes the remaining blocks in the current time step and advances the simulation to the beginning of the next time step. 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
```

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 breakpoint by clicking the breakpoint button on the debugger toolbar
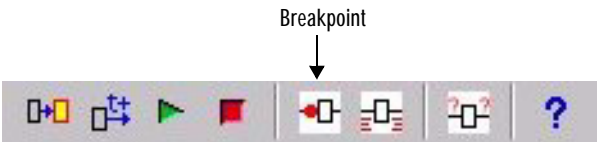
Breakpoint

or checking the appropriate breakpoint conditions (GUI mode)
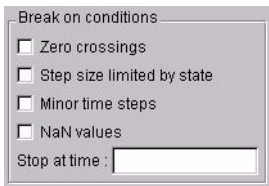
or entering the appropriate breakpoint command (command-line mode).

Command	Causes Simulation to Stop...
break <gcb | s:b>	At the beginning of a block
bafter <gcb | s:b>	At the end of a block
tbreak [t]	At a simulation time step

Command	Causes Simulation to Stop…
nanbreak	At the occurrence of an underflow or overflow (NaN) or infinite (Inf) value
xbreak	When the simulation reaches the state that determines the simulation step size.
zcbreak	When a zero-crossing occurs between simulation time steps.

Setting Breakpoints at Blocks

The debugger lets you specify a breakpoint at the beginning of the execution of a block or at the end of the execution of a block (command-line mode only).

Specifying a Breakpoint at the Start of a Block's Execution

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 graphically or via a block index in command-line mode. To set a breakpoint graphically at the beginning of a block's execution, select the block in the model window and click on the debugger's toolbar or enter

```
break gcb
```

at the debugger command line. To specify the block via its block index (command-line mode only), enter

```
break s:b
```

where s:b is the block's index (see "About Block Indexes" on page 11-6).

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 11–20).

In GUI mode, the debugger's **Watch points** panel displays the blocks where breakpoints exist.
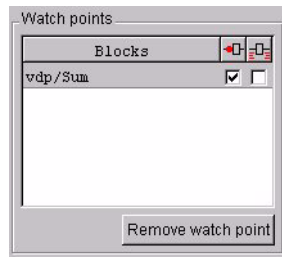
Setting a Breakpoint at the End of a Block's Execution

In command-line mode, the debugger allows you to set a breakpoint at the end of a block's execution, using the bafter command. As with break, you can specify the block graphically or via its block index.

Clearing Breakpoints from Blocks

To clear a breakpoint temporarily, uncheck the first checkbox next to the breakpoint in the **Watch points** panel (GUI mode only). To clear a breakpoint permanently in GUI mode, select the breakpoint in the **Watch points** panel and click the **Remove watch** point button. In command-line mode use the clear command to clear breakpoints. 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.

Setting Breakpoints at Time Steps

To set a breakpoint at a time step, enter a time in the debugger's **Stop at time** field (GUI mode) or enter the time, using the tbreak command. The debugger to stop the simulation at the beginning of the first time step that follows the specified time. For example, starting vdp 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 'vdp' outputs
```

Breaking on Nonfinite Values

Checking the debugger's **NaN values** option or entering the nanbreak command causes the simulation to stop when a computed value is infinite or outside the range of values that can be represented by the machine running the simulation. This option is useful for pinpointing computational errors in a Simulink model.

Breaking on Step-Size Limiting Steps

Checking the **Step size limited by state** option or entering the xbreak command causes the debugger to stop the simulation when the model uses a variable-step solver and the solver encounters a state that limits the size of the steps that it can take. This command is useful in debugging models that appear to require an excessive number of simulation time steps to solve.

Breaking at Zero-Crossings

Checking the **Zero crossings** option or entering 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 allows you to display block I/O by selecting the appropriate buttons on the debugger toolbar

Watch Block I/O   Display Block I/O
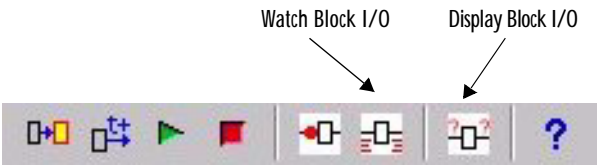
or by entering the appropirate debugger command.

Command	Displays a Block's I/O…
probe	Immediately
disp	At every breakpoint
trace	Whenever the block executes

Displaying I/O of Selected Block

To display the I/O of a block, select the block and click ⟨icon⟩ in GUI mode or enter the probe command in command-line mode.

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.

Command	Description
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 debugger prints the current inputs and outputs of the selected block in the debugger output pane (GUI mode) or the MATLAB command window.

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.

Displaying Block I/O Automatically at Breakpoints

The disp command causes the debugger to display a specified block's inputs and outputs whenever it halts the simulation. You can specify a block either by entering its block index or by selecting it in the block diagram and entering gcb as the disp command argument. You can remove any block from the debugger'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.

Note Automatic display of block I/O at breakpoints is not available in the debugger's GUI mode.

The disp command is useful when you need to monitor the I/O of a specific block or set of blocks as you step through a simulation. Using the disp command, you can specify the blocks you want to monitor and the debugger will then redisplay the I/O of those blocks on every step. Note that the debugger always displays the I/O of the current block when you step through a model

block by block, using the `step` command. So, you do not need to use the `disp` command if you are interested in watching only the I/O of the current block.

Watching Block I/O

To watch a block, select the block and click ⬚ in the debugger toolbar or enter the `trace` command. In GUI mode, if a breakpoint exists on the block, you can set a watch on it as well by checking the watch checkbox for the block in the **Watch points** pane. In command-line mode, you can also specify the block by specifying its block index in the `trace` command. You can remove a block from the debugger's list of trace points, using the `untrace` command.

The debugger displays a watched block's I/O whenever the block executes. Watching a block allows you obtain a complete record of the block's I/O without having to stop the simulation.

Displaying Algebraic Loop Information

The `atrace` command causes the debugger to display information about a model's algebraic loops (see "Algebraic Loops" on page 3-18) 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/Velocity')
(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/Velocity')
```

Displaying Integration Information

The i show command toggles display of integration information. When enabled, this option causes the debugger to print a message each time that the simulation 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. In GUI mode, the debugger displays the sorted list in its **Execution Order** panel. In command-line mode, the slist command displays the model's block execution order in the MATLAB command window. 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' (Outport)
  0:2    'vdp/Integrator2' (Integrator)
  0:3    'vdp/Out2' (Outport)
  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.

```
sldebug clutch
[Tm=0                    ] **Start** of system 'clutch' outputs
```

```
(sldebug @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.

```
sldebug vdp
[Tm=0                       ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): slist
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
  0:0    'vdp/Integrator1' (Integrator)
  0:1    'vdp/Out1' (Outport)
  0:2    'vdp/Integrator2' (Integrator)
  0:3    'vdp/Out2' (Outport)
  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.

```
(sldebug @0:0 'clutch/Clutch Pedal'): zclist
  2:3    'clutch/Unlocked/Sign' (Signum)
  0:4    'clutch/Lockup Detection/Velocities Match' (HitCross)
  0:10   'clutch/Lockup Detection/Required Friction
            for Lockup/Abs' (Abs)
  0:11   'clutch/Lockup Detection/Required Friction for
            Lockup/ Relational Operator' (RelationalOperator)
  0:18   'clutch/Break Apart Detection/Abs' (Abs)
  0:20   'clutch/Break Apart Detection/Relational Operator'
            (RelationalOperator)
  0:24   'clutch/Unlocked' (SubSystem)
  0:27   'clutch/Locked' (SubSystem)
```

Displaying Algebraic Loops

The `ashow` command highlights a specified algebraic loop or the algebraic loop that contains a specified block. To highlight a specified algebraic loop, type `ashow s#n,` where s is the index of the system (see "Displaying a Model's Block Execution Order" on page 11-19) 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 Debugger Status

In GUI mode, the debugger displays the settings of various debug options, such as conditional breakpoints, in its **Status** panel. In command-line mode, the

status command displays debuggers settings. For example, the following sequence of commands displays the initial debug settings for the vdp model.

```
sim('vdp',[0,10],simset('debug','on'))
[Tm=0                       ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): status
  Current simulation time: 0 (MajorTimeStep)
  Last command: ""
  Stop in minor times steps is disabled.
  Break at zero crossing events is disabled.
  Break when step size is limiting by a state is disabled.
  Break on non-finite (NaN,Inf) values is disabled.
  Display of integration information is disabled.
  Algebraic loop tracing level is at 0.
```

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.

Command	Short Form	Repeat	Description
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.

Command	Short Form	Repeat	Description
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

s:b	The block whose system index is s and block index is b.
gcb	Current block.
s#n	The algebraic loop numbered n in system s.
clear	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 nth algebraic loop in system s. ashow clear removes algebraic loop highlights from the model diagram.

See Also atrace, slist

atrace

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.

Command	Displays for Each Algebraic Loop
`atrace 0`	No information
`atrace 1`	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
`atrace 2`	Same as level 1
`atrace 3`	Level 2 plus Jacobian matrix used to solve loop
`atrace 4`	Level 3 plus intermediate solutions of the loop variable

See Also   `systems, states`

Purpose   Insert a break point after a block is executed.

Syntax   bafter gcb
                bafter s:b

Arguments   s:b   The block whose system index is s and block index is b.

                gcb   Current block.

Description The bafter command inserts a breakpoint after execution of the specified
                block.

See Also   break, xbreak, tbreak, nanbreak, zcbreak, slist

break

Purpose	Insert a break point before a block is executed.
Syntax	break gcb
	break s:b
Arguments	s:b The block whose system index is s and block index is b.
	gcb Current block.
Description	The break command inserts a breakpoint before execution of the specified block.
See Also	bafter, tbreak, xbreak, nanbreak, zcbreak, slist

Purpose   Show a specified block.

Syntax   bshow s:b

Arguments   s:b   The block whose system index is s and block index is b.

Description This command opens the model window containing the specified block and
                selects the block.

See Also   slist

clear

Purpose	Clear a breakpoint from a block.
Syntax	clear gcb clear s:b
Arguments	s:b The block whose system index is s and block index is b.
	gcb Current block.
Description	The clear command clears a breakpoint from the specified block.
See Also	bafter, slist

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

disp

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

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.

ishow

Purpose   Enable or disable display of integration information.

Syntax   ishow

Description   The ishow command toggles display of integration information during a simulation.

See Also   atrace

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

nanbreak

Purpose   Set or clear nonfinite value break mode.

Syntax   nanbreak

Description   The nanbreak command causes the debugger to break whenever the simulation encounters a nonfinite (NaN or Inf) value. If nonfinite break mode is set, nanbreak clears it.

See Also   break, bafter, xbreak, tbreak, zcbreak

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

probe

Purpose	Displays a block's state.
Syntax	probe [<s:b	gcb>] [level io	(all)]
Arguments	s:b The block whose system index is s and block index is b.
	gcb Current block.
	level io Display block's I/O.
	level all Display all information regarding a block's current state, including inputs and outputs, states, and zero crossings.
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

Purpose   Abort simulation.

Syntax   quit

Description The quit command terminates the current simulation.

See Also   stop

run

Purpose   Run the simulation to completion.

Syntax   run

Description   The run command runs the simulation from the current breakpoint to its final
                   time step. It ignores breakpoints and display points.

See Also   continue, stop, quit

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`

states

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   i show

Purpose	List a model's nonvirtual systems.
Syntax	systems
Description	The systems command lists a model's nonvirtual systems in the MATLAB command window.
See Also	slist

status

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`

stop

Purpose	Stop the simulation.
Syntax	stop
Description	The stop command stops the simulation.
See Also	continue, run, quit

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

trace

Purpose	Display a block's I/O each time the block executes.
Syntax	`trace gcb` `trace s:b`
Arguments	`s:b` The block whose system index is s and block index is b. `gcb` Current block.
Description	The `trace` command registers a block as a trace point. The debugger displays the I/O of each registered block each time the block executes.
See Also	disp, probe, untrace, slist

Purpose   Remove a block from the debugger's list of display points.

Syntax   undisp gcb
                 undisp s:b

Arguments   s:b   The block whose system index is s and block index is b.

                 gcb   Current block.

Description The undisp command removes the specified block from the debugger's list of
                 display points.

See Also   disp, slist

untrace

Purpose	Remove a block from the debugger's list of trace points.
Syntax	`untrace gcb` `untrace s:b`
Arguments	`s:b` The block whose system index is s and block index is b. `gcb` Current block.
Description	The `untrace` command removes the specified block from the debugger's list of trace points.
See Also	`trace, slist`

Purpose   Break when the debugger encounters a step-size-limiting state.

Syntax   xbreak

Description   The xbreak command pauses execution of the model when the debugger encounters a state that limits the size of the steps that the solver takes. If xbreak mode is already on, xbreak turns the mode off.

See Also   break, bafter, zcbreak, tbreak, nanbreak

zcbreak

Purpose	Toggle breaking at nonsampled zero-crossing events.
Syntax	zcbreak
Description	The zcbreak command causes the debugger to break when a nonsampled zero-crossing event occurs. If zero-crossing break mode is already on, zcbreak turns the mode off.
See Also	break, bafter, xbreak, tbreak, nanbreak, zclist

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

zclist

Performance Tools

About the Simulink Performance Tools Option

The Simulink Performance Tools product includes the following tools:

- Simulink Accelerator
- Model Differencing Tool
- Profiler
- Model Coverage Tool

Note You must have the Performance Tools option installed on your system to use these tools.

The Simulink Accelerator

The Simulink Accelerator speeds up the execution of Simulink models. The Accelerator uses portions of the Real-Time Workshop, a MathWorks product that automatically generates C code from Simulink models, and your C compiler to create an executable. Note that although the Simulink Accelerator takes advantage of Real-Time Workshop technology, the Real-Time Workshop is not required to run it. Also, if you do not have a C compiler installed on your Windows PC, you can use the lcc compiler provided by The MathWorks.

Note You must have the Simulink Performance Tools option installed on your system to use the accelerator.

How Does It Work?

The Simulink Accelerator works by creating and compiling C code that takes the place of the interpretive code that Simulink uses when in Normal mode (that is, when Simulink is not in Accelerator mode). The Accelerator generates the C code from your Simulink model, and MATLAB's mex function invokes your compiler and dynamically links the generated code to Simulink.

The Simulink Accelerator removes much of the computational overhead required by Simulink models when in Normal mode. It works by replacing blocks that are designed to handle any possible configuration in Simulink with compiled versions customized to your particular model's configuration. Through this method, the Accelerator is able to achieve substantial improvements in performance for larger Simulink models. The performance gains are tied to the size and complexity of your model. In general, as size and complexity grow, so do gains in performance. Typically, you can expect a 2X-to-6X gain in performance for models that use built-in Simulink blocks.

How to Run the Simulink Accelerator

To activate the Simulink Accelerator, select **Accelerator** under the
Simulation menu for your model. This picture shows the procedure using the
F14 flight control model.
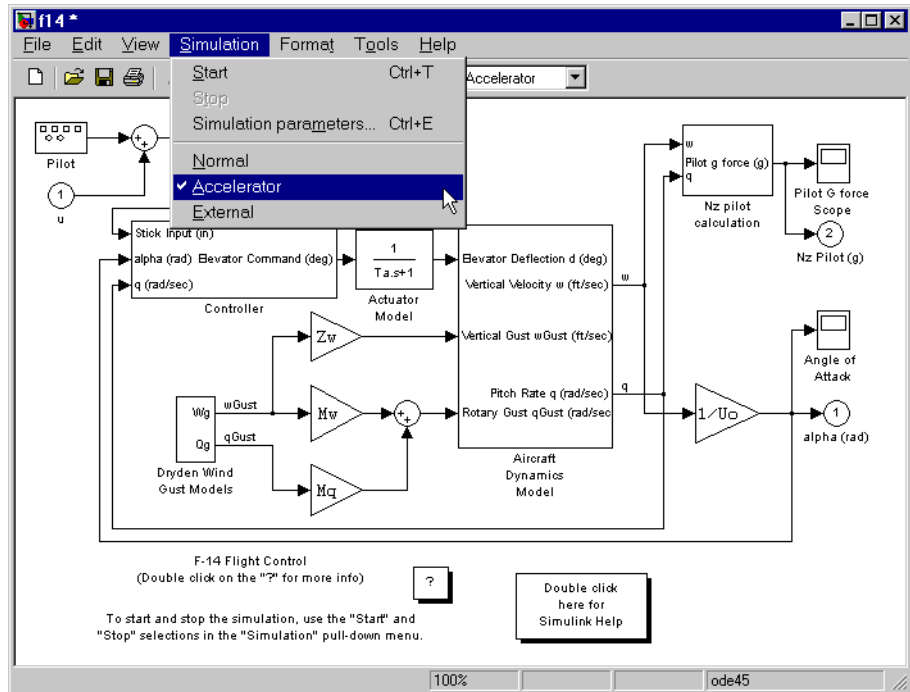
Figure 12-1: Selecting Accelerator Mode in Simulink

Alternatively, you can select **Accelerator** from the menu located on the
right-hand side of the toolbar.

To begin the simulation, select **Start** from the **Simulation** menu. When you
start the simulation, the Accelerator generates the C code and compiles it. The
Accelerator then does the following:

- Places the generated code in a subdirectory called model name_accel _rtw (in
  this case, f14_accel _rtw)
- Places a compiled MEX-file in the current working directory
- Runs the compiled model

Note If your code does not compile, the most likely reason is that you have not set up the `mex` command correctly. Run `mex -setup` at the MATLAB prompt and select your C compiler from the list shown during the setup.

The Accelerator uses Real-Time Workshop technology to generate the code used to accelerate the model. However, the generated code is suitable only for acceleration of the model. If you want to generate code for other purposes, you must use the Real-Time Workshop.

Handling Changes in Model Structure

After you have used the Simulink Accelerator to simulate a model, the MEX-file containing the compiled version of the model remains available for use in later simulations. Even if you exit MATLAB, you can reuse the MEX-file in later MATLAB or Simulink sessions.

If you alter the structure of your Simulink model, for example, by adding or deleting blocks, the Accelerator automatically regenerates the C code and updates (overwrites) the existing MEX-file.

Examples of model structure changes that require the Accelerator to rebuild include:

- Changing the method of integration
- Adding or deleting blocks or connections between blocks
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states in the model
- Changing function in the Trigonometric Function block
- Changing the signs used in a Sum block
- Adding a Target Language Compiler™ (TLC) file to inline an S-function

The Simulink Accelerator displays a warning when you attempt any impermissible model changes during simulation. The warning will not stop the current simulation. To make the model alterations, stop the simulation, make the changes, and restart.

Some changes are permitted in the middle of simulation. Simple changes like adjusting the value of a Gain block do not cause a warning. When in doubt, try to make the change. If you do not see a warning, the Accelerator accepted the change.

Note that the Accelerator does not display warnings that blocks generate *during* simulation. Examples include divide-by-zero and integer overflow. This is a different set of warnings that those discussed above.

Increasing Performance of Accelerator Mode

In general, the Simulink Accelerator creates code optimized for speed with most blocks available in Simulink. There are situations, however, where you can further improve performance by adjusting your simulation or being aware of Accelerator behavior. These include:

- **Simulation Parameters** Pane — The options in the **Simulation Parameters Diagnostics** and **Advanced** panes can affect Accelerator performance. To increase the performance:
  - Disable **Consistency checking** and **Bounds checking** on the **Diagnostics** pane
  - Set **Signal storage reuse** on in the **Advanced** pane
- Stateflow — The Accelerator is fully compatible with Stateflow, but it does not improve the performance of the Stateflow portions of models. Disable Stateflow debugging and animation to increase performance of models that include Stateflow blocks.
- User-written S-functions — The Accelerator cannot improve simulation speed for S-functions unless you inline them using the Target Language Compiler. *Inlining* refers to the process of creating TLC files that direct Real-Time Workshop to create C code for the S-function. This eliminates unnecessary calls to the Simulink application program interface (API).

  For information on how to inline S-functions, consult the *Target Language Compiler Reference Guide*, which is available on the MathWorks Web site, `www.mathworks.com`. It is also available on the documentation CD provided with MATLAB.

- S-functions supplied by Simulink and blocksets — Although the Simulink Accelerator is compatible with all the blocks provided with Simulink and

blocksets, it does not improve the simulation speed for M-file or C-MEX S-Function blocks that do not have an associated inlining TLC file.

- Logging large amounts of data — If you use Workspace I/O, To Workspace, To File, or Scope blocks, large amounts of data will slow the Accelerator down. Try using decimation or limiting outputs to the last N data points.
- Large models — In both Accelerator and Normal mode, Simulink can take significant time to initialize large models. Accelerator speed up can be minimal if run times (from start to finish of a single simulation) are small.

Blocks That Do Not Show Speed Improvements

The Simulink Accelerator is compatible with all MathWorks blocksets, but only two, the Fixed Point Blockset and the DSP Blockset, achieve significantly improved performance with the Accelerator.

Although you can greatly improve simulation performance of your models that use Simulink, Fixed Point Blockset, and DSP Blockset blocks, there is a subset of Simulink and DSP Blockset blocks that are currently not sped up by the Accelerator. The following table lists these blocks.

Table 12-1: Blocks That Do Not Achieve Performance Increases

Simulink Blocks	DSP Blockset Blocks
Display	Biquadratic Filter
From File	Convolution
From Workspace	Direct-Form II Transpose Filter
Inport (root level only)	Dyadic Analysis Filter Bank/ Dyadic Synthesis Filter Bank
MATLAB Fcn	FIR Decimation/FIR Interpolation/ FIR Rate Conversion
Outport (root level only)	From Wave Device/From Wave File
Scope	Integer Delay/Variable Integer Delay
To File	Matrix Multiply/Matrix To Workspace

Table 12-1: Blocks That Do Not Achieve Performance Increases (Continued)

Simulink Blocks	DSP Blockset Blocks
To Workspace	Triggered Signal To Workspace/ Triggered Signal From Workspace
Transport Delay	Time-Varying Direct-Form II Transpose Filter
Variable Transport Delay	To Wave File/To Wave Device
XY Graph	Wavelet Analysis/Wavelet Synthesis
	Zero Pad

In addition, the Accelerator does not speed up user-written S-Function blocks unless you inline them using the Target Language Compiler and have set SS_OPTION_USE_TLC_WITH_ACCELERATOR in the S-function itself. See "Controlling S-Function Execution" on page 12-11 for more information.

Using the Simulink Accelerator with the Simulink Debugger

If you have large and complex models that you need to debug, the Simulink Accelerator can shorten the length of your debugging sessions. For example, if you need to set a time break that is very large, you can use the Accelerator to reach the breakpoint more quickly.

To run the Simulink debugger while in Accelerator mode:

- Select **Accelerator** from the **Simulation** menu, then type

  sldebug model name

  at the MATLAB prompt.
- At the debugger prompt, set a time break,

  tbreak 10000
  continue
- Once you reach the breakpoint, use the debugger command emode (execution mode) to toggle between Accelerator and Normal mode. Note that when the execution is set to Accelerator, block stepping is not permitted.

For more information on the Simulink debugger, see Chapter 11, "Simulink Debugger."

Interacting with the Simulink Accelerator Programmatically

Using three commands, set_param, sim, and accelbuild, you can control the execution of your model from the MATLAB prompt or from M-files. This section describes the syntax for these commands and the options available.

Controlling the Simulation Mode

You can control the simulation mode from the MATLAB prompt using

```
set_param(gcs, 'simulationmode', 'mode')
```

or

```
set_param(model name, 'simulationmode', 'mode')
```

You can use gcs ("get current system") to set parameters for the currently active model (i.e., the active model window) and model name if you want to specify the model name explicitly. The simulation mode can be either normal or accelerator.

Simulating an Accelerated Model

You can also simulate an accelerated model using

```
sim(gcs);   % Blocks the MATLAB prompt until simulation complete
```

or

```
set_param(gcs, 'simulationcommand', 'start'); % Returns to the
                                              % MATLAB prompt
                                              % immediately
```

Again, you can substitute the model name for gcs if you prefer to specify the model explicitly.

Building Simulink Accelerator MEX-Files Independent of Simulation

You can build the Simulink Accelerator MEX-file without actually simulating the model by using the accelbuild command, for example,

```
accelbuild f14
```

Creating the Accelerator MEX-files in batch mode using `accelbuild` allows you to build the C code and executables prior to running your simulations. When you use the Accelerator interactively at a later time, it does not need to generate or compile MEX-files at the start of the accelerated simulations.

You can use the `accelbuild` command to specify build options such as turning on debugging symbols in the generated MEX-file.

```
accelbuild f14 OPT_OPTS=-g
```

Comparing Performance

If you want to compare the performance of the Simulink Accelerator to Simulink in Normal mode, use `tic`, `toc`, and the `sim` command. To run the F14 example, use this code (make sure you're in Normal mode).

```
tic, [t,x,y]=sim('f14', 1000);toc

elapsed_time =

    14.1080
```

In Accelerator mode, this is the result.

```
elapsed_time =

     6.5880
```

The results above were achieved on a Windows PC with a 233 MHz Pentium processor.

Note that for models with very short run times, the Normal mode simulation may be faster, since the Accelerator checks at the beginning of any run to see if it must regenerate the MEX-file. This adds a small overhead to the run-time.

Customizing the Simulink Accelerator Build Process

Typically no customization is necessary for the Simulink Accelerator build process. Since, however, the Accelerator uses the same underlying mechanisms as the Real-Time Workshop to generate code and build the MEX-file, you can use three parameters to control the build process.

```
AccelMakeCommand
AccelSystemTargetFile
```

```
AccelTemplateMakeFile
```

The three options allow you to specify custom Make command, System target, and Template makefiles. Each of these parameters governs a portion of the code generation process. Using these options requires an understanding of how the Real-Time Workshop generates code. For a description of the Make command, the System target file, and Template makefile, see the *Real-Time Workshop User's Guide*, which is available on the MathWorks Web site, www.mathworks.com, and on the documentation CD provided with MATLAB.

The syntax for setting these parameters is

```
set_param(gcs, 'parameter', 'string')
```

or

```
set_param(modelname, 'parameter', 'string')
```

where gcs ("get current system") is the currently active model and 'parameter' is one of the three parameters listed above. Replace string with your string that defines a custom value for that parameter.

Controlling S-Function Execution

Inlining S-functions using the Target Language Compiler increases performance when used with the Simulink Accelerator. By default, however, the Accelerator ignores an inlining TLC file for an S-function, even though the file exists.

One example of why this default was chosen is a device driver S-Function block for an I/O board. The S-function TLC file is typically written to access specific hardware registers of the I/O board. Since the Accelerator is not running on a target system, but rather is a simulation on the host system, it must avoid using the inlined TLC file for the S-function.

Another example is when the TLC file and MEX-file versions of an S-function are not compatible in their use of work vectors, parameters, and/or initialization.

If your inlined S-function is not complicated by these issues, you can direct the Accelerator to use the TLC file instead of the S-function MEX-file by specifying SS_OPTION_USE_TLC_WITH_ACCELERATOR in the mdlInitializeSizes function of the S-function. When set, the Accelerator uses the inlining TLC file and full performance increases are realized.

For example,

```
static void mdlInitializeSizes(SimStruct *S)
{
/* Code deleted */
ssSetOptions(S, SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}
```

Model Differencing Tool

The **Model Differencing Tool** finds and displays differences between two Simulink models. This allows you to determine quickly the differences between, for example, versions of the same model.

Note You must have the Simulink Performance Tools option installed on your system to use this tool.

To use the tool, open one of two models to be compared and select **Model differences** from the Simulink **Tools** menu. The **Model Differencing Tool** appears along with a **Select Second Model** dialog box.
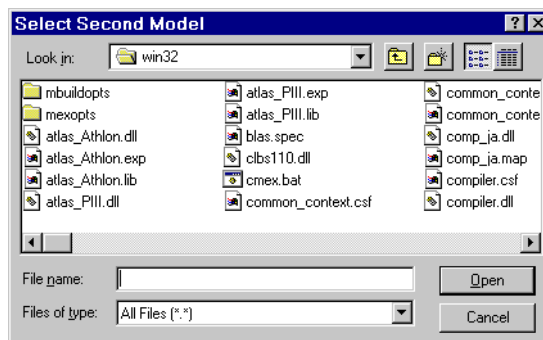
Use the **Select Second Model** dialog box to select the other model to be compared. The **Model Differences Tool** opens the second model, if necessary, and arranges itself alongside the two models.
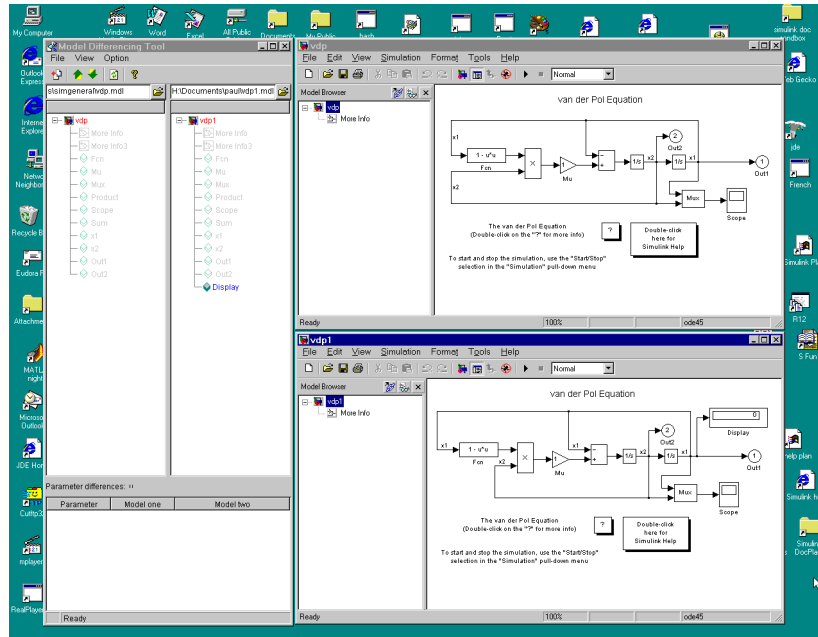
The **Model Differences Tool** contains three panes. The top left pane displays the contents of the first model as an expandable list. The top right pane displays the contents of the second model. Colors indicate differences between the two models.

- Blue marks the blocks that appear in only one of the two models.
- Red marks the blocks that appear in both models but with different parameter values or content (in the case of subsystems).
- Green marks blocks that are identical in both models.

Clicking on a block in either pane highlights the corresponding block icon(s) in the model view(s). The bottom pane displays parameter differences between versions of a selected block that exists in both models.

Display Options

The tool offers some display options. Select **Show items with differences only** from the **Options** menu to omit blocks that do not differ in the two models. Select **Include only non-graphical differences** to display only blocks that differ in parameter values or content. This option omits subsystem blocks that contain only graphical differences, such as block location or background color.

Model Differences Report

Select **HTML Report** from the **View** menu to display an HTML report summarizing the differences between the two models.

The report starts by listing all the blocks that differ between the two models. This summary is followed by difference reports for each block that has different instances in the two models.

Profiler

The Simulink simulation profiler collects performance data while simulating your model and generates a report, called a *simulation profile*, based on the data. The simulation profile generated by the profiler shows you how much time Simulink spends executing each function required to simulate your model. The profile enables you to determine which parts of your model require the most time to simulate and hence where to focus your model optimization efforts.

Note You must have the Simulink Performance Tools option installed on your system to use the profiler.

How the Profiler Works

The following pseudocode summarizes the execution model on which the profiler is based.

```
Sim()
    ModelInitialize().
    ModelExecute()
        for t = tStart to tEnd
        Output()
        Update()
        Integrate()
            Compute states from derivs by repeatedly calling:
                MinorOutput()
                MinorDeriv()
            Locate any zero crossings by repeatedly calling:
                MinorOutput()
                MinorZeroCrossings()
        EndIntegrate
        Set time t = tNew.
    EndModelExecute
    ModelTerminate
EndSim
```

According to this conceptual model, Simulink executes a Simulink model by invoking the following functions zero, one, or more times, depending on the function and the model.

Function	Purpose	Level
sim	Simulate the model. This top-level function invokes the other functions required to simulate the model. The time spent in this function is the total time required to simulate the model.	System
ModelInitialize	Set up the model for simulation.	System
ModelExecute	Execute the model by invoking the output, update, integrate, etc., functions for each block at each time step from the start to the end of simulation.	System
Output	Compute the outputs of a block at the current time step.	Block
Update	Update a block's state at the current time step.	Block
Integrate	Compute a block's continuous states by integrating the state derivatives at the current time step.	Block
MinorOutput	Compute a block's output at a minor time step.	Block
MinorDeriv	Compute a block's state derivatives at a minor time step.	Block
MinorZeroCrossings	Compute a block's zero crossing values at a minor time step.	Block

Function	Purpose	Level
Model Terminate	Free memory and perform any other end-of-simulation cleanup.	System
Nonvirtual Subsystem	Compute the output of a nonvirtual subsystem (see "Atomic Versus Virtual Subsystems" on page 3-13) at the current time step by invoking the output, update, integrate, etc., functions for each block that it contains. The time spent in this function is the time required to execute the nonvirtual subsystem.	Block

The profiler measures the time required to execute each invocation of these functions and generates a report at the end of the model that details how much time was spent in each function.

Enabling the Profiler

To profile a model, open the model and select **Profiler** from the Simulink **Tools** menu. Then start the simulation. When the simulation finishes, Simulink generates and displays the simulation profile for the model in the MATLAB help browser.

The Simulation Profile

Simulink stores the simulation profile in the MATLAB working directory.
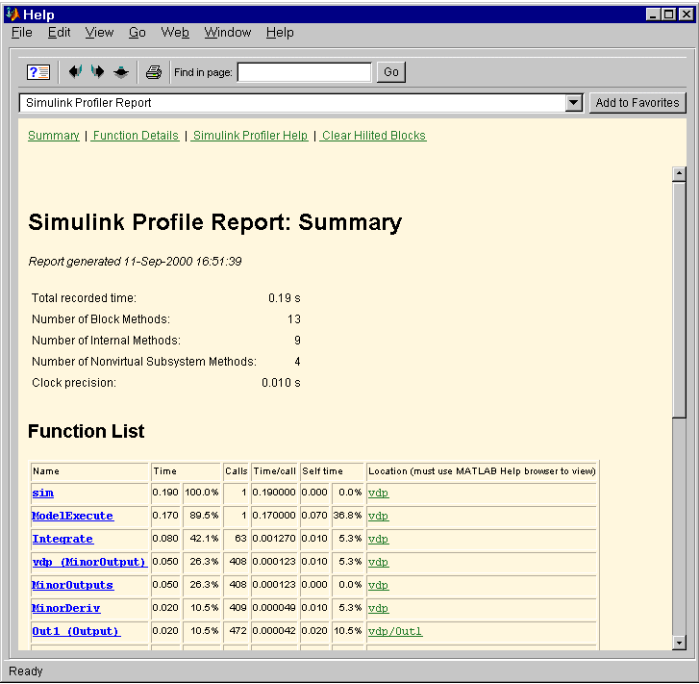
The report has two sections: a summary and a detailed report.

Summary Section

The summary file displays the following performance totals.

Item	Description
Total Recorded Time	Total time required to simulate the model.
Number of Block Methods	Total number of invocations of block-level functions (e.g., Output())

Item	Description
Number of Internal Methods	Total number of invocations of system-level functions (e.g., Model Execute)
Number of Nonvirtual Subsystem Methods	Total number of invocations of nonvirtual subsystem functions
Clock Precision	Precision of the profiler's time measurement

The summary section then shows summary profiles for each function invoked to simulate the model. For each function listed, the summary profile specifies the following information.

Item	Description
Name	Name of function. This item is a hyperlink. Clicking it displays a detailed profile of this function.
Time	Total time spent executing all invocations of this function as an absolute value and as a percentage of the total simulation time
Calls	Number of times this function was invoked
Time/Call	Average time required for each invocation of this function, including the time spent in functions invoked by this function
Self Time	Average time required to execute this function, excluding time spent in functions called by this function
Location	Specifies the block or model executed for which this function is invoked. This item is a hyperlink. Clicking it highlights the corresponding icon in the model diagram. Note that the link works only if you are viewing the profile in the MATLAB help browser.

Detailed Profile Section

This section contains detailed profiles for each function invoked to simulate the model. Each detailed profile contains all the information shown in the summary profile for the function. In addition, the detailed profile displays the function (parent function) that invoked the profiled function and the functions (child functions) invoked by the profiled function. Clicking on the name of the parent or a child function takes you to the detailed profile for that function.

Model Coverage Tool

The Model Coverage Tool determines the extent to which a model test case exercises simulation pathways through a model. The percentage of pathways that a test case exercises is called its *model coverage*. Model coverage is a measure of how thoroughly a test tests a model. The Model Coverage Tool therefore helps you to validate your model tests.

Note You must have the Simulink Performance Tools option installed on your system to use the Model Coverage Tool.

How the Model Coverage Tool Works

The Model Coverage Tool works by analyzing the execution of blocks that serve as decision points in your model. The block types that represent decision points include

- Switch
- Multiport Switch
- Triggered subsystem (Subsystem containing a Trigger block)
- Enabled subsystem (Subsystem containing an Enable block)
- Absolute Value
- Saturation

If a model includes Stateflow charts, the tool also analyzes the states and transitions of those charts. During a simulation run, the tool records changes of state of the branch blocks and of states and transitions. At the end of the simulation, the tool computes for each decision point block and for each state and transition, the ratio of actual branches versus potential branches.

Using the Model Coverage Tool

To develop effective tests with the Model Coverage Tool,

1 Develop one or more test cases for your model (see "Creating and Running Test Cases" on page 12-24).

2 Run the test cases to verify that the model behavior is correct.

3 Analyze the coverage reports produced by Simulink.

4 Using the information in the coverage reports, modify the test cases to increase their coverage or add new test cases that cover areas not covered by the current set of test cases.

5 Repeat the preceding steps until you are satisfied with the coverage of your test set.

Note Simulink comes with an online demonstration of the use of the Model Coverage Tool to validate model tests. To run the demo, type `simcovdemo` at the MATLAB command prompt.

Creating and Running Test Cases

The Test Coverage Tool provides two MATLAB commands, `cvtest` and `cvsim`, for creating and running test cases. The `cvtest` command creates test cases to be run by the `cvsim` command (see "cvsim" on page 12-33 and "cvtest" on page 12-33).

You can also run the coverage tool interactively. To do so, select **Coverage Settings** from the Simulink **Tools** menu. Simulink displays the **Coverage Settings** dialog box (see "Coverage Settings Dialog Box" on page 12-29). Check **Enable Coverage Reporting** and select **OK** to dismiss the dialog. Then select **Start** from the **Simulation** menu or the start button on the Simulink toolbar.

By default, Simulink saves the data as a workspace object named `covdata` and displays the data as an HTML report at the end of the simulation run. You can select other options for generating, saving, and reporting coverage data. See the "Coverage Settings Dialog Box" on page 12-29 for more information.

Note You cannot run simulations with both model coverage reporting and acceleration options enabled. Simulink disables coverage reporting if the accelerator is enabled. If a model includes links to Stateflow library charts and you want the Model Coverage Tool to include the charts in its coverage report, you must open the library charts before starting the simulation. If a referenced library chart is not open, the tool omits the chart from its report.

The Coverage Report

The coverage report generated by the Model Coverage Tool contains the following sections.

Coverage Summary

The coverage summary sections has three subsections.
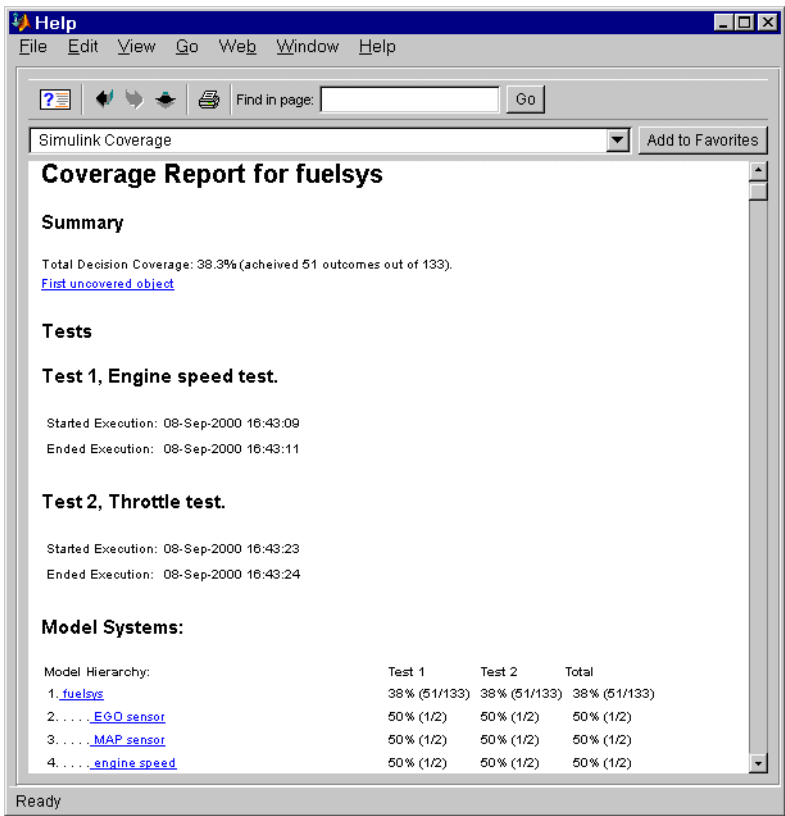
- The "Summary" section gives the total coverage of all test cases for the entire model.
- The "Tests" section lists the simulation start and stop time of each test case and any setup commands that preceded the simulation. The heading for the

each test case includes the test case label, e.g., "Test throttle," specified using the cvtest command.

- The "Model Systems" section summarizes the results for each subsystem. Clicking on the name of the subsystem takes you to a detailed report for that subsystem.

Details Section

The "Details" section reports the model coverage results in detail.
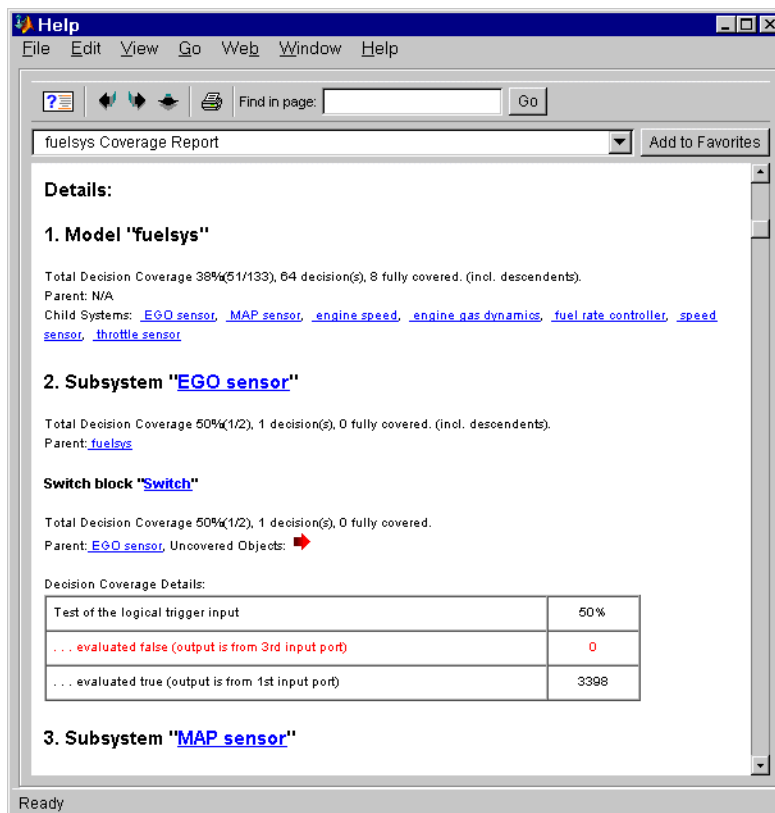
The "Details" section starts with a summary of results for the model as a whole followed by a list of subsystems and charts that the model contains. Subsections on each subsystem and chart follow. Clicking on the name of a

subsystem or chart in the model summary takes you to a detailed report on that subsystem or chart.

Subsystem Report

The section for each subsystem starts with a summary of the test coverage results for the subsystem and a list of the subsystems that it contains. The overview is followed by block reports, one for each block that represents a decision point in the subsystem.

Block Report

The section for each block has a table that lists possible decision outcomes and the number of times that an outcome occurred in each test simulation. The report highlights outcomes that did not occur in red. Clicking on the block name causes Simulink to display the block diagram containing the block. Simulink also highlights the block to help you find it in the diagram.

Note The hyperlinks to the model are valid only for the current MATLAB session. To restore the hyperlinks in a subsequent session, regenerate the report.

The section for each block contains a backward and a forward arrow. Clicking the forward arrow takes you to the next section in the report that lists an uncovered outcome. Clicking the back arrow takes you back to the previous uncovered outcome in the report.

Chart Report

The detailed report for each Stateflow chart has a similar format, with decision tables for each state and transition in the chart.

Coverage Settings Dialog Box

The **Coverage Settings** dialog box allows you to select model coverage reporting options.
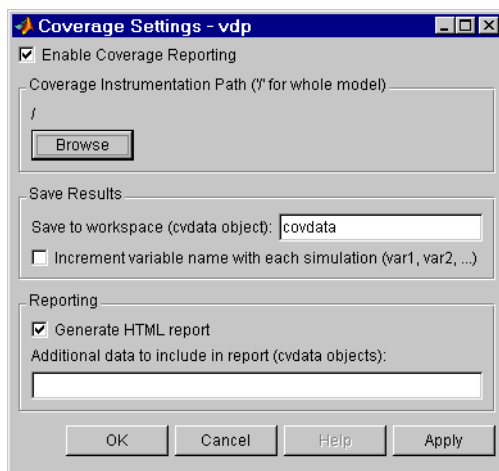
The dialog box includes the following options.

Enable Coverage Reporting

Causes Simulink to gather and report model coverage data during simulation.

Coverage Instrumentation Path

Path of the subsystem for which Simulink gathers and reports coverage data. By default, Simulink generates coverage data for the entire model. To restrict coverage reporting to a particular subsystem, select **Browse**.

Simulink displays a **System Selector** dialog.
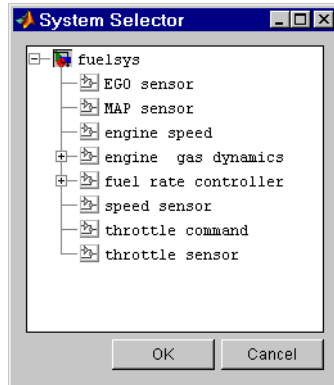
Select the subsystem for which you want coverage reporting to be enabled. Click **OK** to dismiss the dialog.

Save to workspace

Name of workspace object containing coverage data generated by Simulink.

Increment variable name with each simulation

If selected, this option causes Simulink to increment the name of the coverage data object with each simulation. This prevents the current simulation run from overwriting the results of the previous run.

Generate HTML report

Causes Simulink to create an HTML report containing the coverage data. Simulink displays the report in the MATLAB help browser at the end of the simulation.

Additional data to include in report

Names of coverage data from previous runs to include in the current report along with the current coverage data. This option and the previous option allow

you to generate a single report containing the results of multiple simulation runs.

Model Coverage Commands

cvhtml

Produce an HTML report of cvdata object(s).

```
cvhtml(file, data)
```

Create an HTML report of the coverage results in the cvdata object data. The report will be written to file.

```
cvhtml(file, data1, data2,...)
```

Create a combined report of several data objects. The results from each object will be displayed in a separate column. Each data object must correspond to the same root subsystem or the function will produce errors.

```
cvhtml(file, data, data2,..., detail)
```

Specify the detail level of the report with the value of detail, an integer between 0 and 3. Greater numbers indicate greater detail. The default value is 2.

cvload

Load coverage tests and results from file.

```
[TESTS, DATA] = CVLOAD(FILENAME)
```

Load the tests and data stored in the text file FILENAME.CVT. The tests that are successfully loaded are returned in TESTS, a cell array of cvtest objects. DATA is a cell array of cvdata objects that were successfully loaded. DATA has the same size as TESTS but may contain empty elements if a particular test has no results.

Special considerations:

• If a model with the same name exists in the coverage database, only the compatible results will be loaded from file and they will reference the existing model to prevent duplication.

- If the Simulink models referenced from the file are open but do not exist in the coverage database, the coverage tool resolves the links to the existing models.

- When loading several files that reference the same model, only the results that are consistent with the earlier files will be loaded.

cvreport

Report the information in a cvdata object. This command has the following forms.

```
cvreport(file, data)
```

Create a text report of the coverage results in the cvdata object data. The report will be written to file. If file is empty the report will be displayed at the command prompt.

```
cvreport(file, data1, data2, ...)
```

Create a combined report of several test objects. The results from each object will be displayed in a separate column. Each data object must correspond to the same root subsystem or the function will produce errors.

```
cvreport(file, data1, data2, ..., detail)
```

Specify the detail level of the report with the value of detail, an integer between 0 and 3. Greater numbers indicate greater detail. The default value is 2.

cvsave

Save coverage tests and results to file.

```
cvsave(filename, model)
```

Save all the tests and results related to model in the text file filename. cvt.

```
cvsave(filename, test1, test2, ...)
```

Save the specified tests in the text file filename. cvt. Information about the referenced model(s) is also saved.

```
cvsave(filename, data1, data2, ...)
```

Save the specified data objects, the tests that created them, and the referenced model(s) structure in the text file filename.cvt.

cvsim

Run a test case.

Note You do not have to enable model coverage reporting (see "Creating and Running Test Cases" on page 12-24) to use this command.

This command can take the following forms.

```
data = cvsim(test)
```

Execute the cvtest object test by starting a simulation run for the corresponding model. The results are returned in a cvdata object.

```
[data, t, x, y] = cvsim(test)
```

Returns the simulation time vector, t, state values, x, and output values, y.

```
[data,t,x,y] = cvsim( test, timespan, options)
```

Override the default simulation values. For more information see the sim command.

```
[data1, data2, ...] = cvsim( test1, test2, ... )
```

Execute a set of tests and return the results in cvdata objects.

```
[data1, t, x, y] = cvsim(root, label, setupcmd)
```

Create and execute a cvtest object.

cvtest

Creates a test case. This command has the following syntax.

```
test = cvtest(root, label, setupcmd)
```

where root is the name or handle to the model or subsystem to be tested, label is a string that identifies the test case, and setupcmd is a MATLAB command that cvsim executes in the base workspace before running the instrumented

model. The second two arguments are optional. The `cvtest` command returns a handle to the registered test case.

A

Model and Block Parameters

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. See set_param on page 10-27 for more information on this command.

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 4–70. 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 "The Simulation Parameters Dialog Box" on page 5-8. 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
AbsTol	Absolute error tolerance	scalar {1e–6}
AlgebraicLoopMsg	Algebraic loop diagnostic	none | {warning} | error
ArrayBoundsChecking	Enable array bounds checking	'none' | 'warning' | 'error'
BooleanDataType	Enable Boolean mode	on | {off}
BufferReuse	Enable reuse of block I/O buffers	{on} | off
CloseFcn	Close callback	command or variable
ConfigurationManager	Configuration manager for this model.	text
ConsistencyChecking	Consistency checking	on | {off}
Created	Date and time model was created.	text
Creator	Name of model creator.	text
Decimation	Decimation factor	scalar {1}
Description	Description of this model.	text
ExternalInput	Time and input variable names	scalar or vector [t, u]

Table A-1: Model Parameters (Continued)

Parameter	Description	Values
`FinalStateName`	Final state name	variable `{xFinal}`
`FixedStep`	Fixed step size	scalar `{auto}`
`InitialState`	Initial state name or values	variable or vector `{xInitial}`
`InitialStep`	Initial step size	scalar `{auto}`
`InvariantConstants`	Invariant constant setting	`on` | `{off}`
`LimitDataPoints`	Limit output	`on` | `{off}`
`LoadExternalInput`	Load input from workspace	`on` | `{off}`
`LoadInitialState`	Load initial state	`on` | `{off}`
`MaxDataPoints`	Maximum number of output data points to save	scalar `{1000}`
`MaxOrder`	Maximum order for ode15s	`1` | `2` | `3` | `4` | `{5}`
`MaxStep`	Maximum step size	scalar `{auto}`
`MinStepSizeMsg`	Minimum step size diagnostic	`{warning}` | `error`
`ModelVersionFormat`	Format of model's version number.	`text`
`ModifiedBy`	Last modifier of this model.	`text`
`ModifiedDateFormat`	Format of modified date.	`text`
`Name`	Model name	`text`
`ObjectParameters`	Names/attributes of model parameters.	`structure`
`OutputOption`	Output option	`AdditionalOutputTimes` | `{RefineOutputTimes}` | `SpecifiedOutputTimes`
`OutputSaveName`	Simulation output name	variable `{yout}`
`OutputTimes`	Values for chosen `OutputOption`	vector `{[]}`
`PaperOrientation`	Printing paper orientation	`portrait` | `{landscape}`
`PaperPosition`	Position of diagram on paper	`[left, bottom, width, height]`

Table A-1: Model Parameters (Continued)

Parameter	Description	Values
PaperPositionMode	Paper position mode	auto | {manual}
PaperSize	Size of PaperType in PaperUnits	[width height] (read only)
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
PostLoadFcn	Post-load callback	command or variable
PreLoadFcn	Pre-load callback	command or variable
Refine	Refine factor	scalar {1}
RelTol	Relative error tolerance	scalar {1e–3}
SampleTimeColors	**Sample Time Colors** menu option	on | {off}
SaveFcn	Save callback	command or variable
SaveFinalState	Save final state	on | {off}
SaveFormat	Format used to save data to the MATLAB workspace	Array | Structure | StructureWithTime
SaveOutput	Save simulation output	{on} | off
SaveState	Save states	on | {off}
SaveTime	Save simulation time	{on} | off
ShowLineWidths	**Show Line Widths** menu option	on | {off}
SimParamPage	**Simulation Parameters** dialog box page to display (page last displayed)	{Solver} | WorkspaceI/O | Diagnostics

Table A-1: Model Parameters (Continued)

Parameter	Description	Values
Solver	Solver	{ode45} | ode23 | ode113 | ode15s | ode23s | ode5 | ode4 | ode3 | ode2 | ode1 | FixedStepDiscrete | VariableStepDiscrete
StartFcn	Start simulation callback	command or variable
StartTime	Simulation start time	scalar {0.0}
StateSaveName	State output name	variable {xout}
StopFcn	Stop simulation callback	command or variable
StopTime	Simulation stop time	scalar {10.0}
TimeSaveName	Simulation time name	variable {tout}
UnconnectedInputMsg	Unconnected input ports diagnostic	none | {warning} | error
UnconnectedLineMsg	Unconnected lines diagnostic	none | {warning} | error
UnconnectedOutputMsg	Unconnected output ports diagnostic	none | {warning} | error
Version	Simulink version used to modify the model (read-only)	(release)
WideVectorLines	**Wide Vector Lines** menu option	on | {off}
ZeroCross	Intrinsic zero crossing detection (see "Zero Crossing Detection" on page 3–14)	{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 4–70. Examples of commands that change these parameters follow this table.

Table A-2: Common Block Parameters

Parameter	Description	Values
`AttributesFormat String`	Specifies parameters to be displayed below block in a block diagram	string
`BackgroundColor`	Block icon background	`black` | `{white}` | `red` | `green` | `blue` | `cyan` | `magenta` | `yellow` | `gray` | `lightBlue` | `orange` | `darkGreen`
`BlockDescription`	Block description	text
`BlockType`	Block type	text
`CloseFcn`	Close callback	MATLAB expression
`CompiledPortWidths`	Structure of port widths	`scalar and vector`
`CopyFcn`	Copy callback	MATLAB expression
`DeleteFcn`	Delete callback	MATLAB expression
`Description`	User-specifiable description	text
`DialogParameters`	Names/attributes of parameters in blocks parameter dialog,	structure
`DropShadow`	Display drop shadow	`{off}` | `on`
`FontAngle`	Font angle	(system-dependent) `{normal}` | `italic` | `oblique`
`FontName`	Font	`{Helvetica}`
`FontSize`	Font size	`{10}`
`FontWeight`	Font weight	(system-dependent) `light` | `{normal}` | `demi` | `bold`

Table A-2: Common Block Parameters (Continued)

Parameter	Description	Values
ForegroundColor	Block name, icon, outline, output signals, and signal label	{black} | white | red | green | blue | cyan | magenta | yellow | gray | lightBlue | orange | darkGreen
InitFcn	Initialization callback	MATLAB expression
InputPorts	Array of input port locations	[h1,v1; h2,v2; ...]
LinkStatus	Link status of block.	none |resolved | unresolved | implicit
LoadFcn	Load callback	MATLAB expression
ModelCloseFcn	Model close callback	MATLAB expression
Name	Block's name	string
NameChangeFcn	Block name change callback	MATLAB expression
NamePlacement	Position of block name	{normal} | alternate
ObjectParameters	Names/attributes of block's parameters	structure
OpenFcn	Open callback	MATLAB expression
Orientation	Where block faces	{right} | left | down | up
OutputPorts	Array of output port locations	[h1,v1; h2,v2; ...]
Parent	Name of the system that owns the block	string
ParentCloseFcn	Parent subsystem close call-back	MATLAB expression
Position	Position of block in model window	vector [left top right bottom] *not* enclosed in quotes
PostSaveFcn	Post-save callback	MATLAB expression
PreSaveFcn	Pre-save callback	MATLAB expression
Selected	Block selected state	on | {off}
ShowName	Display block name	{on} | off

Table A-2: Common Block Parameters (Continued)

Parameter	Description	Values
StartFcn	Start simulation callback	MATLAB expression
StopFcn	Termination of simulation callback	MATLAB expression
Tag	User-defined string	' '
Type	Simulink object type (read-only)	'block'
UndoDeleteFcn	Undo block delete callback	MATLAB expression
UserData	Any MATLAB data type (not saved in the mdl file)	[ ]

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])
```

A-9

Block-Specific Parameters

These tables list block-specific parameters for all Simulink blocks. The type of the block appears in parentheses after the block name. Some Simulink blocks are implemented as masked subsystems. The tables indicate masked blocks by adding the designation "masked" after the block type.

Note The type listed for nonmasked blocks is the value of the block's BlockType parameter; the type listed for masked blocks is the value of the block's MaskType parameter. For more information, see "Mask Parameters" on page A-25.

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

Block (Type)/Parameter	Dialog Box Prompt	Values
Band-Limited White Noise (Continuous White Noise) (masked)		
Chirp Signal (chirp) (masked)		
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Clock (Clock) (no block-specific parameters)		
Constant (Constant)		
Value	Constant value	scalar or vector {1}
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Digital Clock (Digital Clock)		
SampleTime	Sample time	scalar (sample period) {1} or vector [period offset]

Table A-3: Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
Digital Pulse Generator		
VectorParams1D	Interpret vector parameters as 1-D	off {on}
From File (FromFile)		
FileName	Filename	filename {untitled.mat}
From Workspace (FromWorkspace)		
VariableName	Matrix table	matrix {[T,U]}
Pulse Generator (Pulse Generator) (masked)		
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Ramp (Ramp) (masked)		
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Random Number (RandomNumber)		
Seed	Initial seed	scalar or vector {0}
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Repeating Sequence (Repeating table) (masked)		
Signal Generator (Signal Generator)		
WaveForm	Wave form	{sine} | square | sawtooth | random
Amplitude	Amplitude	scalar or vector {1}
Frequency	Frequency	scalar or vector {1}
Units	Units	{Hertz} | rad/sec
VectorParams1D	Interpret vector parameters as 1-D	off {on}

A-11

Table A-3: Sources Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
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]
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Step (Step)		
Time	Step time	scalar or vector {1}
Before	Initial value	scalar or vector {0}
After	Final value	scalar or vector {1}
VectorParams1D	Interpret vector parameters as 1-D	off {on}
Uniform Random Number (Uniform RandomNumber)		
Minimum	Minimum	scalar or vector {−1}
Maximum	Maximum	scalar or vector {1}
Seed	Initial Seed	scalar or vector {0}
SampleTime	Sample Time	scalar or vector {0}
VectorParams1D	Interpret vector parameters as 1-D	off {on}

Table A-4: Sinks Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
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}`
`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}`

Table A-4: Sinks Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
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}
SampleTime	Sample time	scalar (sample period) {−1} or vector [period offset]
XY Graph (XY scope.) (masked)		

Table A-5: Discrete Library Block Parameters

Block (Type)/Parameter	Dialog Box Prompt	Values
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}
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

A-15

Table A-5: Discrete Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
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 (Type)/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)

Block (Type)/Parameter	Dialog Box Prompt	Values
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

Block (Type)/Parameter	Dialog Box Prompt	Values
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)

Block (Type)/Parameter	Dialog Box Prompt	Values
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 (`Relational Operator`)		
`Operator`	Operator	== | != | < | {<=} | >= | >
Relational Operator (`Relational Operator`)		
`Operator`	Operator	== | != | < | {<=} | >= | >
Rounding Function (`Rounding`)		
`Operator`	Function	{floor} | ceil | round | fix
Sign (`Signum`) (no block-specific parameters)		
Slider Gain (`SliderGain`) (masked)		

A-19

Table A-7: Math Library Block Parameters (Continued)

Block (Type)/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 (Type)/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)) (masked)		
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

Block (Type)/Parameter	Dialog Box Prompt	Values
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 (`Manual Switch`) (masked)		
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)

Block (Type)/Parameter	Dialog Box Prompt	Values
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

Block (Type)/Parameter	Dialog Box Prompt	Values
Bus Selector (BusSelector)		
InputSignals		cell array of the input signals nested to reflect the signal hierarchy
Configurable Subsystem (mask)		
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}

Table A-10: Signals & Systems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
SampleTime	Sample time	scalar (sample period) {−1} or vector [period offset]
Data Type Conversion		
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}

Table A-10: Signals & Systems Library Block Parameters (Continued)

Block (Type)/Parameter	Dialog Box Prompt	Values
PortWidth	Port width	scalar {−1}
SampleTime	Sample time	scalar (sample period) {−1} or vector [period offset]
Merge		
Model Info (CMBlock) (mask)		
Mux (Mux)		
Inputs	Number of inputs	scalar or vector {3}
Out (Outport)		
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
ProbeCompexSignal	Probe complex signal	{on} | off
Subsystem (SubSystem)		
ShowPortLabels	Show/Hide Port Labels **Format** 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	Description/Prompt	Values
Mask	Turns mask on or off.	{on}	off
MaskCallbackString	Mask parameter callbacks	delimited string
MaskCallbacks	Mask parameter callbacks	cell array
MaskDescription	Block description	string
MaskDisplay	Drawing commands	display commands
MaskEditorHandle	Mask editor figure handle (for internal use)	handle
MaskEnableString	Mask parameter enable status	delimited string
MaskEnables	Mask parameter enable status	cell array of strings, each either 'on' or 'off'
MaskHelp	Block help	string
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
MaskInitialization	Initialization commands	MATLAB command
MaskNames		
MaskPrompts	Prompt (see below)	cell array of strings
MaskPromptString	Prompt (see below)	delimited string

Table A-11: Mask Parameters (Continued)

Parameter	Description/Prompt	Values
`MaskPropertyNameString`		
`MaskSelfModifiable`	Indicates that the block can modify itself.	`on	{off}`
`MaskStyles`	Control type (see below)	cell array `{Edit}	Checkbox	Popup`
`MaskStyleString`	Control type (see below)	`{Edit}	Checkbox	Popup`
`MaskTunableValues`	Tunable parameter attributes	cell array of strings
`MaskTunableValueString`	Tunable parameter attributes	delimited string
`MaskType`	Mask type	string
`MaskValues`	Block parameter values (see below)	`cell array of strings`
`MaskValueString`	Block parameter values (see below)	delimited string
`MaskVariables`	Variable (see below)	string
`MaskVisibilities`	Specifies visibility of parameters	

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

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.

Index