Simulink®

Simulation and Model-Based Design

- Modeling
- Simulation
- Implementation

Using Simulink®

Version 6

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`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
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

Using Simulink

Printing History:			
	November 1990	First printing	New for Simulink 1
	December 1996	Second printing	Revised for Simulink 2
	January 1999	Third printing	Revised for Simulink 3 (Release 11)
	November 2000	Fourth printing	Revised for Simulink 4 (Release 12)
	July 2002	Fifth printing	Revised for Simulink 5 (Release 13)
	April 2003	Online only	Revised for Simulink 5.1 (Release 13SP1)
	April 2004	Online only	Revised for Simulink 5.1.1 (Release 13SP1+)
	June 2004	Sixth printing	Revised for Simulink 6.0 (Release 14)

Contents

Getting Started

1

How Simulink Works

2

3 Simulink Basics

4 Creating a Model

5 Working with Blocks

6 Working with Signals

7 Working with Data

8 Modeling with Simulink

9 Exploring, Searching, and Browsing Models

10 Running Simulations

11 Analyzing Simulation Results

12 Creating Masked Subsystems

13 Simulink Debugger

14 Simulink Accelerator

15 Using the Embedded MATLAB Function Block

Index

1

Getting Started

The following sections use examples to give you a quick introduction to using Simulink® to model and simulate dynamic systems.

What Is Simulink?

Simulink® is a software package for modeling, simulating, and analyzing dynamic 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.

Tool for Interactive Simulation

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 the analysis tools in MATLAB®, so you can take the results and analyze and visualize them. A goal of Simulink is to give you 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.

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.

Tool for Model-Based Design

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. Simulink 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.

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 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 the MATLAB 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.

Related Products

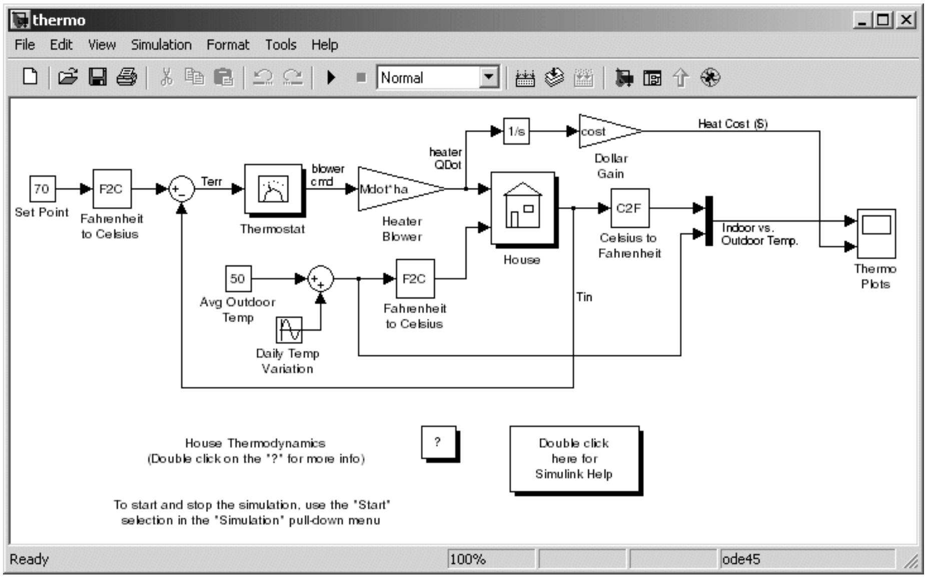
The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with Simulink and that extend the capabilities of Simulink. For information about these related products, see `http://www.mathworks.com/products/simulink/related.html`.

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.



House Thermodynamics
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start" selection in the "Simulation" pull-down menu

Double click here for Simulink Help

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, click 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 click 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 1-6.

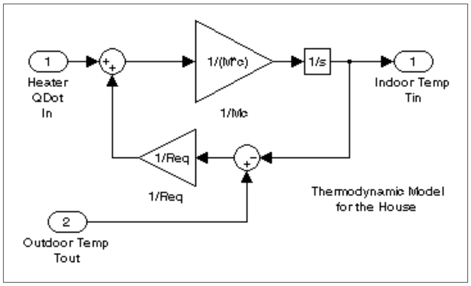
6 When you're finished running the simulation, close the model by choosing **Close** from the **File** menu.

Description of the Demo

The demo models the thermodynamics of a house. 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.

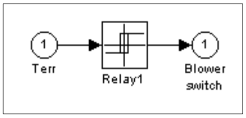
The model uses subsystems to simplify the model diagram and create reusable systems. A subsystem is a group of blocks that is represented by a Subsystem block. This model contains five subsystems: one named Thermostat, one named House, and three Temp Convert subsystems (two convert Fahrenheit to Celsius,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

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



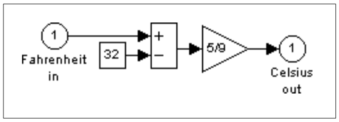
House subsystem

The Thermostat subsystem models the operation of a thermostat, determining when the heating system is turned on and off. Double-click 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.
- 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 you are building models:

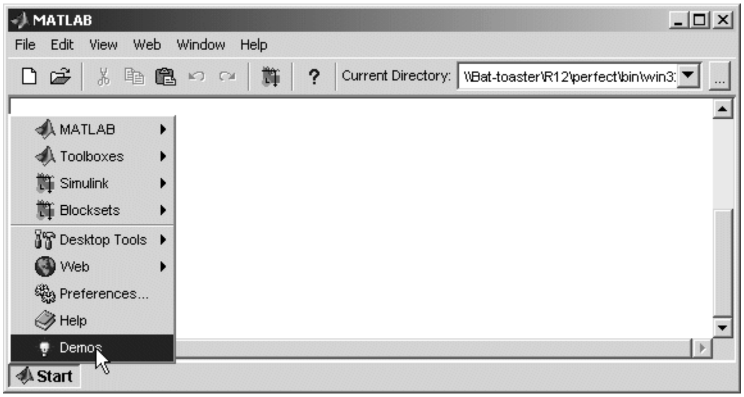
- Running the simulation involves specifying parameters and starting the simulation with the **Start** command, described in “Diagnosing Simulation Errors” on page 10-72.
- You can encapsulate complex groups of related blocks in a single block, called a subsystem. See “Creating Subsystems” on page 4-20 for more information.
- You can customize the appearance of and design a dialog box for a block by using the masking feature, described in detail in Chapter 12, “Creating Masked Subsystems.” The `thermo` model uses the masking feature to customize the appearance of all the Subsystem blocks that it contains.
- Scope blocks display graphic output much as an actual oscilloscope does.

Other Useful Demos

Other demos illustrate useful modeling concepts. You can access these demos from the MATLAB Command Window:

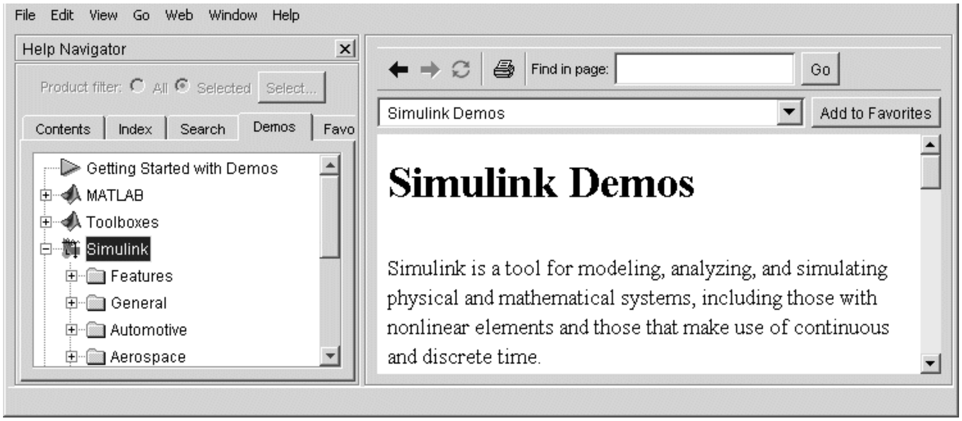
1 Click the **Start** button on the bottom left corner of the MATLAB Command Window.

The **Start** menu appears.


MATLAB
File Edit View Web Window Help
Current Directory: \\Bat-toaster\R12\perfect\bin\win3:
MATLAB
Toolboxes
Simulink
Blocksets
Desktop Tools
Web
Preferences...
Help
Demos
Start

2 Select **Demos** from the menu.

The MATLAB Help browser appears with the **Demos** pane selected.



Simulink Demos

Simulink is a tool for modeling, analyzing, and simulating physical and mathematical systems, including those with nonlinear elements and those that make use of continuous and discrete time.

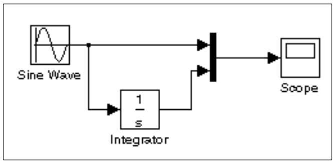
3 Click the **Simulink** entry in the **Demos** pane.

The entry expands to show groups of Simulink demos. Use the browser to navigate to demos of interest. The browser displays explanations of each demo and includes a link to the demo itself. Click on a demo link to start the demo.

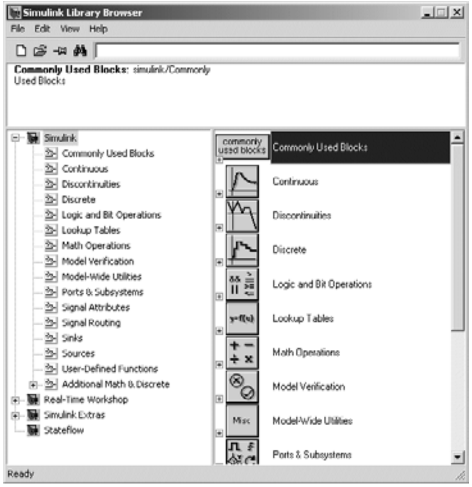
Building a Model

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 the tasks are described in more detail in the next chapter.

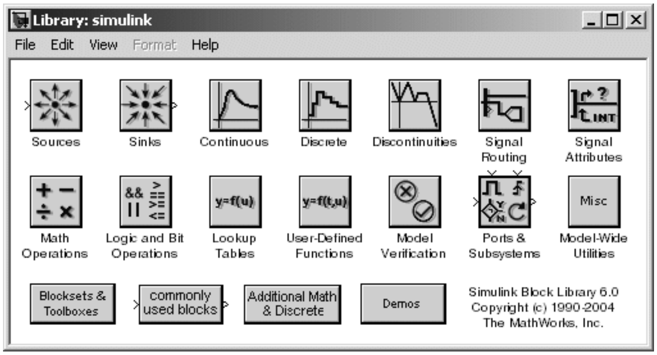
The model integrates a sine wave and displays the result along with the sine wave. The block diagram of the model looks like this.



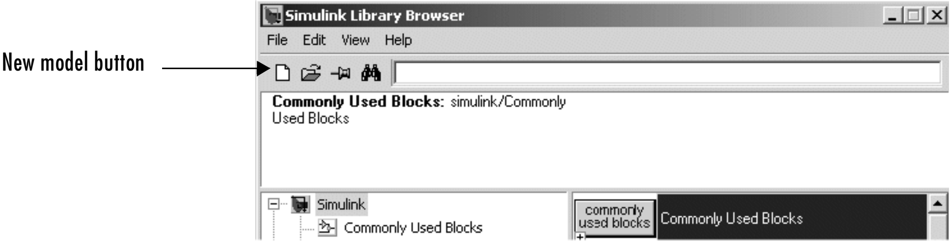
To create the model, first enter `simulink` in the MATLAB Command Window. On Microsoft Windows, the Simulink Library Browser appears.



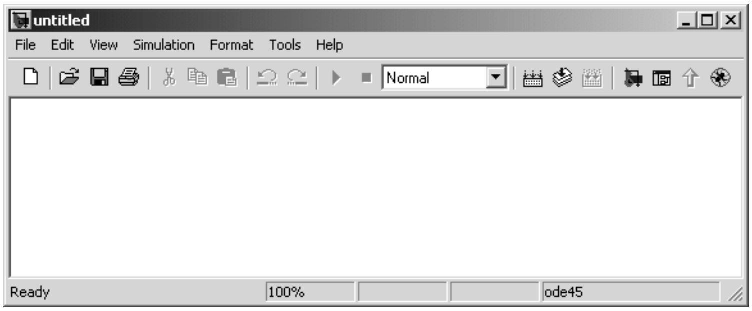
On UNIX, the Simulink library window appears.



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, click the New Model button on the Library Browser's toolbar.



Simulink opens a new model window.



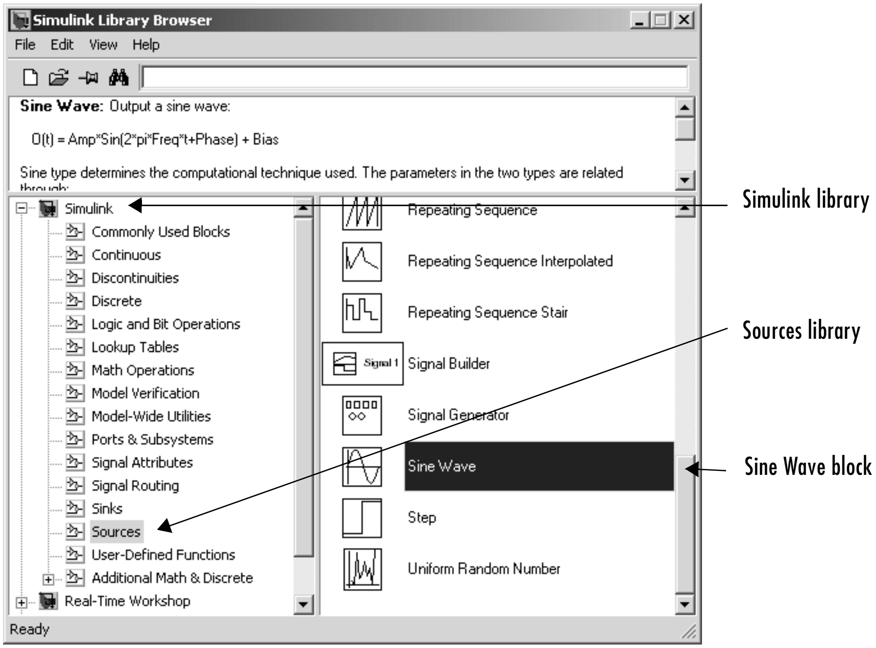
To create this model, you need to copy blocks into the model from the following Simulink block libraries:

- Sources library (the Sine Wave block)
- Sinks library (the Scope block)
- Continuous library (the Integrator block)
- Signal Routing 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 and 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 the Sources node to display the Sources library blocks. Finally, click the Sine Wave node to select the Sine Wave block.

Here is how the Library Browser should look after you have done this.



Sine Wave: Output a sine wave:

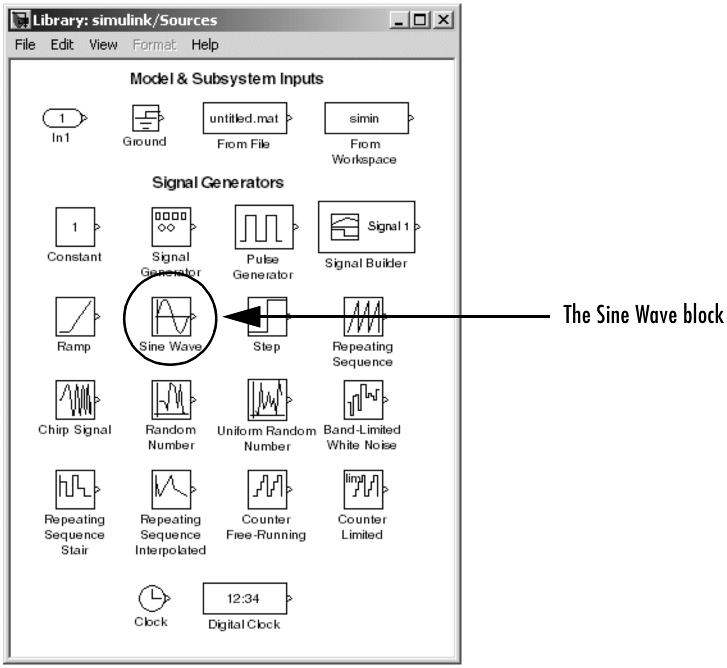
O(t) = Amp*Sin(2*pi*Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

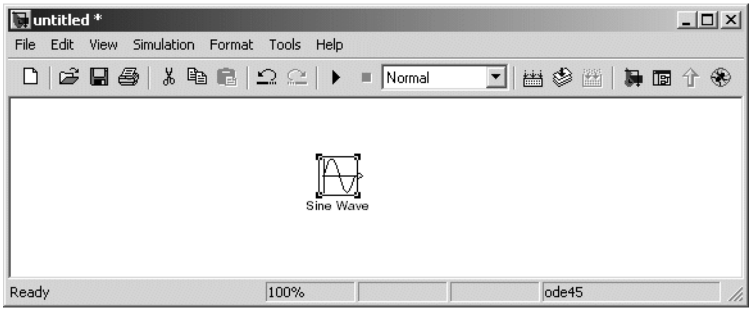
Now drag a copy of the Sine Wave block from the browser and drop it in the model window.

To copy the Sine Wave block from the Sources library window, open the Sources window by double-clicking the Sources icon in the Simulink library window. (On Windows, you can open the Simulink library window by right-clicking the Simulink node in the Library Browser and then clicking the resulting **Open Library** button.)

Simulink displays the Sources library window.

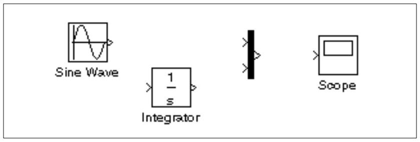


Now drag the Sine Wave block from the Sources window to your model window.

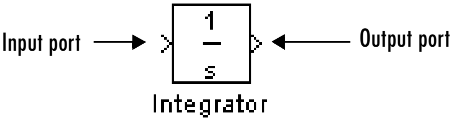


Copy the rest of the blocks in a similar manner from their respective libraries into the model window. You can move a block from one place in the model window to another by dragging the block. You can move a block a short distance by selecting the block, then pressing the arrow keys.

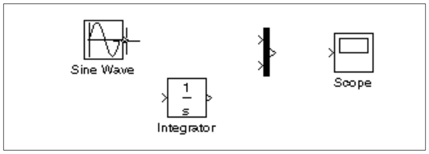
With all the blocks copied into the model window, the model should look something like this.



If you examine the blocks, 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.

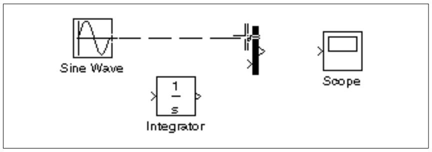


Now it's time to connect the blocks. Connect the Sine Wave block to the top input port of the Mux block. Position the pointer over the output port on the right side of the Sine Wave block. Notice that the cursor shape changes to crosshairs.

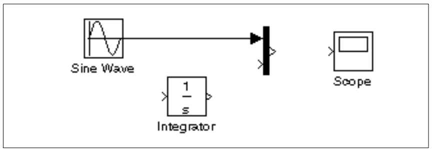


Hold down the mouse button and move the cursor to the top input port of the Mux block.

Notice that the line is dashed while the mouse button is down and that the cursor shape changes to double-lined crosshairs as it approaches the Mux block.



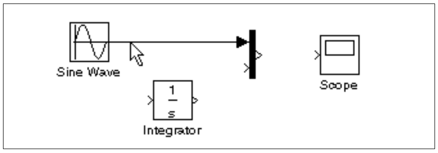
Now release the mouse button. The blocks are connected. You can also connect the line to the block by releasing the mouse button while the pointer is over the block. If you do, the line is connected to the input port closest to the cursor's position.



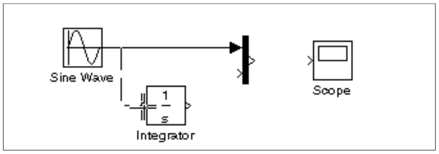
If you look again at the model at the beginning of this section (see "Building a Model" on page 1-9), you'll notice that most of the lines connect output ports of blocks to input ports of other blocks. However, one line connects a *line* to the input port of another block. This line, called a *branch 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:

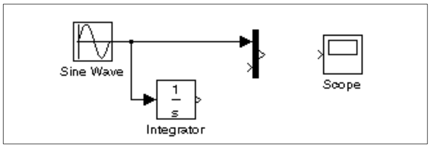
1 First, position the pointer *on the line* between the Sine Wave and the Mux block.



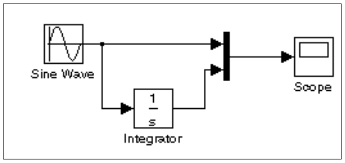
2 Press and hold down the **Ctrl** key (or click the right mouse button). Press the mouse button, then drag the pointer to the Integrator block's input port or over the Integrator block itself.



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

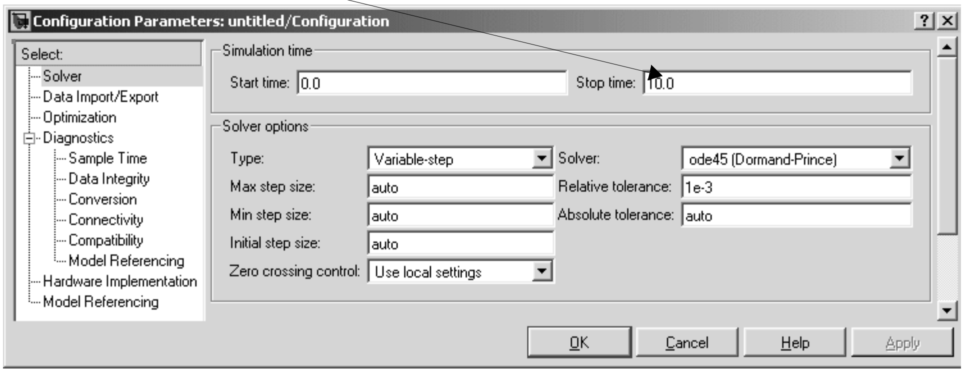


Finish making block connections. When you're done, your model should look something like this.



Now set up Simulink to run the simulation for 10 seconds. First, open the **Configuration Parameters** dialog box by choosing **Configuration 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



Configuration Parameters: untitled/Configuration

Simulation time

Start time: 0.0

Stop time: 10.0

Solver options

Type: Variable-step

Solver: ode45 (Dormand-Prince)

Max step size: auto

Relative tolerance: 1e-3

Min step size: auto

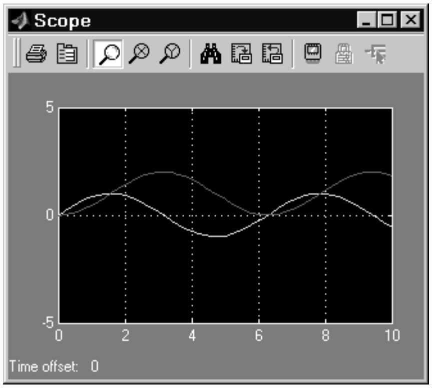
Absolute tolerance: auto

Initial step size: auto

Zero crossing control: Use local settings

Close the **Configuration Parameters** dialog box by clicking the **OK** button. Simulink applies the parameters and closes the dialog box.

Now double-click the Scope block to open its display window. Finally, choose **Start** from the **Simulation** menu and watch the simulation output on the Scope.



The simulation stops when it reaches the stop time specified in the **Configuration Parameters** dialog box or when you choose **Stop** from the **Simulation** menu or click the **Stop** button on the model window's toolbar (Windows only).

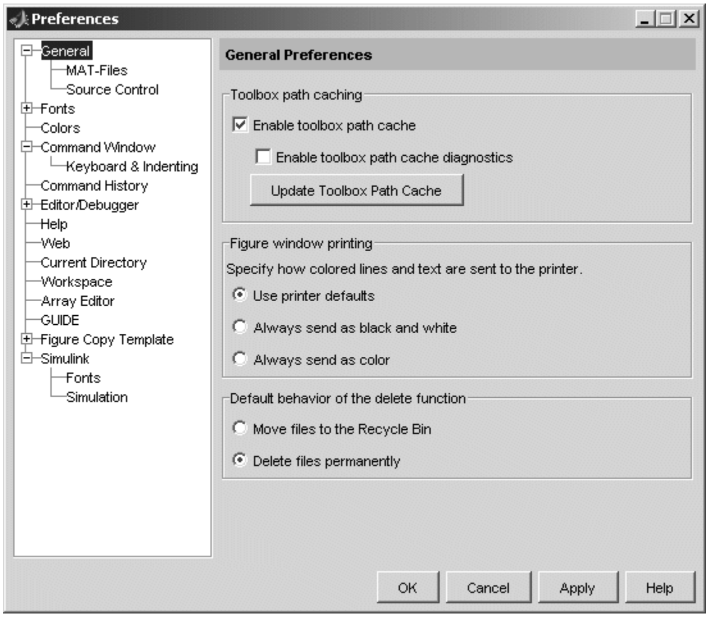
To save this model, choose **Save** from the **File** menu and enter a filename and location. That file contains the description of the model.

To terminate Simulink and MATLAB, choose **Exit MATLAB** (on a Microsoft Windows system) or **Quit MATLAB** (on a UNIX system). You can also enter `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 some Simulink options. To display the **Preferences** dialog box, select **Preferences** from the Simulink **File** menu.



General Preferences

Toolbox path caching

☑ Enable toolbox path cache

☐ Enable toolbox path cache diagnostics

Figure window printing

Specify how colored lines and text are sent to the printer.

● Use printer defaults

○ Always send as black and white

○ Always send as color

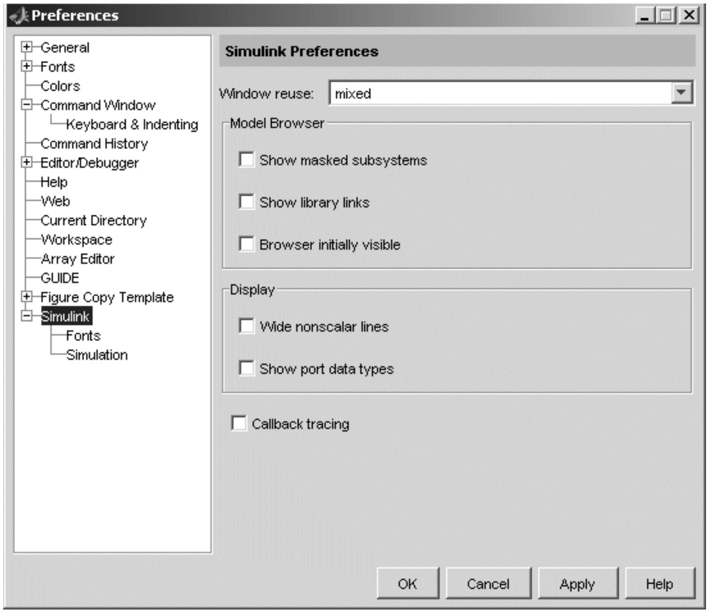
Default behavior of the delete function

○ Move files to the Recycle Bin

● Delete files permanently

Miscellaneous Preferences

Selecting Simulink in the left hand pane of the preferences dialog box displays a **Simulink Preferences** pane on the right side of the dialog box.



Simulink Preferences

Window reuse: mixed

Model Browser

☐ Show masked subsystems

☐ Show library links

☐ Browser initially visible

Display

☐ Wide nonscalar lines

☐ Show port data types

☐ Callback tracing

This pane 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 subsystems (see "Window Reuse" on page 4-23).

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 9-22).

Display

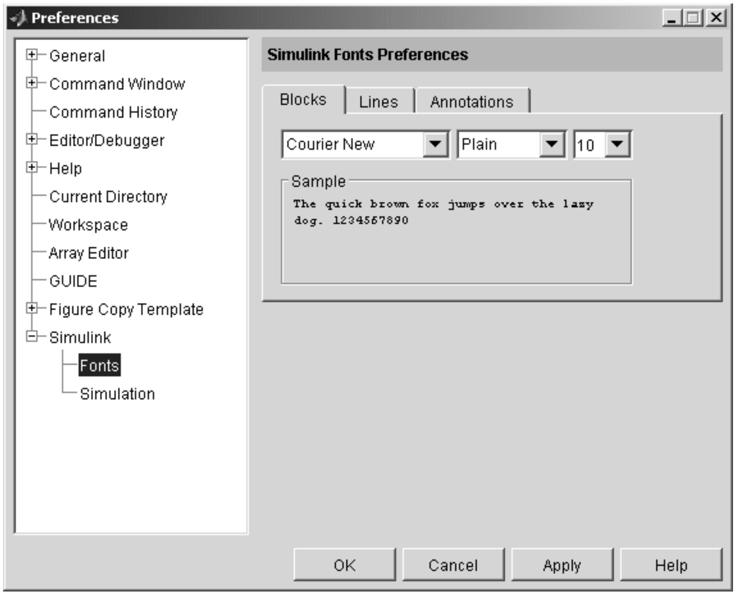
Specifies whether to use thick lines to display nonscalar connections between blocks and whether to display port data types on the block diagram (see "Working with Signal Groups" on page 6-41).

Callback tracing

Specifies whether to display the model callbacks that Simulink invokes when simulating a model (see “Using Callback Routines” on page 4-90).

Font Preferences

Selecting the Fonts subnode of the Simulink node in the left side of the dialog box displays a stack of tabbed panes on the right side of the dialog box.



Simulink Fonts Preferences

Courier New | Plain | 10

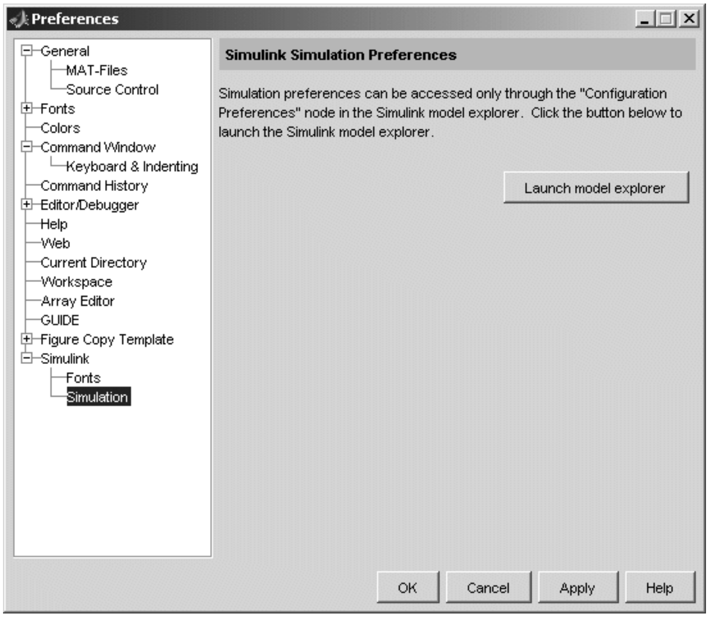
Sample

The quick brown fox jumps over the lazy dog. 1234567890

The panes allow you to specify your preferred fonts for block and line labels and model annotations, respectively.

Simulation Preferences

Selecting the Simulation node beneath the Simulink node in the left side of the dialog box displays a button to start the Model Explorer (see “The Model Explorer” on page 9-2).



Simulink Simulation Preferences

Simulation preferences can be accessed only through the "Configuration Preferences" node in the Simulink model explorer. Click the button below to launch the Simulink model explorer.

Use the Model Explorer to set your simulation preferences.

2

How Simulink Works

The following sections explain how Simulink models and simulates dynamic systems. This information can be helpful in creating models and interpreting simulation results.

Introduction

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

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

Modeling Dynamic Systems

A Simulink block diagram model is a graphical representation of a mathematical model of a dynamic system. A mathematical model of a dynamic system is described by a set of equations. The mathematical equations described by a block diagram model are known as algebraic, differential, and/or difference equations.

Block Diagram Semantics

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

Simulink extends these classic block diagram models by introducing the notion of two classes of blocks, nonvirtual block and virtual blocks. Nonvirtual blocks represent elementary systems. A virtual block is provided for graphical organizational convenience and plays no role in the definition of the system of equations described by the block diagram model. Examples of virtual blocks are the Bus Creator and Bus Selector which are used to reduce block diagram clutter by managing groups of signals as a “bundle.” You can use virtual blocks to improve the readability of your models.

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

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

To summarize the meaning of time-based block diagrams:

- Simulink block diagrams define time-based relationships between signals and state variables. The solution of a block diagram is obtained by evaluating these relationships over time, where time starts at a user specified “start time” and ends at a user specified “stop time.” Each evaluation of these relationships is referred to as a time step.

- Signals represent quantities that change over time and are defined for all points in time between the block diagram’s start and stop time.
- The relationships between signals and state variables are defined by a set of equations represented by blocks. Each block consists of a set of equations (block methods). These equations define a relationship between the input signals, output signals and the state variables. Inherent in the definition of a equation is the notion of parameters, which are the coefficients found within the equation.

Creating Models

Simulink provides a graphical editor that allows you to create and connect instances of block types (see Chapter 4, “Creating a Model”) selected from libraries of block types (see the “Block Reference” in the online Simulink Help) via a library browser. Simulink provides libraries of blocks representing elementary systems that can be used a building blocks. The blocks supplied with Simulink are called built-in blocks. Simulink users can also create their own block types and use the Simulink editor to create instances of them in a diagram. Customer-defined blocks are called custom blocks.

Time

Time is an inherit component of block diagrams in that the results of a block diagram simulation change with time. Put another way, a block diagram represents the instantaneous behavior of a dynamic system. Determining a system’s behavior over time thus entails repeatedly executing the model at intervals, called time steps, from the start of the time span to the end of the time span. Simulink refers to the repeated execution of a model at successive time steps as simulating the system that the model represents. It is possible to simulate a system manually, i.e., to execute its model manually. However, this is unnecessary as the Simulink engine performs this task automatically on command from the user.

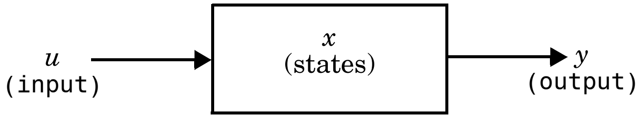
States

Typically the current values of some system, and hence model, outputs are functions of the previous values of temporal variables. Such variables are called states. Computing a model's outputs from a block diagram hence entails saving the value of states at the current time step for use in computing the outputs at a subsequent time step. Simulink performs this task during simulation for models that define states.

Two types of states can occur in a Simulink model: discrete and continuous states. A continuous state changes continuously. Examples of continuous states are the position and speed of a car. A discrete state is an approximation of a continuous state where the state is updated (recomputed) using finite (periodic or aperiodic) intervals. An example of a discrete state would be the position of a car shown on a digital odometer where it is updated every second as opposed to continuously. In the limit, as the discrete state time interval approaches zero, a discrete state becomes equivalent to a continuous state.

Blocks implicitly define a model's states. In particular, a block that needs some or all of its previous outputs to compute its current outputs implicitly defines a set of states that need to be saved between time steps. Such a block is said to have states.

The following is a graphical representation of a block that has states.



Blocks that define continuous states include the following standard Simulink blocks:

- Integrator
- State-Space
- Transfer Fcn
- Zero-Pole

The total number of a model's states is the sum of all the states defined by all its blocks. Determining the number of states in a diagram requires parsing the diagram to determine the types of blocks that it contains and then aggregating the number of states defined by each instance of a block type that defines states. Simulink performs this task during the Compilation phase of a simulation.

Continuous States

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

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

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

Discrete States

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

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

Modeling Hybrid Systems

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

Block Parameters

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

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

Each time you change parameters, you change the meaning of the model. Simulink lets you modify the parameter values during execution of your model. For example, you can pause simulation, change parameter values, and continue simulation. It should be pointed out that parameter changes do not immediately occur, but are queued up and then applied at the start of the next time step during model execution. Returning to our example of the constant block, the function it defines is $signal(t) = Cons\tan tValue$ for all time. If we were to allow the constant value to be changed immediately, then the solution at the point in time at which the change occurred would be invalid, thus we must queue the change for processing on the next time step.

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 in your model are nontunable except for those that you specify. This can speed up execution of large models and enable generation of faster code from your model. See "Model Parameter Configuration Dialog Box" on page 10-47 for more information.

Block Sample Times

Every Simulink block is considered to have a sample time, even continuous blocks (e.g., blocks that define continuous states, such as the Integrator block) and blocks that do not define states, such as the Gain block. Discrete blocks allows you to specify their sample times via a Sample Time parameter. Continuous blocks are considered to have an infinitesimal sample time called a continuous sample time. A block that is neither discrete or continuous is said to have an implicit sample time that it inherits from its inputs. The implicit sample time is continuous if any of the block's inputs are continuous. Otherwise, the implicit sample time is discrete. An implicit discrete sample time is equal to the shortest input sample time if all the input sample times are integer multiples of the shortest time. Otherwise, the implicit 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 (continuous), magenta (constant), yellow (hybrid), red (fastest discrete), and so on. See "Mixed Continuous and Discrete Systems" on page 2-40 for more information.

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 the Simulink Subsystem block, and provide the block with a parameter dialog, using the Simulink block mask facility. To create a block programmatically, you create an M-file or a MEX-file that contains the block's system functions (see *Writing S-Functions* in the online Help for Simulink). The resulting file is called an S-function. You then associate the S-function with instances of the Simulink S-Function block in your model. You can add a parameter dialog to your S-Function block by wrapping it in a Subsystem block and adding the parameter dialog to the Subsystem block.

Systems and Subsystems

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

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

Flattening the Model Hierarchy

While preparing a model for execution, Simulink generates internal “systems” that are collections of block methods (equations) that are evaluated together. The semantics of time-based block diagrams doesn’t require creation of these systems. Simulink creates these internal systems as a means to manage the execution of the model. Roughly speaking, there will be one system for the top-level block diagram window which is referred to as the root system, and several lower-level system derived from the nonvirtual subsystem and other elements within the block diagram. You will see these systems within the Simulink Debugger. The act of creating these “internal” systems is often referred to as flattening the model hierarchy.

Conditionally Executed Subsystems

You can create conditionally executed subsystems that are executed only when a transition occurs on a triggering, function-call, action, or enabling input (see “Creating Conditionally Executed Subsystems” on page 4-26).

Conditionally executed subsystems are atomic. Unconditionally executed subsystems are virtual by default. You can, however, designate an unconditionally executed subsystem as atomic. This is useful if you need to ensure that the equations defined by a subsystem are evaluated “together” as a unit.

Signals

Simulink uses the term *signal* to refer to a time varying quantity that has values at all points in time. 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.

On the block diagram, you will find that the signals are represented with lines that have an arrow head. The source of the signal corresponds to the block that writes to the signal during evaluation of its block methods (equations). The destinations of the signal are blocks that read the signal during the evaluation of its block methods (equations). A good analogy of the meaning of a signal is to consider a classroom. The teacher is the one responsible for writing on the white board and the students read what is written on the white board when they choose to. This is also true of Simulink signals, a reader of the signal (a block method) can choose to read the signal as frequently or infrequently as so desired.

Block Methods

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

Method Types

Simulink assigns names to the types of functions performed by block methods. Common method types include:

- Outputs

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

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

  Computes the derivatives of the block's continuous states at the current time step, given the block's inputs and the values of the states at the previous time step.

Method Naming Convention

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

```
BlockType.MethodType
```

For example, Simulink refers to the method that computes the outputs of a Gain block as

```
Gain.Outputs
```

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

```
g1.Outputs
```

Model Methods

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

Simulating Dynamic Systems

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.

A Simulink component called the Simulink Engine responds to a Start command, performing the following steps.

Model Compilation

First, the Simulink engine invokes the model compiler. The model compiler converts the model to an executable form, a process called compilation. In particular, the compiler

- Evaluates the model's block parameter expressions to determine their values.
- Determines signal attributes, e.g., name, data type, numeric type, and dimensionality, not explicitly specified by the model and checks that each block can accept the signals connected to its inputs.
- 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.
- Performs block reduction optimizations.
- Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain (see "Solvers" on page 2-17).
- Sorts the blocks into the order in which they need to be executed during the execution phase (see "Solvers" on page 2-17).
- Determines the sample times of all blocks in the model whose sample times you did not explicitly specify.

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. Simulink sorts the blocks into the correct order during the model initialization phase.

Direct-Feedthrough Ports. In order to create a valid update ordering, Simulink categorizes a block's input ports according to the relationship of outputs to inputs. An input port whose current value determines the current value of one of the block's outputs is called a *direct-feedthrough* port. Examples of blocks that have direct-feedthrough ports include the Gain, Product, and Sum blocks. Examples of blocks that have non-direct-feedthrough inputs include the Integrator block (its output is a function purely of its state), the Constant block (it does not have an input), and the Memory block (its output is dependent on its input 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 blocks whose direct-feedthrough ports it drives.

  This rule ensures that the direct-feedthrough inputs to blocks will be valid when the blocks are updated.
- Blocks that do not have direct feedthrough inputs can be updated in any order as long as they are updated before any blocks whose direct-feedthrough inputs they drive.

  Putting all blocks that do not have direct-feedthrough ports at the head of the update list in any order satisfies this rule. It thus allows Simulink to ignore these blocks during the sorting process.

The result of applying these rules is an update list in which blocks without direct feedthrough ports appear at the head of the list in no particular order followed by blocks with direct-feedthrough ports 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 a direct-feedthrough output of a block is connected directly or indirectly to the corresponding direct-feedthrough input of the block. Such loops seemingly create a deadlock condition, because Simulink needs the value of the direct-feedthrough input to compute the 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 ports 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 2-24.

Link Phase

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

Method Execution Lists

In the Link phase, the Simulink engine also creates method execution lists. These lists list the most efficient order in which to execute a model’s block methods to compute its outputs. Simulink uses the sorted lists generated during the Compile phase to construct the method execution lists.

Block Priorities

Simulink allows you to assign update priorities to blocks (see “Assigning Block Priorities” on page 5-20). Simulink executes the output methods of higher priority blocks before those of lower priority blocks. Simulink honors the priorities only if they are consistent with its block sorting rules.

Simulation Loop Phase

The simulation now enters the simulation loop phase. In this phase, the Simulink engine successively computes the states and outputs of the system at intervals from the simulation start time to the finish time, using information provided by the model. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called the step size. The step size depends on the type of solver (see “Solvers” on page 2-17) used to compute the system’s continuous states, the system’s fundamental sample time (see “Modeling and Simulating Discrete Systems” on page 2-31), and whether the system’s continuous states have discontinuities (see “Zero-Crossing Detection” on page 2-19).

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

At the start of the simulation, the model specifies the initial states and outputs of the system to be simulated. At each step, 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.

Loop Iteration

At each time step, the Simulink Engine

1. Computes the model's outputs.

   The Simulink Engine initiates this step by invoking the Simulink model Outputs method.The model Outputs method in turn invokes the model system Outputs method, which invokes the Outputs methods of the blocks that the model contains in the order specified by the Outputs method execution lists generated in the Link phase of the simulation (see "Solvers" on page 2-17).

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

2. Computes the model's states.

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

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

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

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

   Simulink uses a technique called zero-crossing detection to detect discontinuities in continuous states. See "Zero-Crossing Detection" on page 2-19 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.

Solvers

Simulink simulates a dynamic system by computing its states at successive time steps over a specified time span, using information provided by the model. The process of computing the successive states of a system from its model is known as solving the model. No single method of solving a model suffices for all systems. Accordingly, Simulink provides a set of programs, known as *solvers*, that each embody a particular approach to solving a model. The **Configuration Parameters** dialog box allows you to choose the solver most suitable for your model (see "Choosing a Solver Type" on page 10-7).

Fixed-Step Solvers Versus Variable-Step Solvers

Simulink solvers fall into two basic categories: fixed-step and variable-step.

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

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

Continuous Versus Discrete Solvers

Simulink provides both continuous and discrete solvers.

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

Mathematicians have developed a wide variety of numerical integration techniques for solving the ordinary differential equations (ODEs) that represent the continuous states of dynamic systems. Simulink provides an extensive set of fixed-step and variable-step continuous solvers, each implementing a specific ODE solution method (see "Choosing a Solver Type" on page 10-7).

Discrete solvers exist primarily to solve purely discrete models. They compute the next simulation time step for a model and nothing else. They do not compute continuous states and they rely on the model's blocks to update the model's discrete states.

Note You can use a continuous solver, but not a discrete solver, to solve a model that contains both continuous and discrete states. This is because a discrete solver does not handle continuous states. If you select a discrete solver for a continuous model, Simulink disregards your selection and uses a continuous solver instead when solving the model.

Simulink provides two discrete solvers, a fixed-step discrete solver and a variable-step discrete solver. The fixed-step solver by default chooses a step size and hence simulation rate fast enough to track state changes in the fastest block in your model. The variable-step solver adjusts the simulation step size to keep pace with the actual rate of discrete state changes in your model. This can avoid unnecessary steps and hence shorten simulation time for multirate models (see “Determining Step Size for Discrete Systems” on page 2-36 for more information).

Minor Time Steps

Some continuous solvers subdivide the simulation time span into major and minor time steps, where a minor time step represents a subdivision of the major time step. The solver produces a result at each major time step. It uses results at the minor time steps to improve the accuracy of the result at the major time step.

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 discontinuity in its position. Because discontinuities 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 might 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 2-23 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.

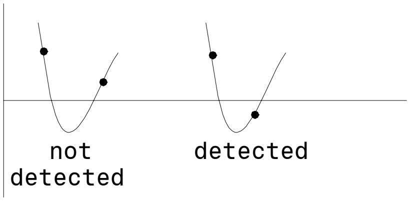
Zero-crossing signals have a direction attribute, which can have these values:

- *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 steps over the crossing without detecting it.

The following figure shows a signal that crosses zero. In the first instance, the integrator steps over the event. 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 “Maximum order” on page 10-35.

Note Using the `Refine output` option (see “Output options” on page 10-42) does 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 massless 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 use the **Zero crossing control** option on the **Solver** pane of the **Configuration Parameters** dialog box (see “Zero crossing control” on page 10-33) to disable zero-crossing detection. Although disabling zero-crossing detection can 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

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

Block	Description of Zero Crossing
Abs	One: to detect when the input signal crosses zero in either the rising or falling direction.
Backlash	Two: one to detect when the upper threshold is engaged, and one to detect when the lower threshold is engaged.
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.
Integrator	If the reset port is present, to detect when a reset occurs. If the output is limited, there are three zero crossings: one to detect when the upper saturation limit is reached, one to detect when the lower saturation limit is reached, and one to detect when saturation is left.
MinMax	One: for each element of the output vector, to detect when an input signal is the new minimum or maximum.
Relay	One: if the relay is off, to detect the switch on point. If the relay is on, to detect the switch off point.
Relational Operator	One: to detect when the output changes.
Saturation	Two: one to detect when the upper limit is reached or left, and one to detect when the lower limit is reached or left.
Sign	One: to detect when the input crosses through zero.
Step	One: to detect the step time.

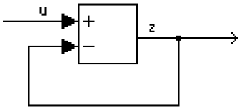
Block	Description of Zero Crossing (Continued)
Subsystem	For conditionally executed subsystems: one for the enable port if present, and one for the trigger port, if present.
Switch	One: to detect when the switch condition occurs.

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 as follows:

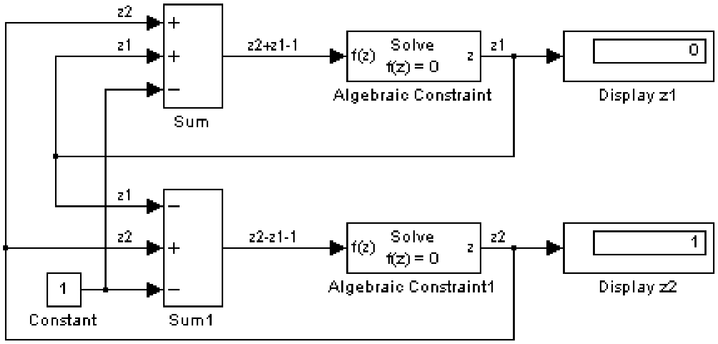
- The Math Function block
- The Gain block
- The Integrator block's initial condition ports
- The Product block
- The State-Space block when there is a nonzero D matrix
- The Sum block
- The Transfer Fcn block when the numerator and denominator are of the same order
- The Zero-Pole block when there are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

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



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

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



The Algebraic Constraint block is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal *F(z)* to zero and outputs an algebraic state *z*. This block outputs the value necessary to produce a zero at the input. The output must affect the input through some 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

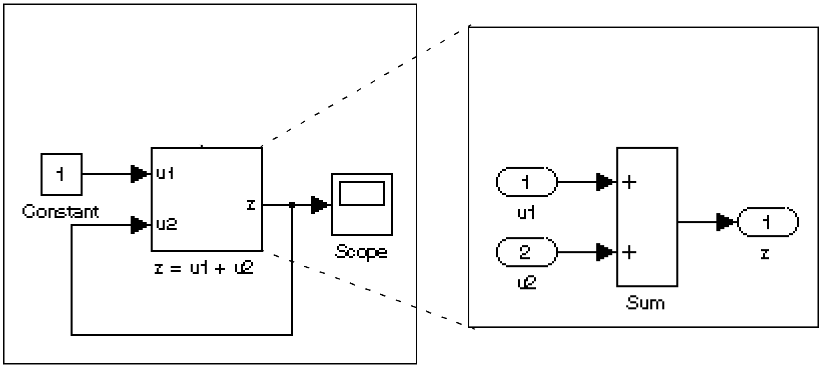
$$z2 + z1 - 1 = 0$$
$$z2 - z1 - 1 = 0$$

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

When a model contains an algebraic loop, 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



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.

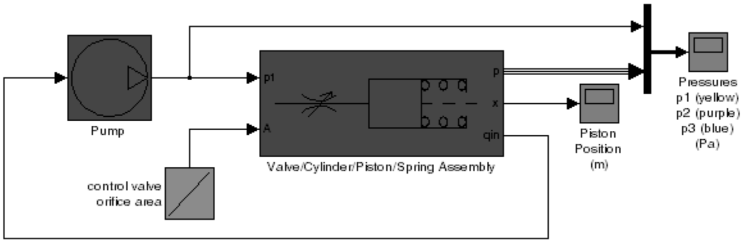
Highlighting Algebraic Loops

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

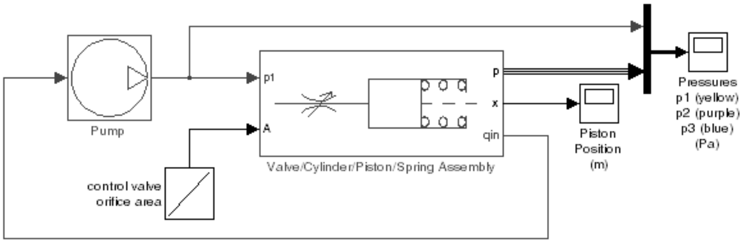
To cause Simulink to highlight algebraic loops that it detects when updating or simulating a model, set the `Algebraic loop` diagnostic on the **Diagnostics** pane of the **Configuration Parameters** dialog box to `Error` (see "The

Configuration Parameters Dialog Box" on page 10-30 for more information). This causes Simulink to display an error dialog (the Diagnostics Viewer) and recolor portions of the diagram that represent the algebraic loops that it detects. Simulink uses red to color the blocks and lines that constitute the loops. Closing the error dialog restores the diagram to its original colors.

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



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



In this example, Simulink has colored the algebraic loop red, making it stand out from the rest of the diagram.

Eliminating Algebraic Loops

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

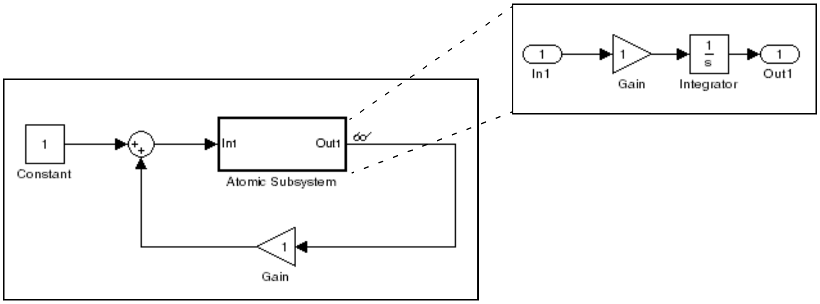
- Atomic Subsystem
- Enabled Subsystem
- Model

To enable automatic algebraic loop elimination for a loop involving a particular instance of an Atomic Subsystem or Enabled Subsystem block, select the **Minimize algebraic loop occurrences** parameter on the block's parameters dialog box. To enable algebraic loop elimination for a loop involving a Model block, check the **Minimize algebraic loop occurrences** parameter on the **Model Referencing** configuration parameters dialog (see "Model Referencing Pane" on page 10-67) of the model referenced by the Model block. If a loop includes more than one instance of these blocks, you should enable algebraic loop elimination for all of them, including nested blocks.

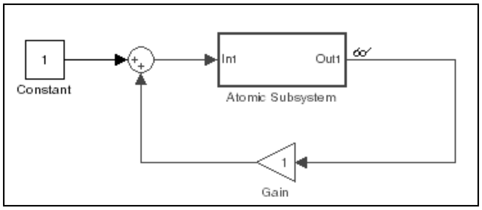
The Simulink **Minimize algebraic loop** solver diagnostic allows you to specify the action Simulink should take, for example, display a warning message, if it is unable to eliminate an algebraic loop involving a block for which algebraic loop elimination is enabled. See "The Diagnostics Pane" on page 10-48 for more information.

Algebraic loop minimization is off by default because it is incompatible with conditional input branch optimization in Simulink (see "The Optimization Pane" on page 10-43) and with single output/update function optimization in Real-Time Workshop®. If you need these optimizations for an atomic or enabled subsystem or referenced model involved in an algebraic loop, you must eliminate the algebraic loop yourself.

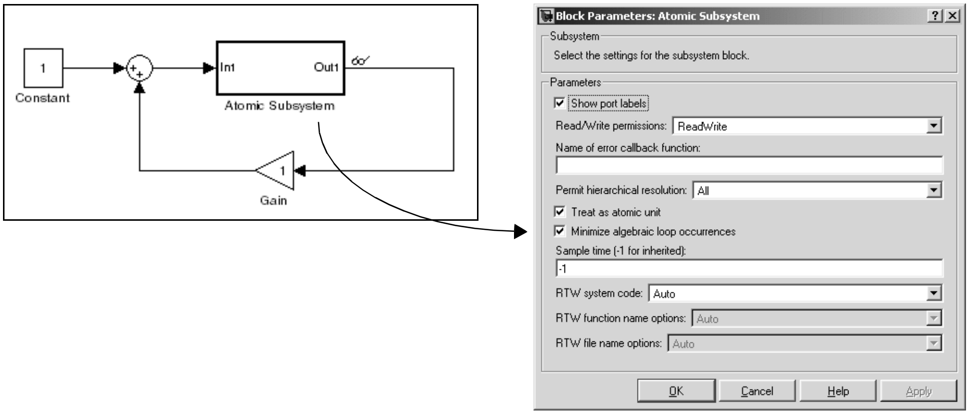
As an example of the ability of Simulink to eliminate algebraic loops, consider the following model.



Simulating this model with the solver's Algebraic Loop diagnostic set to error (see "The Diagnostics Pane" on page 10-48) reveals that this model contains an algebraic loop involving its atomic subsystem.



Checking the atomic subsystem's **Minimize algebraic loop occurrences** parameter causes Simulink to eliminate the algebraic loop from the compiled version of the model.



Block Parameters: Atomic Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

☑ Show port labels

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

☑ Treat as atomic unit

☑ Minimize algebraic loop occurrences

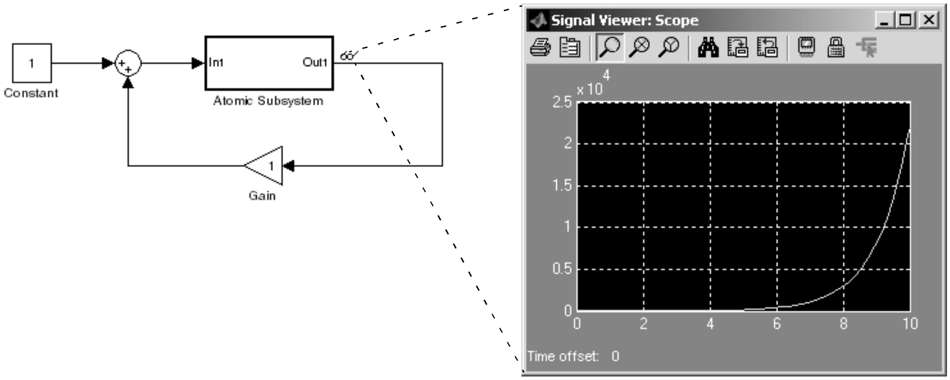
Sample time (-1 for inherited): -1

RTW system code: Auto

RTW function name options: Auto

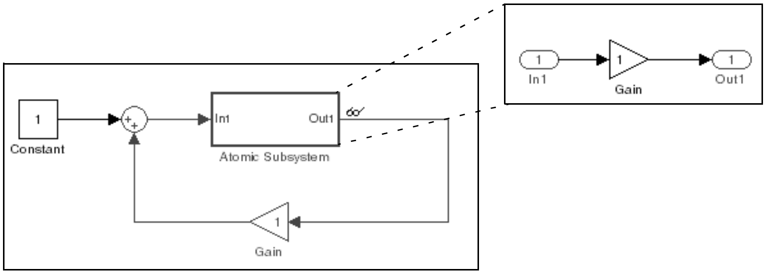
RTW file name options: Auto

As a result, the model now simulates without error.



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

If you remove the Integrator block from the atomic subsystem, Simulink is unable to eliminate the algebraic loop. Hence, attempting to simulate the model results in an error.



Modeling and Simulating Discrete Systems

Simulink has the ability to simulate discrete (sampled data) systems, including systems whose components operate at different rates (*multirate systems*) and systems that mix discrete and continuous components (*hybrid systems*). This capability stems from two key Simulink features:

- SampleTime block parameter

  Some Simulink blocks have a SampleTime parameter that you can use to specify the block's sample time, i.e., the rate at which it executes during simulation. All blocks have either an explicit or implicit sample time parameter. Continuous blocks are examples of blocks that have an implicit (continuous) sample time. It is possible for a block to have multiple sample times as provided with blocksets such as the Signal Processing Blockset or created by a user using the S-Function block.

- Sample-time inheritance

  Most standard Simulink blocks can inherit their sample time from the blocks connected to their inputs. Exceptions include blocks in the Continuous library and blocks that do not have inputs (e.g., blocks from the Sources library). In some cases, source blocks can inherit the sample time of the block connected to its input.

The ability to specify sample times on a block-by-block basis, either directly through the SampleTime parameter or indirectly through inheritance, enables you to model systems containing discrete components operating at different rates and hybrid systems containing discrete and continuous components.

Specifying Sample Time

Simulink allows you to specify the sample time of any block that has a `SampleTime` parameter. You can use the block's parameter dialog box to set this parameter. You do this by entering the sample time in the **Sample time** field on the dialog box. You can enter either the sample time alone or a vector whose first element is the sample time and whose second element is an offset: `[`T_s`, `T_o`]`. Various values of the sample time and offset have special meanings.

The following table summarizes valid values for this parameter and how Simulink interprets them to determine a block's sample time.

Sample Time	Usage
`[`T_s`, `T_o`]` $0 > T_s < T_{sim}$ $\lvert T_o \rvert < T_p$	Specifies that updates occur at simulation times $t_n = n * T_s + \lvert T_o \rvert$ where `n` is an integer in the range `1..`T_{sim}/T_s and T_{sim} is the length of the simulation. Blocks that have a sample time greater than 0 are said to have a *discrete sample time*. The offset allows you to specify that Simulink update the block later in the sample interval than other blocks operating at the same rate.
`[0, 0], 0`	Specifies that updates occur at every major and minor time step. A block that has a sample time of 0 is said to have a *continuous sample time*.
`[0, 1]`	Specifies that updates occur only at major time steps, skipping minor time steps (see "Minor Time Steps" on page 2-19). This setting avoids unnecessary computations for blocks whose sample time cannot change between major time steps. The sample time of a block that executes only at major time steps is said to be *fixed in minor time step*.

Sample Time	Usage
`[-1, 0]`, -1	If the block is not in a triggered subsystem, this setting specifies that the block inherits its sample time from the block connected to its input (inheritance) or, in some cases, from the block connected to its output (back inheritance). If the block is in a triggered subsystem, you must set the SampleTime parameter to this setting. Note that specifying sample-time inheritance for a source block can cause Simulink to assign an inappropriate sample time to the block if the source drives more than one block. For this reason, you should avoid specifying sample-time inheritance for source blocks. If you do, Simulink displays a warning message when you update or simulate the model.
inf	The meaning of this sample time depends on whether the active model configuration's inline parameters optimization (see "Inline parameters" on page 10-44) is enabled. If the inline parameters optimization is enabled, `inf` signifies that the block's output can never change (see "Invariant Constants" on page 2-39). This speeds up simulation and the generated code by eliminating the need to recompute the block's output at each time step. If the inline parameters optimization is disabled or the block with `inf` sample time drives an output port of a conditionally executed subsystem, Simulink treats `inf` as `-1`, i.e., as inherited sample time. This allows you to tune the block's parameters during simulation.

Changing a Block's Sample Time

You cannot change the SampleTime parameter 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.

Compiled Sample Time

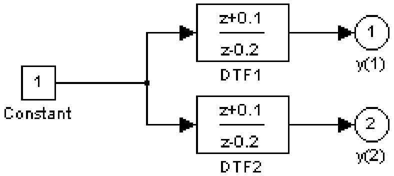
During the compilation phase of a simulation, Simulink determines the sample time of the block from its SampleTime parameter (if it has a SampleTime parameter), sample-time inheritance, or block type (Continuous blocks always have a continuous sample time). It is this compiled sample time that determines the sample rate of a block during simulation. You can determine the compiled sample time of any block in a model by first updating the model and then getting the block's CompiledSampleTime parameter, using the `get_param` command.

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 with both discrete and continuous blocks. For example, consider this simple multirate discrete model.

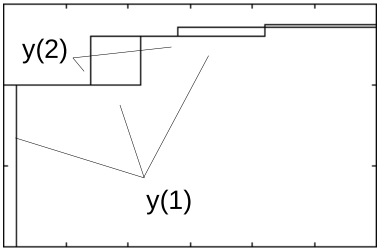


For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to `[1 0.1]`, which gives it an offset of `0.1`. The DTF2 Discrete Transfer Fcn block's **Sample time** is set to `0.7`, with no offset.

Starting the simulation and plotting the outputs using the `stairs` function

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

produces this plot



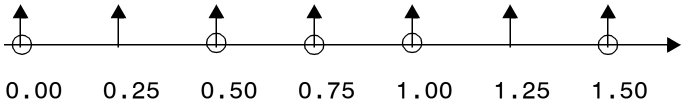
For the DTF1 block, which has an offset of `0.1`, there is no output until `t = 0.1`. Because the initial conditions of the transfer functions are zero, the output of DTF1, y(1), is zero before this time.

Determining Step Size for Discrete Systems

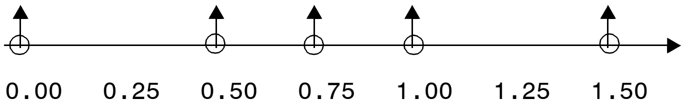
Simulating a discrete system requires that the simulator take a simulation step at every *sample time hit*, that is, at integer multiples of the system's shortest sample time. Otherwise, the simulator might miss key transitions in the system's states. Simulink avoids this by choosing a simulation step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

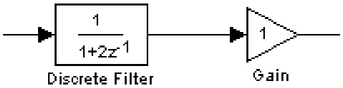


Variable-Step Solver

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 Real-Time Workshop®).

Sample Time Propagation

When updating a model's diagram, for example, at the beginning of a simulation, Simulink uses a process called sample time propagation to determine the sample times of blocks that inherit their sample times. The figure below illustrates a Discrete Filter block with a sample time of `Ts` driving a Gain block.

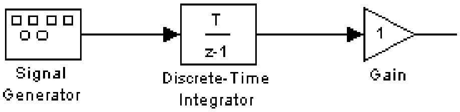


Because the Gain block's output is simply the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to that of the filter's sample rate. This is the fundamental mechanism behind sample time propagation in Simulink.

Simulink assigns an inherited sample time to a block based on the sample times of the blocks connected to its inputs. If all the inputs have the same sample time, Simulink assigns that sample time to the block. If the 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.

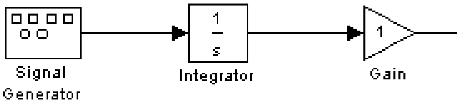
Note A Model block can inherit its sample time from its inputs only if the inputs and outputs of the model that it references do not depend on the sample time (see “Model Block Sample Times” on page 4-61 for more information).

Under some circumstances, Simulink also back propagates 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.

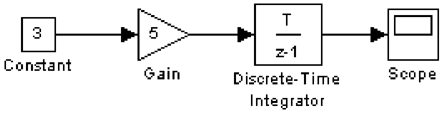
Replacing the Discrete-Time Integrator block with a continuous Integrator block, as shown below, and recoloring the model by choosing **Update diagram** from the **Edit** menu cause the Signal Generator and Gain blocks to change to continuous blocks, as indicated by their being colored black.



Invariant Constants

Simulink by default assigns Constant blocks a sample time of infinity, also referred to as a *constant sample time*. This means that the outputs of any blocks that inherit a constant sample time from a Constant block do 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 the inline parameters option (see "Inline parameters" on page 10-44) 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.

Note Simulink displays an error if you connect a Constant, Model, or S-Function block with constant sample time to the output port of a conditionally executed subsystem. To avoid the error, either change the sample time of the block to a nonconstant sample time or insert a Signal Conversion block between the block with constant sample time and the output port.

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. Because of 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.

3

Simulink Basics

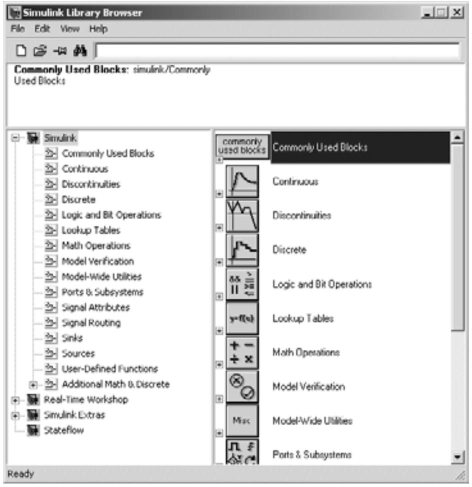
The following sections explain how to perform basic Simulink tasks.

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:

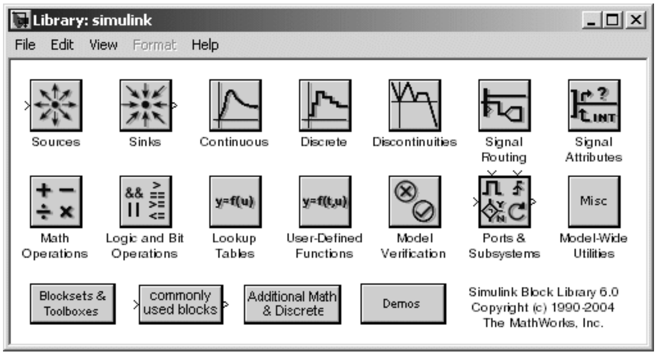
- Click the Simulink icon on the MATLAB toolbar.
- Enter the `simulink` command at the MATLAB prompt.

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



The Library Browser displays a tree-structured view of the Simulink block libraries installed on your system. You can build models by copying blocks from the Library Browser into a model window (see “Editing Blocks” on page 5-4).

On Macintosh or Linux 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.

Opening Models

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 file name 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.

Note If the character encoding of the model to be opened differs from the character encoding of the current MATLAB session, you should change the MATLAB encoding to the model encoding before opening the model, using the `slCharacterEncoding` command.

Avoiding Initial Model Open Delay

You may notice that the first model that you open in a MATLAB session takes longer to open than do subsequent models. This is because to reduce its own startup time and to avoid unnecessary consumption of your system's memory, MATLAB does not load Simulink into memory until the first time you open a Simulink model. You can cause MATLAB to load Simulink at MATLAB startup, and thus avoid the initial model opening delay, using either the `-r` MATLAB command line option or your MATLAB `startup.m` file to run either `load_simulink` (loads Simulink) or `simulink` (loads Simulink and opens the Simulink Library browser) at MATLAB startup. For example, to load Simulink at MATLAB startup on Microsoft Windows systems, create a desktop shortcut with the following target:

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

Similarly, the following command loads Simulink at MATLAB startup on UNIX systems:

```
matlab -r load_simulink
```

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

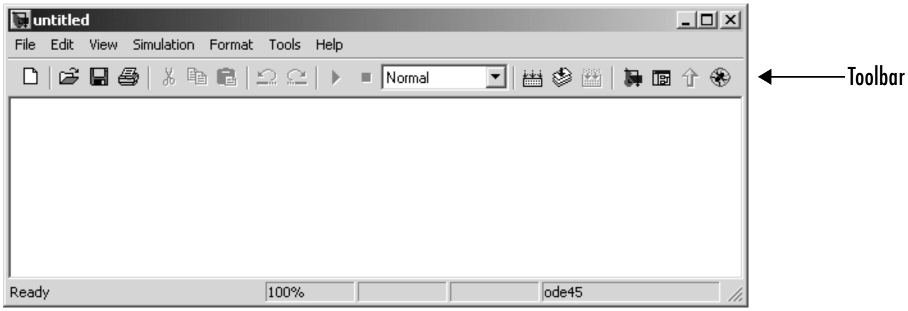
The Simulink menu bar appears near the top of each model window. The menu commands apply to the contents of that window.

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, select the **Toolbar** option on the Simulink **View** menu.



The toolbar contains buttons corresponding to frequently used Simulink commands, such as those for opening, running, and closing models. You can run such commands by clicking the corresponding button. For example, to open a Simulink model, click 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 clearing 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. See Chapter 10, “Running Simulations,” and Chapter 11, “Analyzing Simulation Results,” for more information.

Undoing a Command

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

- Adding, deleting, or moving a block
- Adding, deleting, or moving a line
- Adding, deleting, or moving a model annotation
- Editing a block name
- Creating a subsystem (see “Undoing Subsystem Creation” on page 4-22 for more information)

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

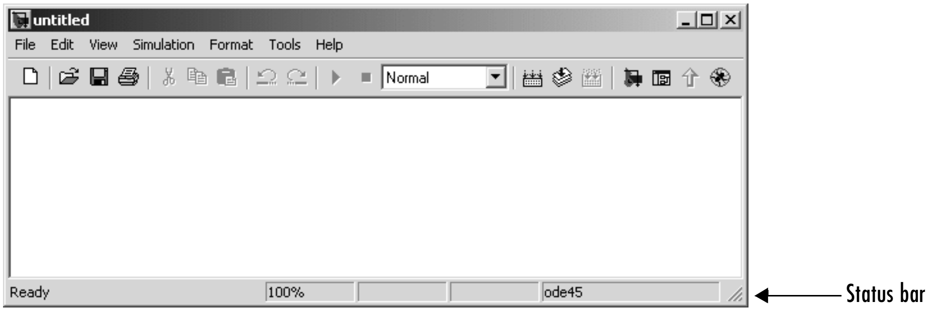
Simulink Windows

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 might find the window sizes too small or too large. If this is the case, resize the window and save the model to preserve the new window dimensions.

Status Bar

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



When a simulation is running, the status bar displays the status of the simulation, including the current simulation time and the name of the current solver. You can display or hide the status bar by selecting or clearing 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.

Panning Block Diagrams

You can use the mouse to pan model diagrams that are too large to fit in the model editor's window. To do this, position the mouse over the diagram and hold down the left mouse button and the P or Q key on the keyboard. Moving the mouse now pans the model diagram in the editor window.

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.

If you are saving a model for the first time, use the **Save** command to provide a name and location for the model file. Model file names must start with a letter and can contain no more than 63 letters, numbers, and underscores. The file name must not be the same as that of a MATLAB command.

If you are saving a model whose model file was previously saved, use the **Save** command to replace the file's contents or the **Save As** command to save the model with a new name or location. You can also use the **Save As** command to save the model in a format compatible with previous releases of Simulink (see "Saving a Model in Earlier Formats" on page 3-9).

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.

Saving a Model in Earlier Formats

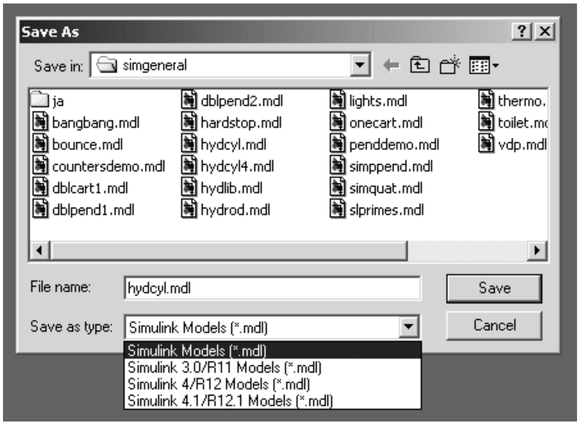
The **Save As** command allows you to save a model created with the latest version of Simulink in formats used by earlier versions of Simulink, including

Simulink 3 (Release 11), Simulink 4 (Release 12), and Simulink 4.1 (Release 12.1). You might want to do this, for example, if you need to make a model available to colleagues who have access only to one of these earlier versions of Simulink.

To save a model in earlier format:

1 Select **Save As** from the Simulink **File** menu.

   Simulink displays the **Save As** dialog box.



Save in: simgeneral

File name: hydcyl.mdl

Save as type: Simulink Models (*.mdl)

2 Select a format from the **Save as type** list on the dialog box.

3 Click the **Save** button.

When saving a model in an earlier version's format, Simulink saves the model in that format regardless of whether the model contains blocks and features that were introduced after that version. If the model does contain blocks or use features that postdate the earlier version, the model might not give correct results when run by the earlier version. For example, matrix and frame signals do not work in Release 11, because Release 11 does not have matrix and frame support. Similarly, models that contain unconditionally executed subsystems marked "Treat as atomic unit" might produce different results in Release 11, because Release 11 does not support unconditionally executed atomic subsystems.

The command converts blocks that postdate the earlier version into empty masked subsystem blocks colored yellow. For example, post-Release 11 blocks include

- Lookup Table (n-D)
- Assertion
- Rate Transition
- PreLookup Index Search
- Interpolation (n-D)
- Direct Lookup Table (n-D)
- Polynomial
- Matrix Concatenation
- Signal Specification
- Bus Creator
- If, WhileIterator, ForIterator, Assignment
- SwitchCase
- Bitwise Logical Operator

Post-Release 11 blocks from Simulink blocksets appear as unlinked blocks.

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).

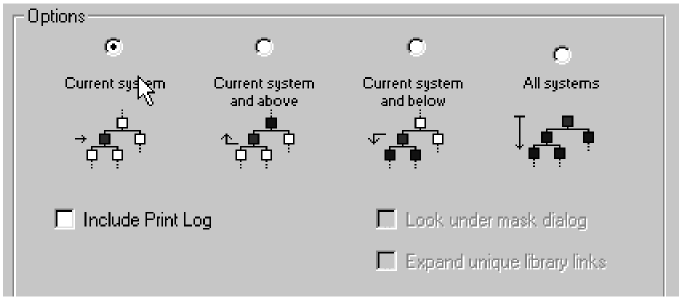
On a Microsoft Windows system, the **Print** menu item prints the block diagram in the current window.

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
- The current system and all systems above it in the model hierarchy
- The current system and all systems below it in the model hierarchy, with the option of looking into the contents of masked and library blocks
- All systems in the model, with the option of looking into the contents of masked and library blocks
- 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.



Options

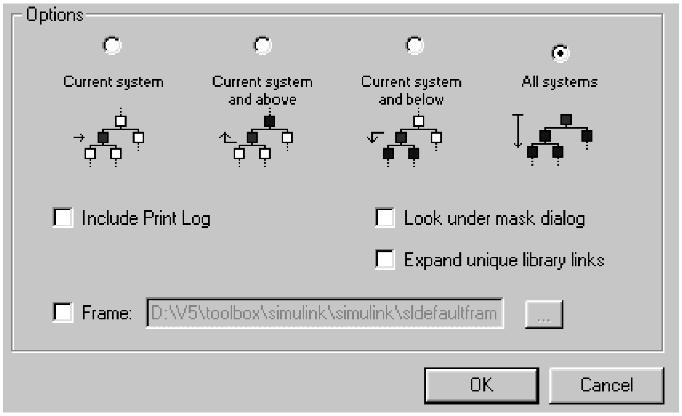
● Current system ○ Current system and above ○ Current system and below ○ All systems

☐ Include Print Log

☐ Look under mask dialog

☐ Expand unique library links

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.



Options

○ Current system

○ Current system and above

○ Current system and below

● All systems

□ Include Print Log

□ Look under mask dialog

□ Expand unique library links

□ Frame: D:\V5\toolbox\simulink\simulink\sldefaultfram

Selecting the **Look Under Mask Dialog** check box prints the contents of masked subsystems when encountered at or below the level of the current block. When you are printing all systems, the top-level system is considered the current block, so 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 “Working with Block Libraries” on page 5-29.

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

Print Command

The format of the `print` command is

```
print -ssys -device filename
```

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

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

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" in the online documentation), using the set_param command. You can set the paper orientation alone, using the MATLAB orient command. On Windows, the **Print** and **Printer Setup** dialog boxes let you set the page type and orientation properties as well.

Positioning and Sizing a Diagram

You can use a model's PaperPositionMode and PaperPosition parameters to position and size the model's diagram on the printed page. The value of the PaperPosition parameter is a vector of form [left bottom width height]. The first two elements specify the bottom left corner of a rectangular area on the page, measured from the page's bottom left corner. The last two elements specify the width and height of the rectangle. When the model's PaperPositionMode is manual, 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.

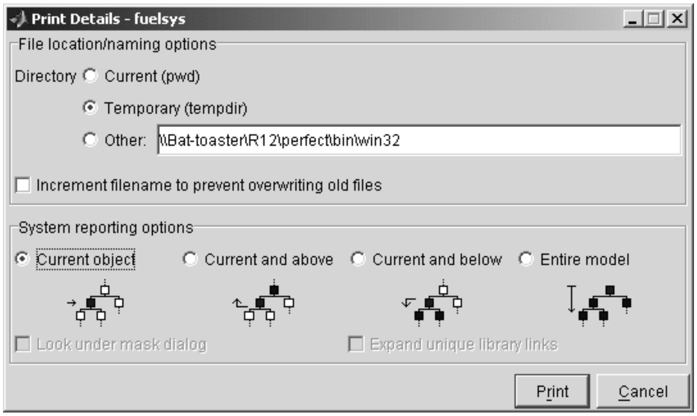
Generating a Model Report

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

To generate a report for the current model:

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

The **Print Details** dialog box appears.



File location/naming options

Directory ○ Current (pwd)

● Temporary (tempdir)

○ Other: \\Bat-toaster\R12\perfect\bin\win32

□ Increment filename to prevent overwriting old files

System reporting options

● Current object ○ Current and above ○ Current and below ○ Entire model

□ Look under mask dialog □ Expand unique library links

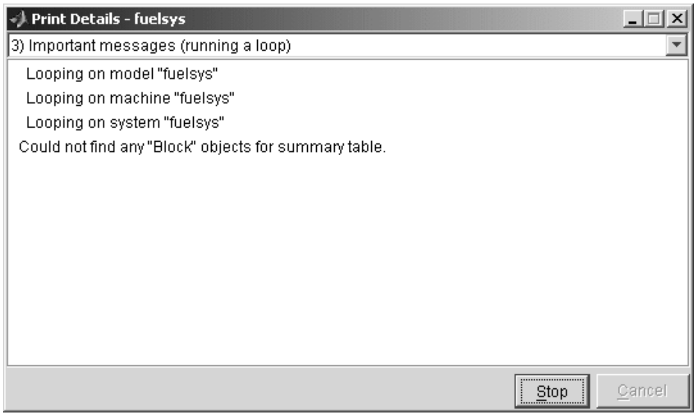
The dialog box allows you to select various report options (see "Model Report Options" on page 3-17).

2 Select the desired report options on the dialog box.

3 Select **Print**.

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

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



```
  Looping on model "fuelsys"
  Looping on machine "fuelsys"
  Looping on system "fuelsys"
Could not find any "Block" objects for summary table.
```

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

Model Report Options

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

Directory

The directory where Simulink stores the HTML report that it generates. The options include your system's temporary directory (the default), your system's current directory, or another directory whose path you specify in the adjacent edit field.

Increment filename to prevent overwriting old files

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

Current object

Include only the currently selected object in the report.

Current and above

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

Current and below

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

Entire model

Include the entire model in the report.

Look under mask dialog

Include the contents of masked subsystems in the report.

Expand unique library links

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

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.

Manipulating Blocks

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

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

Manipulating Lines

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

Task	Microsoft Windows	Macintosh or Linux
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

Manipulating Signal Labels

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

Action	Microsoft Windows	Macintosh or Linux
Create signal label	Double-click line, then enter label	Double-click line, then enter label
Copy signal label	**Ctrl** + drag label	**Ctrl** + drag label
Move signal label	Drag label	Drag label
Edit signal label	Click in label, then edit	Click in label, then edit
Delete signal label	**Shift** + click label, then press **Delete**	**Shift** + click label, then press **Delete**

Manipulating Annotations

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

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

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 Macintosh or Linux system: **Quit MATLAB**

4

Creating a Model

The following sections explain how to create Simulink models.

Creating a New Model

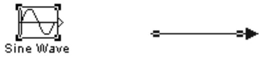
To create a new model, click the **New** button on the Library Browser's toolbar (Windows only) or choose **New** from the library window's **File** menu and select **Model**. You can move the window as you do other windows. Chapter 1, "Getting Started" describes how to build a simple model. "Modeling Equations" on page 8-2 describes how to build systems that model equations.

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



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

Selecting 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.

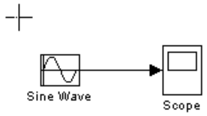
Selecting Multiple Objects One at a Time

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

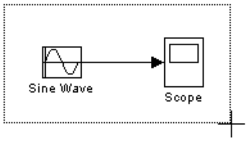
Selecting Multiple Objects Using a Bounding Box

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

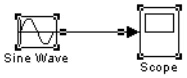
1 Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



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



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



Selecting 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 4-20.

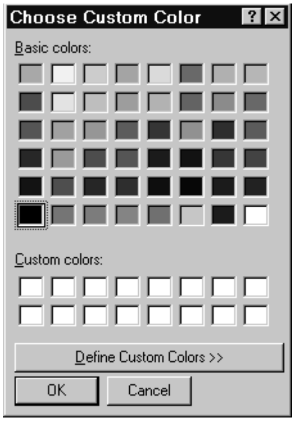
Specifying Block Diagram Colors

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

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

Choosing a Custom Color

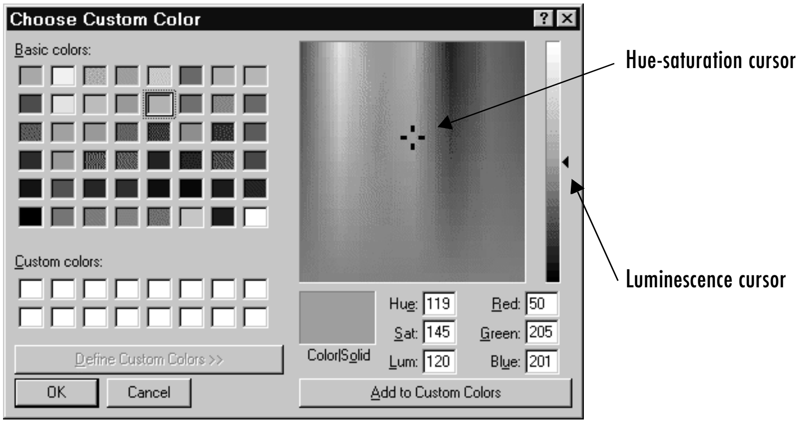
If you choose **Custom**, Simulink displays the Simulink **Choose Custom Color** dialog box.



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

Defining a Custom Color

To define a custom color, click the **Define Custom Colors** button on the **Choose Custom Color** dialog box. The dialog box expands to display a custom color definer.



Hue: 119 Sat: 145 Lum: 120 Red: 50 Green: 205 Blue: 201

The color definer allows you to specify a custom color by

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

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

Specifying Colors Programmatically

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

The following table summarizes the parameters that control block diagram colors.

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

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

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

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

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

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

Displaying Sample Time Colors

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

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

Color	Use
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
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**.

The color that Simulink assigns to each block depends on its sample time relative to other sample times in the model. This means that the same sample time may be assigned different colors in a toplevel model and in the models that it references (see "Referencing Models" on page 4-53). For example, suppose that a model defines three sample times: 1, 2, and 3. Further, suppose that it references a model that defines two sample times: 2 and 3. In this case, blocks operating at the 2 sample rate appear as green in the toplevel model and as red in the referenced model.

It is important to note that Mux and Demux blocks are simply grouping operators; signals passing through them retain their timing information. For this reason, the lines emanating from a Demux block can have different colors if they are driven by sources having different sample times. In this case, the Mux and Demux blocks are color coded as hybrids (yellow) to indicate that they handle signals with multiple rates.

Similarly, Subsystem blocks that contain blocks with differing sample times are also colored as hybrids, because there is no single rate associated with them. If all the blocks within a subsystem run at a single rate, the Subsystem block is colored according to that rate.

Connecting Blocks

Simulink block diagrams use lines to represent pathways for signals among blocks in a model (see “Annotating Diagrams” on page 4-16 for information on signals). Simulink can connect blocks for you or you can connect the blocks yourself by drawing lines from their output ports to their input ports.

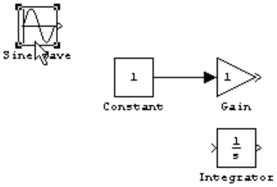
Automatically Connecting Blocks

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

Connecting Two Blocks

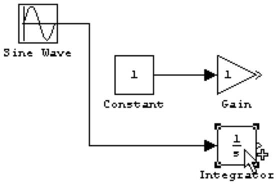
To autoconnect two blocks:

1 Select the source block.

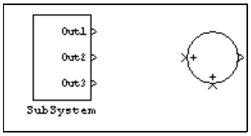


2 Hold down **Ctrl** and left-click the destination block.

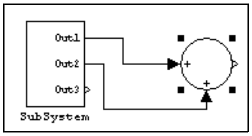
Simulink connects the source block to the destination block, routing the line around intervening blocks if necessary.



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



Before autoconnect



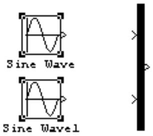
After autoconnect

Connecting Groups of Blocks

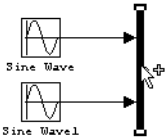
Simulink can connect a group of source blocks to a destination block or a source block to a group of destination blocks.

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

1 Select the source blocks.

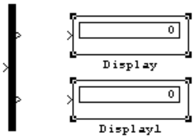


2 Hold down **Ctrl** and left-click the destination block.

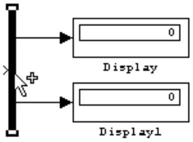


To connect a source block to a group of destination blocks:

1 Select the *destination* blocks.



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



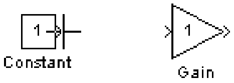
Manually Connecting Blocks

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

Drawing a Line Between Blocks

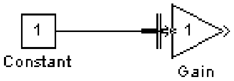
To connect the output port of one block to the input port of another block:

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

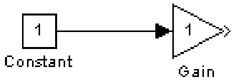


2 Press and hold down the mouse button.

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



4 Release the mouse button. 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.

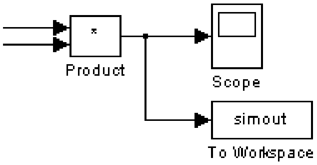


Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

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

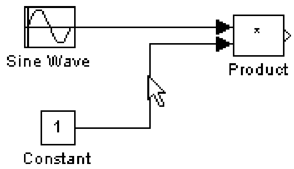
You might want to draw a line with segments exactly where you want them instead of where Simulink draws them. Or you might want to draw a line before you copy the block to which the line is connected. You can do either by drawing line segments.

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.

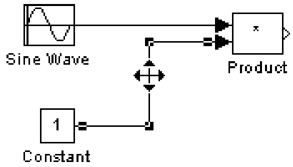
Moving a Line Segment

To move a line segment, follow these steps:

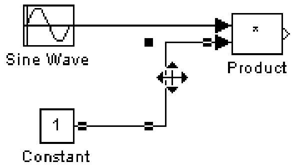
1 Position the pointer on the segment you want to move.



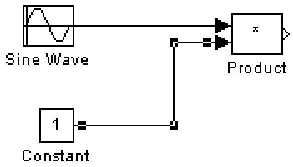
2 Press and hold down the left mouse button.



3 Drag the pointer to the desired location.



4 Release the mouse button.

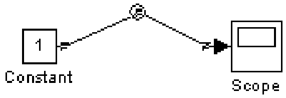


To move the segment connected to an input port, position the pointer over the port and drag the end of the segment to the new location. You cannot move the segment connected to an output port.

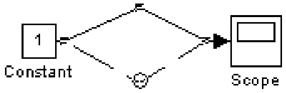
Moving a Line Vertex

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

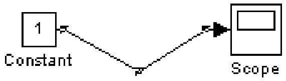
1 Position the pointer on the vertex, then press and hold down the mouse button. The cursor changes to a circle that encloses the vertex.



2 Drag the pointer to the desired location.



3 Release the mouse button.

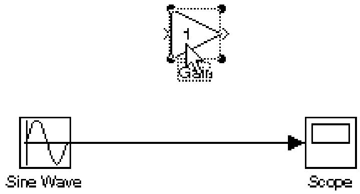


Inserting Blocks in a Line

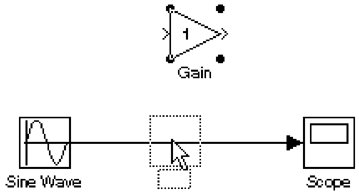
You can insert a block in a line by dropping the block on the line. Simulink inserts the block for you at the point where you drop the block. The block that you insert can have only one input and one output.

To insert a block in a line:

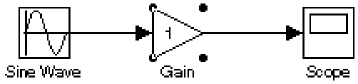
1 Position the pointer over the block and press the left mouse button.



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.

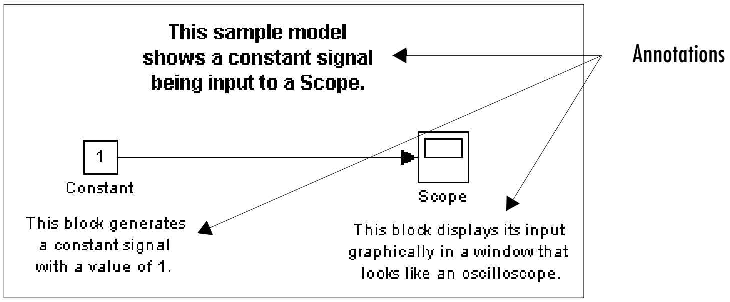


Disconnecting Blocks

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

Annotating Diagrams

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


This sample model shows a constant signal being input to a Scope.
Annotations
1
Constant
Scope
This block generates a constant signal with a value of 1.
This block displays its input graphically in a window that looks like an oscilloscope.

To create a model annotation, double-click an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point. Start typing the annotation contents. Each line is centered 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, click the annotation, then double-click or drag the cursor to select it. Then, enter the new annotation.
- To insert characters, click between two characters to position the insertion point, then insert text.
- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To delete an annotation, hold down the **Shift** key while you select the annotation, then press the **Delete** or **Backspace** key.

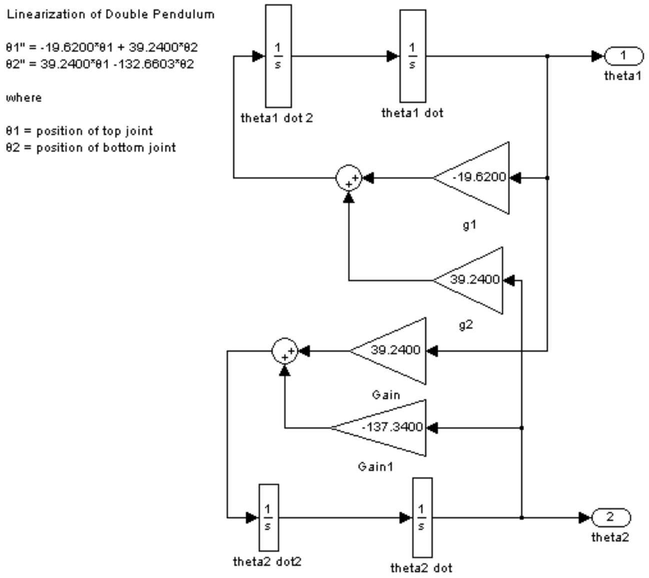
To change the font of all or part of an annotation, select the text in the annotation you want to change, then choose **Font** from the **Format** menu. Select a font and size from the dialog box.

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.

Using TeX Formatting Commands in Annotations

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



Linearization of Double Pendulum

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

where

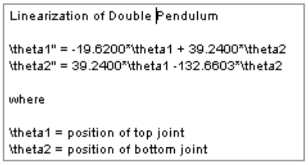
θ1 = position of top joint
θ2 = position of bottom joint

To use TeX commands in an annotation:

1 Select the annotation.

2 Select **Enable TeX Commands** from the **Edit** menu on the model window.

3 Enter or edit the text of the annotation, using TeX commands where needed to achieve the desired appearance.

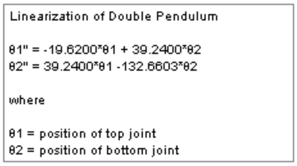


```
Linearization of Double Pendulum

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

where

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

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

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

Simulink displays the formatted text.



```
Linearization of Double Pendulum

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

where

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

Creating Annotations Programmatically

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

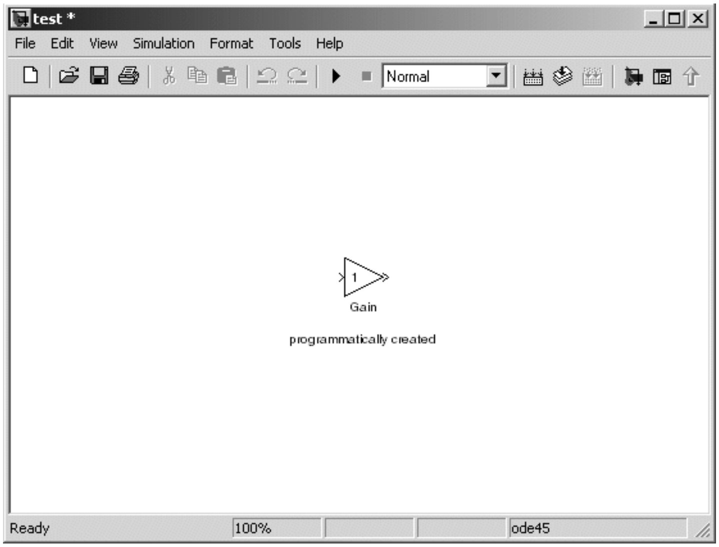
```
add_block('built-in/Note','path/text','Position', [center_x, 0,
0, center_y]);
```

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

```
new_system('test')
open_system('test')
```

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

creates the following model:



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:

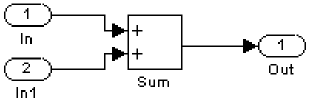
1. Copy the Subsystem block from the Signals & Systems library into your model.

2. Open the Subsystem block by double-clicking it.

   Simulink opens the subsystem in the current or a new model window, depending on the model window reuse mode that you selected (see “Window Reuse” on page 4-23).

3 In the empty Subsystem window, create the subsystem. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output.

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

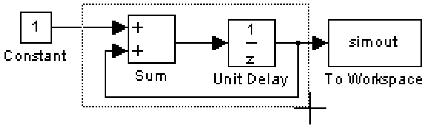


Creating a Subsystem by Grouping Existing Blocks

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

1 Enclose the blocks and connecting lines that you want to include in the subsystem within a bounding box. You cannot specify the blocks to be grouped by selecting them individually or by using the **Select All** command. For more information, see “Selecting Multiple Objects Using a Bounding Box” on page 4-3.

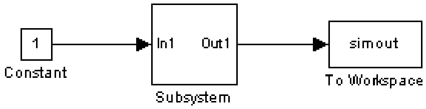
For example, this figure shows a model that represents a counter. The Sum and Unit Delay blocks are selected within a bounding box.



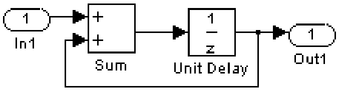
When you release the mouse button, the two blocks and all the connecting lines are selected.

2 Choose **Create Subsystem** from the **Edit** menu. Simulink replaces the selected blocks with a Subsystem block.

This figure shows the model after you choose the **Create Subsystem** command (and resize 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. You can also use the masking feature to customize the block's appearance and dialog box. See Chapter 12, "Creating Masked Subsystems."

Undoing Subsystem Creation

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

Model Navigation Commands

Subsystems allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy, using the Simulink Model Browser (see “The Model Browser” on page 9-22) 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, press **Enter**, or double-click the subsystem.

- **Open block in new window**

  Opens the currently selected subsystem regardless of the Simulink window reuse settings (see “Window Reuse” on page 4-23).

- **Go to Parent**

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

Window Reuse

You can specify whether Simulink 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

Reuse Type	Open Action	Go to Parent (Esc) Action
`replace`	Subsystem appears in a new window. Parent window disappears.	Parent window appears. Subsystem window disappears.
`mixed`	Subsystem appears in its own window.	Parent window rises to front. Subsystem window disappears.

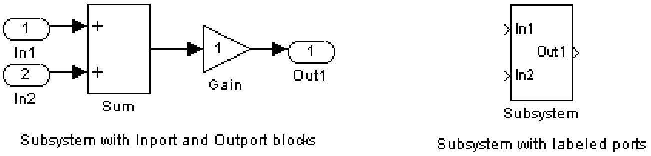
Labeling Subsystem Ports

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

You can hide (or show) the port labels by

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

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



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 the Subsystem block 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 "Working with Block Libraries" on page 5-29).

Creating Conditionally Executed Subsystems

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 you are building complex models that contain components whose execution depends on other components.

Simulink supports the following 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 in “Enabled Subsystems” on page 4-26.
- A *triggered subsystem* executes once each time a trigger event occurs. A trigger event can occur on the rising or falling edge of a trigger signal, which can be continuous or discrete. Triggered subsystems are described in more detail in “Triggered Subsystems” on page 4-31.
- 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 4-35 for more information.
- A *control flow subsystem* executes one or more times at the current time step when enabled by a control flow block that implements control logic similar to that expressed by programming language control flow statements (e.g., `if-then`, `while`, `do`, and `for`. See “Modeling with Control Flow Blocks” on page 4-42 for more information.

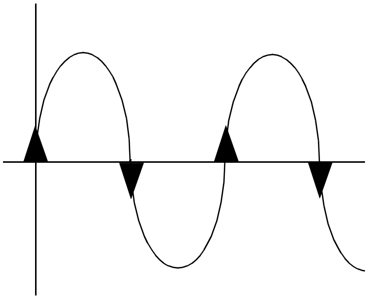
Enabled Subsystems

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.

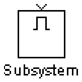
For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled, as shown in this figure. An up arrow signifies enable, a down arrow disable.



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

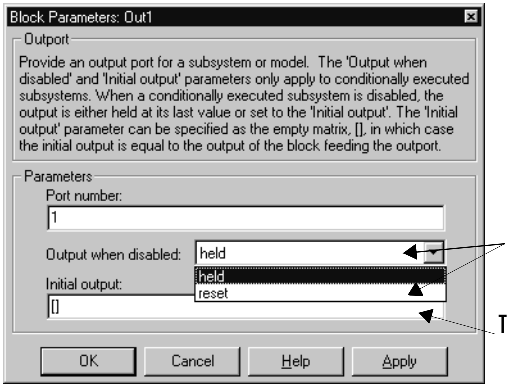
You create an enabled subsystem by copying an Enable block from the Signals & Systems library into a subsystem. Simulink adds an enable symbol and an enable control input port to the Subsystem block.



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

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

- Choose `held` to 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.



Provide an output port for a subsystem or model. The 'Output when disabled' and 'Initial output' parameters only apply to conditionally executed subsystems. When a conditionally executed subsystem is disabled, the output is either held at its last value or set to the 'Initial output'. The 'Initial output' parameter can be specified as the empty matrix, [], in which case the initial output is equal to the output of the block feeding the outport.

Port number: 1

Output when disabled: held

Initial output: []

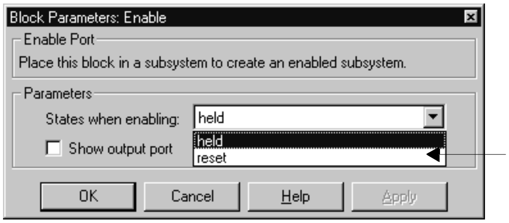
Select an option to set the Outport output while the subsystem is disabled.

The initial condition and the value when reset.

Setting States When the Subsystem Becomes Reenabled. 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 following:

- Choose `held` to cause the states to maintain their most recent values.
- Choose `reset` to cause the states to revert to their initial conditions.



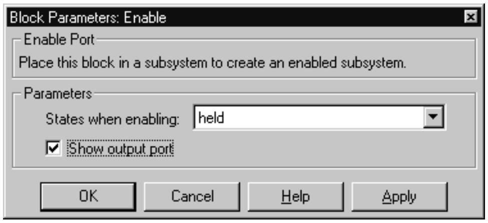
Place this block in a subsystem to create an enabled subsystem.

States when enabling: held

□ Show output port

Select an option to set the states when the subsystem is reenabled.

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.



Block Parameters: Enable

Enable Port

Place this block in a subsystem to create an enabled subsystem.

Parameters

States when enabling: held

☑ Show 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

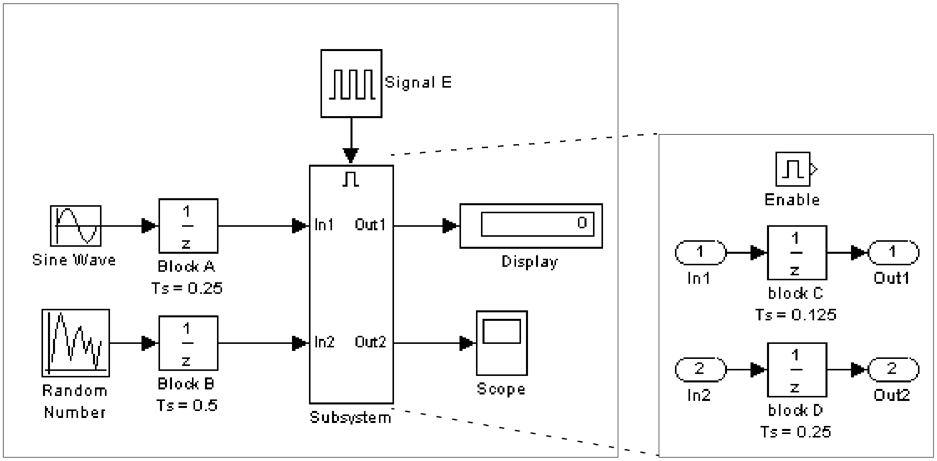
An enabled subsystem can contain any block, whether continuous or discrete. Discrete blocks in an enabled subsystem execute only when the subsystem executes, and only when their sample times are synchronized with the simulation sample time. Enabled subsystems and the model use a common clock.

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.

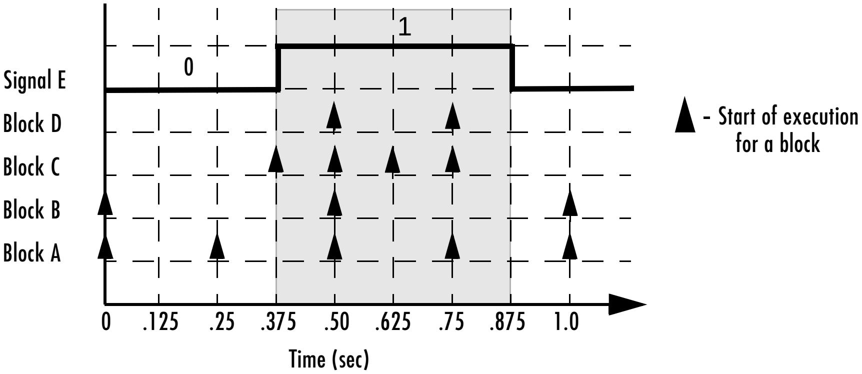
For example, this system contains four discrete blocks and a control signal. The discrete blocks are

- 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 independently of the enable control signal because they are not part of the enabled subsystem. When the enable control signal becomes positive, blocks C and D execute at their assigned sample rates until the enable control signal becomes zero again. Note that block C does not execute at 0.875 second when the enable control signal changes to zero.

Triggered Subsystems

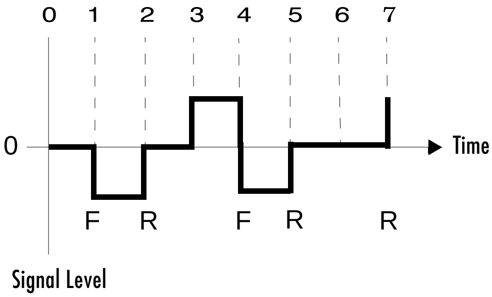
Triggered subsystems are subsystems that execute each time a trigger event occurs.

A triggered subsystem has a single control input, called the *trigger input*, that determines whether the subsystem executes. You can choose from three types of trigger events to force a triggered subsystem to begin execution:

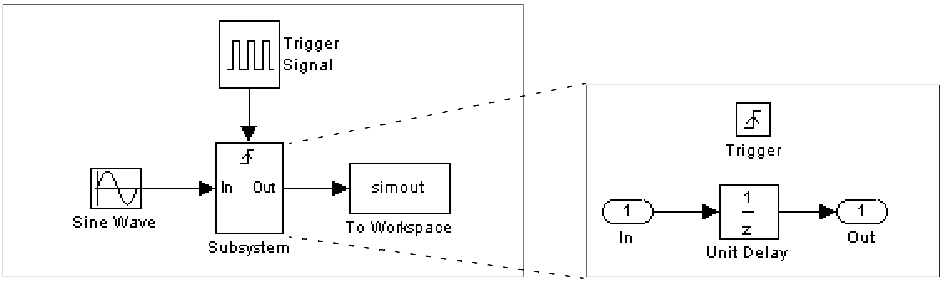
- `rising` triggers execution of the subsystem when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative).
- `falling` triggers execution of the subsystem when the control signal falls from a positive or a zero value to a negative value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

Note In the case of discrete systems, a signal's rising or falling from zero is considered a trigger event only if the signal has remained at zero for more than one time step preceding the rise or fall. This eliminates false triggers caused by control signal sampling.

For example, in the following timing diagram for a discrete system, a rising trigger (R) does not occur at time step 3 because the signal has remained at zero for only one time step when the rise occurs.



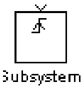
A simple example of a triggered subsystem is illustrated.



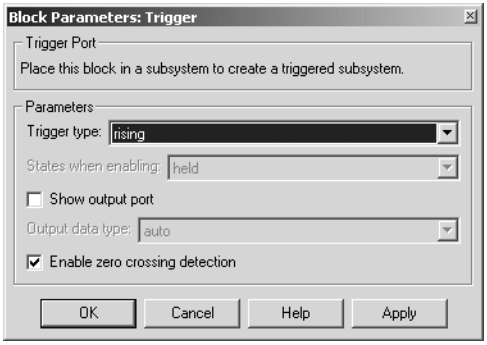
In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Creating a Triggered Subsystem

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



To select the trigger type, open the Trigger block dialog box and select one of the choices for the **Trigger type** parameter, as shown in the following dialog box:



Block Parameters: Trigger

Trigger Port

Place this block in a subsystem to create a triggered subsystem.

Parameters

Trigger type: rising

States when enabling: held

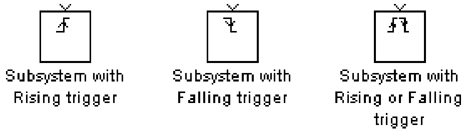
□ Show output port

Output data type: auto

☑ Enable zero crossing detection

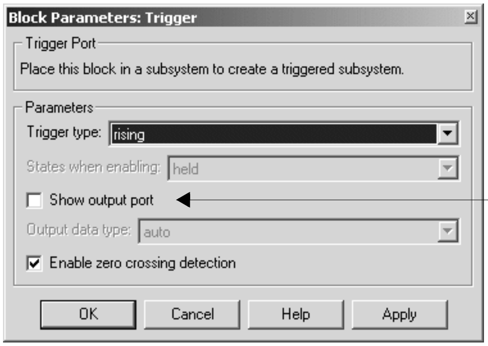
Select the trigger type.

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.



Block Parameters: Trigger

Trigger Port

Place this block in a subsystem to create a triggered subsystem.

Parameters

Trigger type: rising

States when enabling: held

□ Show output port

Output data type: auto

☑ Enable zero crossing detection

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 “Function-Call Subsystems” in the “Implementing Block Features” section of *Writing S-Functions*.

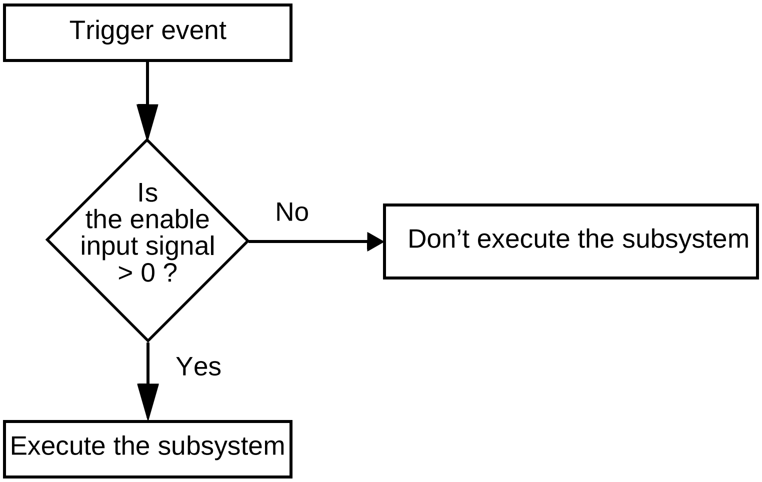
Blocks That a Triggered Subsystem Can Contain

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 times 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.

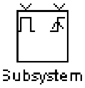


A triggered and enabled subsystem contains both an enable input port and a trigger input port. When the trigger event occurs, Simulink checks the enable input port to evaluate the enable control signal. If its value is greater than zero, Simulink executes the subsystem. If both inputs are vectors, the 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.

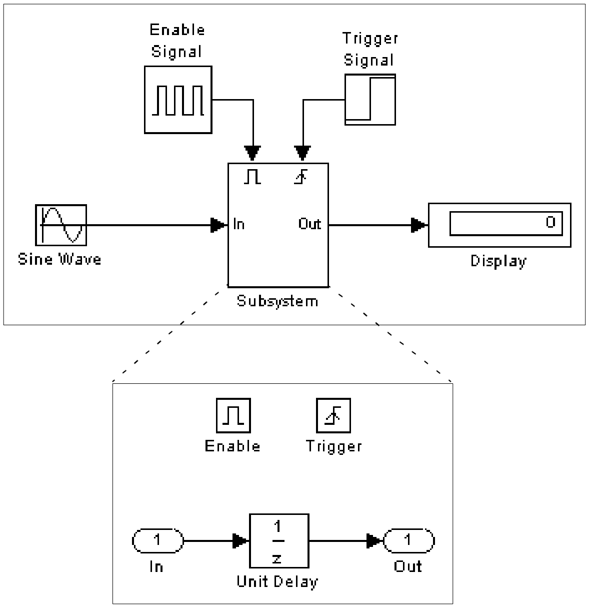


You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see “Setting Output Values While the Subsystem Is Disabled” on page 4-27. Also, you can specify what the values of the states are when the subsystem is reenabled. See “Setting States When the Subsystem Becomes Reenabled” on page 4-28.

Set the parameters for the Enable and Trigger blocks separately. The procedures are the same as those described for the individual blocks.

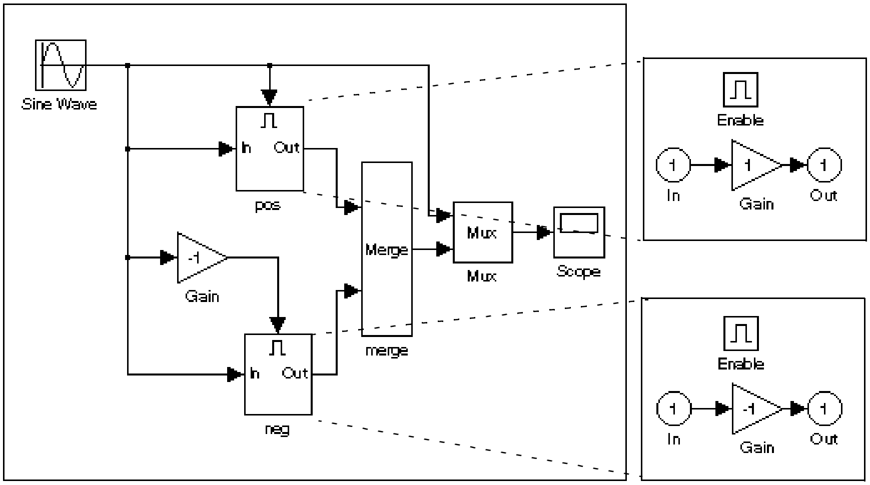
A Sample Triggered and Enabled Subsystem

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



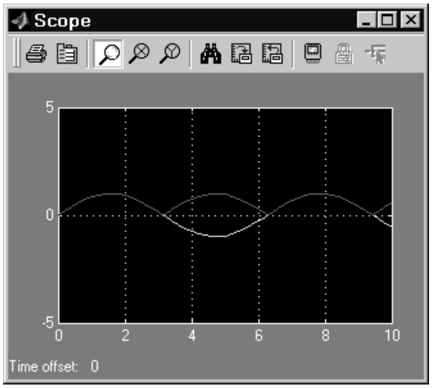
Creating Alternately Executing Subsystems

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 a full-wave rectifier, 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.

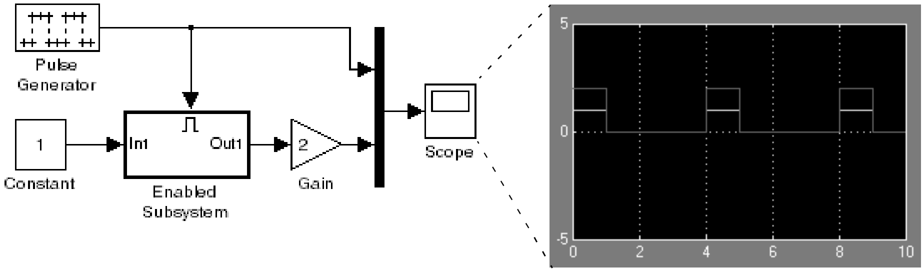
The Scope creates the following display.



Conditional Execution Behavior

To speed simulation of a model, Simulink by default avoids unnecessary execution of blocks connected to Switch, Multiport Switch, and conditionally executed blocks, a behavior called *conditional execution (CE)* behavior. You can disable this behavior for all Switch and Multiport Switch blocks in a model or for specific conditionally executed subsystems (see “Disabling Conditional Execution Behavior” on page 4-40).

The following model illustrates conditional execution behavior.



Simulink computes the outputs of the Constant block and Gain block only when the enabled subsystem executes (i.e., at time steps 0, 4, 8, and so on). This is because the output of the Constant block is required and the input of the Gain block changes only when the enabled subsystem executes. When CE behavior is off, Simulink computes the outputs of the Constant and Gain blocks at every time step, regardless of whether the outputs are needed or change.

In this example, Simulink regards the enabled subsystem as defining an execution context for the Constant and Gain blocks. Although the blocks reside graphically in the model’s root system, Simulink invokes the blocks’ methods during simulation as if the blocks reside in the enabled subsystem.

Propagating Execution Contexts

In general, Simulink defines an *execution context* as a set of blocks to be executed as a unit. At model compilation time, Simulink associates an execution context with the model’s root system and with each of its nonvirtual subsystems. Initially, the execution context of the root system and each nonvirtual subsystem is simply the blocks that it contains.

When compiling a model, Simulink examines each block in the model to determine whether it meets the following conditions:

- Its output is required only by a conditionally executed subsystem or its input changes only as a result of the execution of a conditionally executed.
- The subsystem's execution context can propagate across its boundaries.
- The output of the block is not a testpoint (see "Working with Test Points" on page 6-35).
- The block is allowed to inherit its conditional execution context.

  Simulink does not allow some built-in blocks, e.g., the Delay block, ever to inherit their execution context. Also, S-Function blocks can inherit their execution context only if they specify the `SS_OPTION_CAN_BE_CALLED_CONDITIONALLY` option.
- The block is not a multirate block.
- Its sample time is inherited (`-1`) or constant (`inf`).

If a block meets these conditions and execution context propagation is enabled for the associated conditionally executed subsystem (see "Disabling Conditional Execution Behavior" on page 4-40), Simulink moves the block into the execution context of the subsystem. This ensures that the block's methods are executed during the simulation loop only when the corresponding conditionally executed subsystem executes.

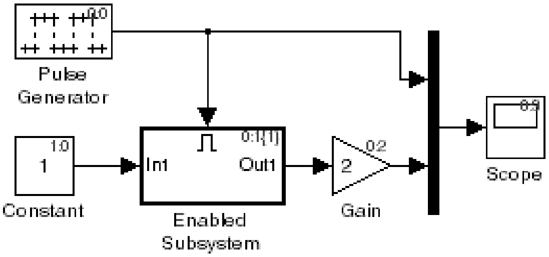
Behavior for Switch Blocks

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

Displaying Execution Contexts

To determine the execution context to which a block belongs, select **Sorted order** from the model window's **Format** menu. Simulink displays the sorted order index for each block in the model in the upper right corner of the block. The index has the format `s:b`, where `s` specifies the subsystem to whose execution context the block, `b`, belongs.

Simulink also expands the sorted order index of conditionally executed subsystems to include the system ID of the subsystem itself in curly brackets as illustrated in the following figure.



In this example, the sorted order index of the enabled subsystem is `0:1{1}`. The `0` indicates that the enabled subsystem resides in the model's root system. The first `1` indicates that the enabled subsystem is the second block on the root system's sorted list (zero-based indexing). The 1 in curly brackets indicates that the system index of the enabled subsystem itself is 1. Thus any block whose system index is 1 belongs to the execution context of the enabled subsystem and hence executes when it does. For example, the Constant block's index, 1:0, indicates that it is the first block on the sorted list of the enabled subsystem, even though it resides in the root system.

Disabling Conditional Execution Behavior

To disable conditional execution behavior for all Switch and Multiport Switch blocks in a model, turn off the `Conditional input branch execution` optimization on the **Optimization** pane of the **Configuration Parameters** dialog box (see "The Optimization Pane" on page 10-43). To disable conditional execution behavor for a specific conditionally executed subsystem, uncheck the **Propagate execution context across subsystem boundary** option on the subsystem's parameter dialog box.

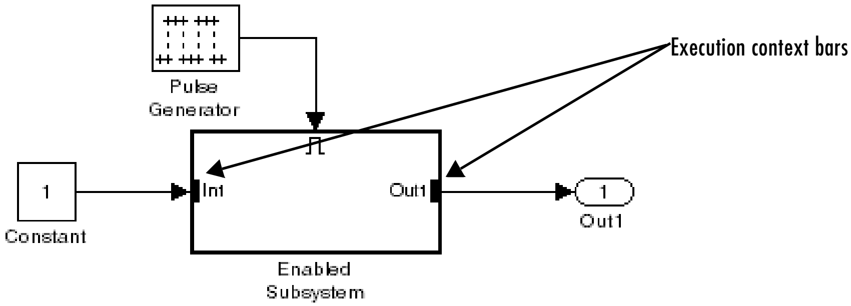
Even if this option is enabled, a subsystem's execution context cannot propagate across its boundaries under either of the following circumstances:

- The subsystem is a triggered subsystem with a latched input port.

- The subsystem has one or more output ports that specify an initial condition, i.e., whose initial condition is other than `[]`. In this case, a block connected to the subsystem's output cannot inherit the subsystem's execution context.

Displaying Execution Context Bars

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



To display the bars, select **Execution context indicator** from the model editor's **Format -> Block Displays** menu.

Modeling with Control Flow Blocks

The control flow blocks are used to implement the logic of the following C-like control flow statements in Simulink:

- `for`
- `if-else`
- `switch`
- `while` (includes `while` and `do-while` control flow statements)

Although all the preceding control flow statements are implementable in Stateflow®, these blocks are intended to provide Simulink users with tools that meet their needs for simpler logical requirements.

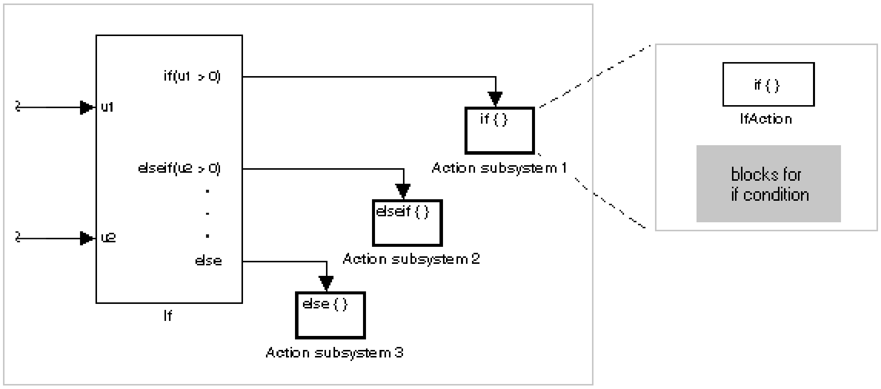
Creating Conditional Control Flow Statements

You create C-like conditional control flow statements using ordinary subsystems and the following blocks from the Subsystems library.

C Statement	Blocks Used
`if-else`	If, Action Port
`switch`	Switch Case, Action Port

If-Else Control Flow Statements

The following diagram depicts a generalized `if-else` control flow statement implementation in Simulink.



Construct a Simulink `if-else` control flow statement as follows:

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

  Inputs to the If block are set in the If block properties dialog. Internally, they are designated as `u1, u2,..., un` and are used to construct output conditions.

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

  Output ports for the If block are also set in its properties dialog. You use the input values `u1, u2, ..., un` to express conditions for the if, elseif, and else condition fields in the dialog. Of these, only the if field is required. You can enter multiple elseif conditions and select a check box to enable the else condition.

- Connect each condition output port to an Action subsystem.

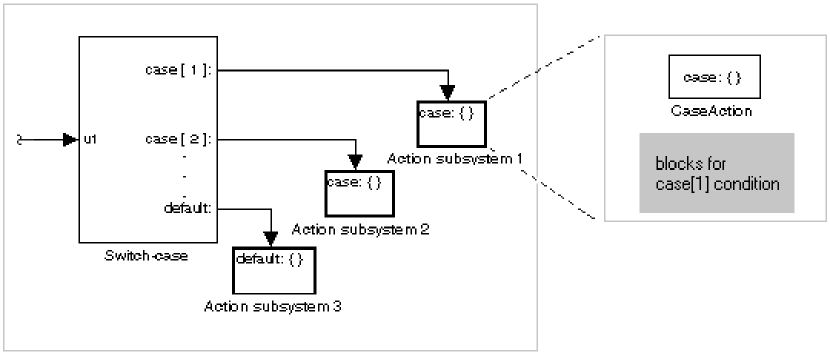
  Each if, elseif, and else condition output port on the If block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing an Action Port block in a subsystem. This creates an atomic Action subsystem with a port named Action, which you then connect to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

For more detailed information, see the reference topics for the If and Action Port blocks.

Note All blocks in an Action subsystem driven by an If or Switch Case block must run at the same rate as the driving block.

Switch Control Flow Statements

The following diagram depicts a generalized `switch` control flow statement implementation in Simulink.



Construct a Simulink `switch` control flow statement as follows:

- Provide a data input to the argument input of the Switch Case block.

  The input to the Switch Case block is the argument to the `switch` control flow statement. This value determines the appropriate case to execute. Noninteger inputs to this port are truncated.
- Add cases to the Switch Case block based on the numeric value of the argument input.

  You add cases to the Switch Case block through the properties dialog of the Switch Case block. Cases can be single or multivalued. You can also add an optional default case, which is true if no other cases are true. Once added, these cases appear as output ports on the Switch Case block.

- Connect each Switch Case block case output port to an Action subsystem.

  Each case output of the Switch Case block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing an Action Port block in a subsystem. This creates an atomic subsystem with a port named Action, which you then connect to a condition on the Switch Case block. Once connected, the subsystem takes on the identity of the condition and behaves like an enabled subsystem. Place all the block programming executed for that case in this subsystem.

For more detailed information, see the reference topics for the Switch Case and Action Port blocks.

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

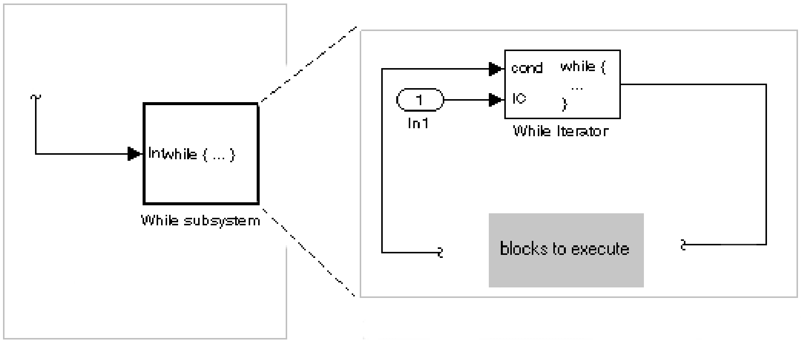
Creating Iterator Control Flow Statements

You create C-like iterator control flow statements using subsystems and the following blocks from the Subsystems library.

C Statement	Blocks Used
`do-while`	While Iterator
`for`	For Iterator
`while`	While Iterator

While Control Flow Statements

The following diagram depicts a generalized C-like `while` control flow statement implementation in Simulink.



In a Simulink `while` control flow statement, the While Iterator block iterates the contents of a While subsystem, an atomic subsystem. For each iteration of the While Iterator block, the block programming of the While subsystem executes one complete path through its blocks.

Construct a Simulink `while` control flow statement as follows:

- Place a While Iterator block in a subsystem.

  The host subsystem becomes a `while` control flow statement as indicated by its new label, `while {...}`. These subsystems behave like triggered subsystems. This subsystem is host to the block programming you want to iterate with the While Iterator block.

- Provide a data input for the initial condition data input port of the While Iterator block.

  The While Iterator block requires an initial condition data input (labeled `IC`) for its first iteration. This must originate outside the While subsystem. If this value is nonzero, the first iteration takes place.

- Provide data input for the conditions port of the While Iterator block.

  Conditions for the remaining iterations are passed to the data input port labeled `cond`. Input for this port must originate inside the While subsystem.

- You can set the While Iterator block to output its iterator value through its properties dialog.

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

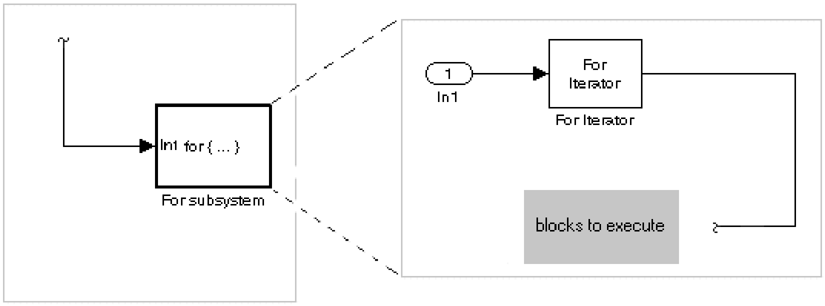
- You can change the iteration of the While Iterator block to `do-while` through its properties dialog.

  This changes the label of the host subsystem to `do {...} while`. With a `do-while` iteration, the While Iteration block no longer has an initial condition (IC) port, because all blocks in the subsystem are executed once before the condition port (labeled `cond`) is checked.

For specific details, see the reference topic for the While Iterator block.

For Control Flow Statements

The following diagram depicts a generalized `for` control flow statement implementation in Simulink.



In a Simulink `for` control flow statement, the For Iterator block iterates the contents of a For Iterator Subsystem, an atomic subsystem. For each iteration of the For Iterator block, the block programming of the For Iterator Subsystem executes one complete path through its blocks.

Construct a Simulink `for` control flow statement as follows:

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

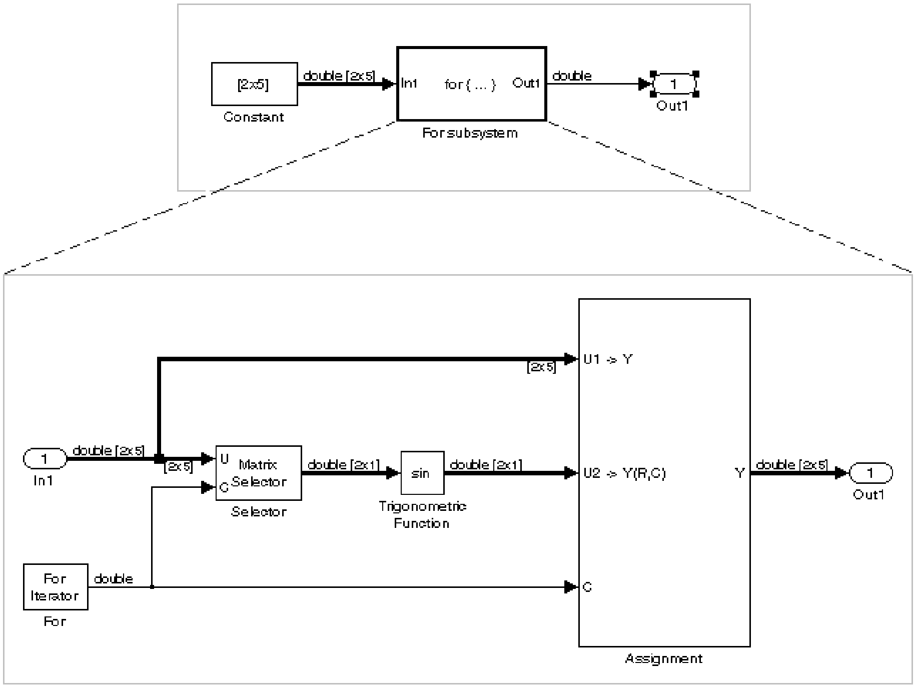
- You can set the For Iterator block to take external or internal input for the number of iterations it executes.

  Through the properties dialog of the For Iterator block you can set it to take input for the number of iterations through the port labeled `N`. This input must come from outside the For Iterator Subsystem.

  You can also set the number of iterations directly in the properties dialog.
- You can set the For Iterator block to output its iterator value for use in the block programming of the For Iterator Subsystem.

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

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



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

1. A 2-by-5 matrix is input to the Selector block and the Assignment block.
2. The Selector block strips off a 2-by-1 matrix from the input matrix at the column value indicated by the current iteration value of the For Iterator block.
3. The sine of the 2-by-1 matrix is taken.
4. The sine value 2-by-1 matrix is passed to an Assignment block.
5. The Assignment block, which takes the original 2-by-5 matrix as one of its inputs, assigns the 2-by-1 matrix back into the original matrix at the column location indicated by the iteration value.

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

Note Experienced Simulink users will note that the sin block is already capable of taking the sine of a matrix. The above example uses the sin block only as an example of changing each element of a matrix with the collaboration of an Assignment block and a For Iterator block.

Comparing Stateflow and Control Flow Statements

Stateflow already possesses the logical capabilities of the Simulink control flow statements. It can call Function-Call subsystems (see “Function-Call Subsystems” on page 4-34) on condition or iteratively. However, since Stateflow provides a great deal more in logical sophistication, if your requirements are simpler, you might find the capabilities of the Simulink control flow blocks sufficient for your needs. In addition, the control flow statements offer a few advantages, which are listed in the following topics.

Sample Times

The Function-Call subsystems that Stateflow can call are triggered subsystems. Triggered subsystems inherit their sample times from the calling block. However, the Action subsystems used in `if-else` and `switch` control flow statements and the While and For subsystems that make up `while` and `for` control flow statements are enabled subsystems. Enabled subsystems can have their own sample times independent of the calling block. This also allows you to use more categories of blocks in your iterated subsystem than in a Function-Call subsystem.

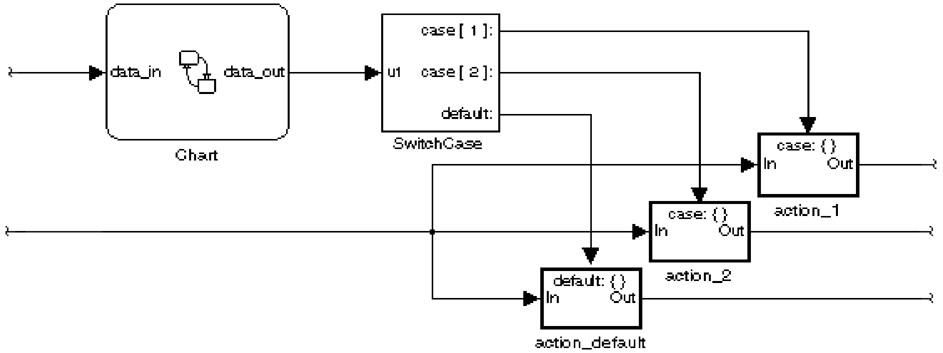
Resetting of States When Reenabled

Simulink control flow statement blocks allow you to retain or reset (to their initial values) the values of states for Action, For, and While subsystems when they are reenabled. For detailed information, see the references for the While Iterator and For Iterator blocks regarding the parameter **States when starting** and the reference for the Action Port block regarding the parameter **States when execution is resumed**.

Using Stateflow with the Control Flow Blocks

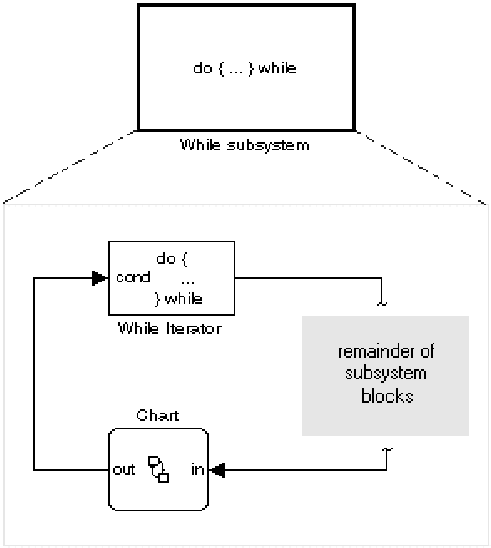
You might want to consider the possibility of using Stateflow and the Simulink control flow blocks together. The following sections contain some examples that give you a few suggestions on how to combine the two.

Using Stateflow with If-Else or Switch Subsystems. In the following model, Stateflow places one of a variety of values in a Stateflow data object. Upon chart termination, a Simulink `if` control flow statement uses that data to make a conditional decision.



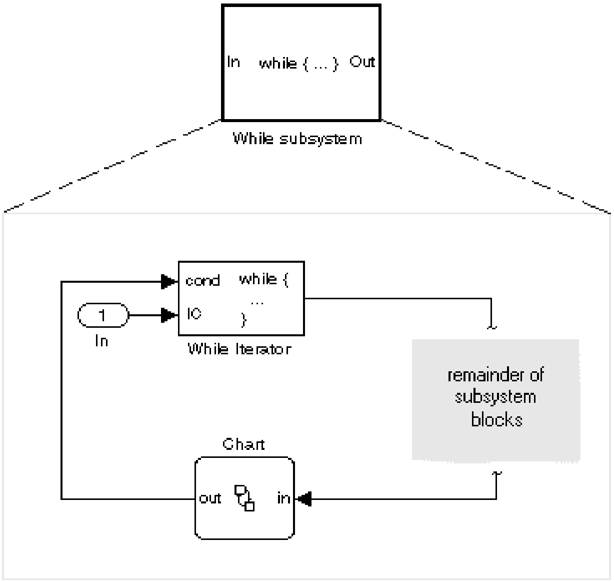
In this case, control is given to a Switch Case block, which uses the value to choose one of several case subsystems to execute.

Using Stateflow with While Subsystems. In the following diagram, Stateflow computes the value of a data object that is available to a condition input of a While Iterator block in `do-while` mode.



The While Iterator block has iterative control over its host subsystem, which includes the Stateflow Chart block. In `do-while` mode, the While block is guaranteed to operate for its first iteration value (= 1). During that time, the Stateflow chart is awakened and sets a data value used by the While Iterator block, which is evaluated as a condition for the next `while` iteration.

In the following diagram, the While block is now set in `while` mode. In this mode, the While Iterator block must have input to its initial condition port in order to execute its first iteration value. This value must come from outside the While subsystem.



If the initial condition is true, the While Iterator block wakes up the Stateflow chart and executes it to termination. During that time the Stateflow chart sets data, which the While Iterator condition port uses as a condition for the next iteration.

Referencing Models

Simulink allows you to include models in other models as blocks, a feature called model referencing. You create references to other models by creating instances of Model blocks in a parent model (see "Creating a Model Reference" on page 4-54). Each instance of a Model block in a parent model represents a reference to another model called a referenced model. A Model block displays inputs and outputs corresponding to the root-level inputs and outputs of the model it references, enabling you to incorporate the referenced model into the block diagram of the parent model.

During simulation, Simulink invokes an automatically generated S-function, called the referenced model's simulation target, to compute the Model block's outputs as needed. If the simulation target does not exist at the beginning of a simulation, Simulink generates it from the referenced model. If the simulation target does exist, Simulink checks whether the referenced model has changed significantly since the target was last generated. If so, Simulink regenerates the target to reflect the changes to the referenced model (see "Building Simulation Targets" on page 4-66 for more information).

Note Real-Time Workshop similarly generates library modules, called Real-Time Workshop targets, for the referenced models and a stand-alone executable for the root model, with the parent target invoking the referenced model targets to compute the referenced model outputs as needed. See the Real-Time Workshop documentation for more information.

A referenced model can itself reference other models. The topmost model in a hierarchy of model references is called the root model. A parent model can contain multiple references to the same model as long as the referenced model does not define global data. You can parameterize model references such that each reference to a model can specify different values for variables that define the model's behavior (see "Parameterizing Model References" on page 4-56 for more information).

Simulink includes a set of demos that illustrate various aspects of model referencing. To run the demos from the MATLAB command line, enter

```
mdlrefdemos
```

Model Referencing Versus Subsystems

Like subsystems, model referencing allows you to organize large models hierarchically, with Model blocks representing major subsystems. However, model referencing has significant advantages over subsystems in many applications. The advantages include:

- Modular development

  You can develop the referenced model independently from the models in which it is used.
- Inclusion by reference

  You can reference a model multiple times in another model without having to make redundant copies and multiple models can reference the same model.
- Incremental loading

  The referenced model is not loaded until it is needed, speeding up model loading (see “Incremental Loading” on page 4-65 for more information).
- Incremental code generation

  Simulink and Real-Time Workshop create binaries to be used in simulations and stand-alone applications to compute the outputs of the included blocks. If the binaries are up-to-date, that is, the binaries are not older than the models from which they were generated, no code generation occurs when models that reference them are simulated or compiled.

Simulink provides a tool to convert atomic subsystems to stand-alone models and to reconfigure the root model by replacing the subsystems with Model blocks. For further information, see “Converting Subsystems to Model References” on page 4-67. It also provides a command, `find_mdlrefs`, to find all models directly or indirectly referenced by a given model.

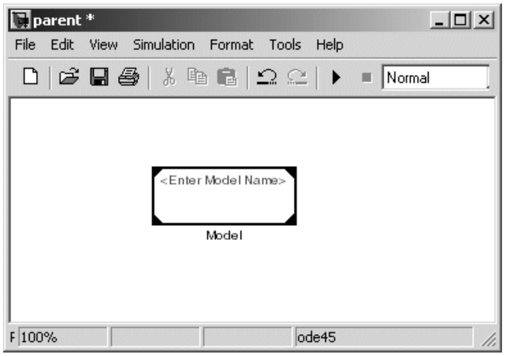
Creating a Model Reference

To create a reference to a model in another model:

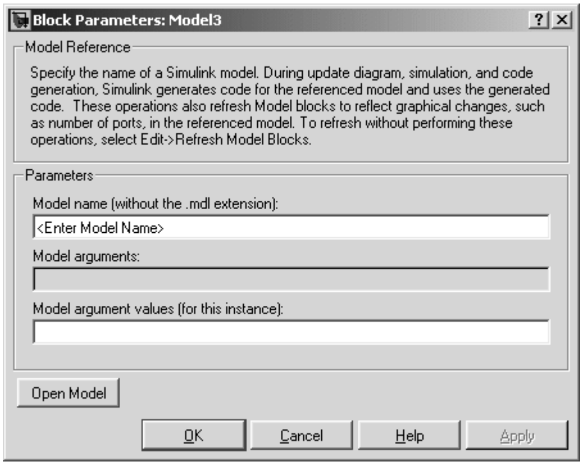
1. If the model is not on the MATLAB path, add it to the MATLAB path.

2. Enable the parent model’s **Inline parameters** optimization if it is not already enabled (see “Inline parameters” on page 10-44).

3 Create an instance of the Model block in the parent model (for example, by opening the Library Browser and dragging an instance from the Ports & Subsystems block library to the parent model).



4 Open the newly created Model block's parameter dialog box.


Block Parameters: Model3

Model Reference

Specify the name of a Simulink model. During update diagram, simulation, and code generation, Simulink generates code for the referenced model and uses the generated code. These operations also refresh Model blocks to reflect graphical changes, such as number of ports, in the referenced model. To refresh without performing these operations, select Edit->Refresh Model Blocks.

Parameters

Model name (without the .mdl extension): <Enter Model Name>

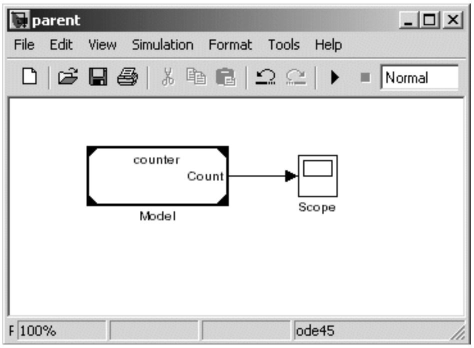
Model arguments:

Model argument values (for this instance):

5 Enter the name of the referenced model in the parameter dialog box's Model name field. The referenced model must be configured to use a fixed-step solver.

6 Click OK to apply the model name and close the dialog box.

If the referenced model contains any root-level inputs or outputs, Simulink displays corresponding input and output ports on the Model block instance that you have created. Use these ports to connect the reference model to other ports in the parent model.



Note See "Referenced Model I/O" on page 4-63 for information on connecting blocks in a parent model to a model that has bus inputs or outputs.

Opening a Referenced Model

To open a referenced model, select the Model block that references the model. Then select **Open Model** from the model editor's **Edit** menu or from the block's context menu.

Parameterizing Model References

Simulink allows you to parameterize references to models, i.e., use workspace variables to determine their behavior. You can parameterize a model in the following ways:

- Use *global nontunable parameters* in the MATLAB workspace or in a model workspace to determine the behavior of all references to a given model.

  A *global nontunable parameter* is a MATLAB variable or a `Simulink.Parameter` object whose storage class is `auto`. The value of such a variable cannot be changed during simulation.

- Use *global tunable parameters* in the MATLAB workspace to determine the behavior of all references to a given model in the model.

  A *global tunable parameter* is a parameter specified by an object of `Simulink.Parameter` class that has a storage class other than `auto`. The value of such a variable can be changed during simulation, allowing you to change the behavior of the referenced models.
- Use model arguments in the model to specify different behavior for different references to the same model (see next section).

Model Referencing and the Inline Parameters Optimization

Simulink does not support the `off` setting of the inline parameters optimization (see “Inline parameters” on page 10-44) for models that contain Model blocks. Simulink ignores the settings in the **Tunable Parameter** dialog box (see “Model Parameter Configuration Dialog Box” on page 10-47) for models that contain Model blocks and for referenced models. To help you convert existing models to model referencing, Simulink provides a command that converts tunable parameters specified in the **Tunable Parameter** dialog box, which do not work with model referencing, to global tunable parameters that do work with model referencing. Type

```
help tunablevars2parameterobjects
```

at the MATLAB command line for more information.

Using Model Arguments

Model arguments let you create references to the same model that behave differently. For example, suppose you want each reference to a counter model to be able to specify initial and increment values for the counter where the specified values can differ from reference to reference. Using model arguments to parameterize references to the counter model allows you to do this.

Note Run the `mdlref_paramargs` demo to see parameterized model references in action.

Using model arguments requires that you

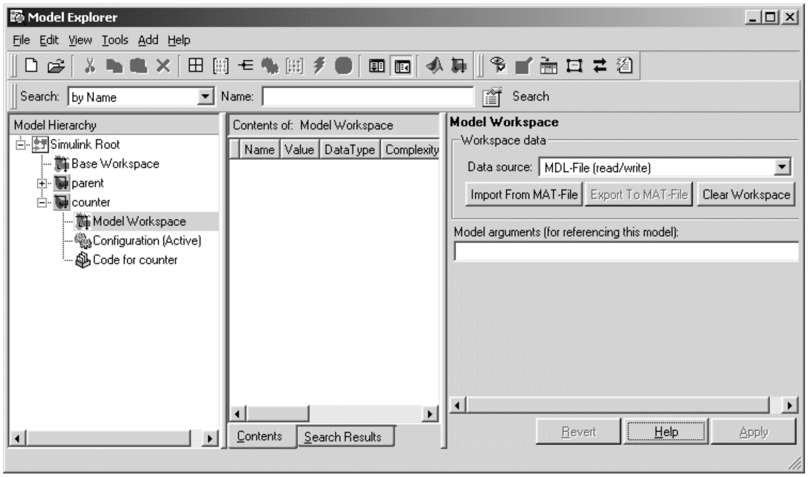
- Declare model workspace variables that determine the model's behavior as model arguments
- Assign values to the model arguments in each reference to the parameterized model

The following sections explain how to perform these tasks.

Declaring Model Arguments

To declare some or all of a model's model workspace variables as model arguments:

1. Open the referenced model.
2. Open the Model Explorer.
3. Select the model's workspace in the Model Explorer.

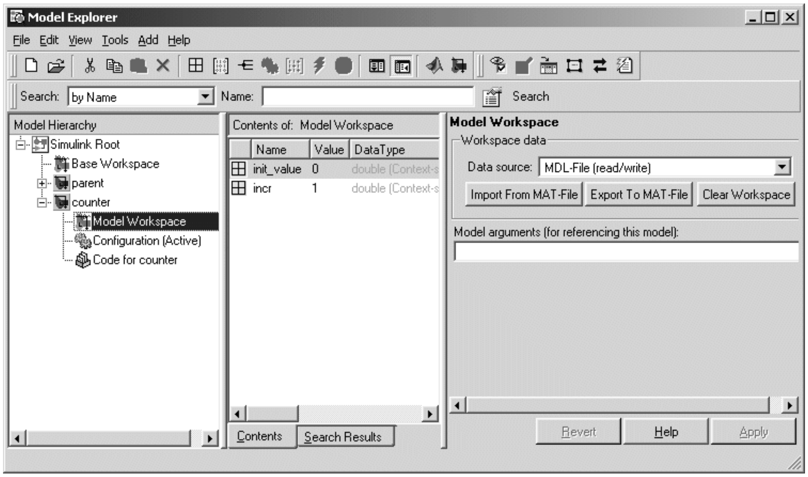


Search: by Name

Contents of: Model Workspace

Data source: MDL-File (read/write)

4 If you have not already done so, use the Model Explorer to create MATLAB variables in the model's workspace that determine the model's behavior.

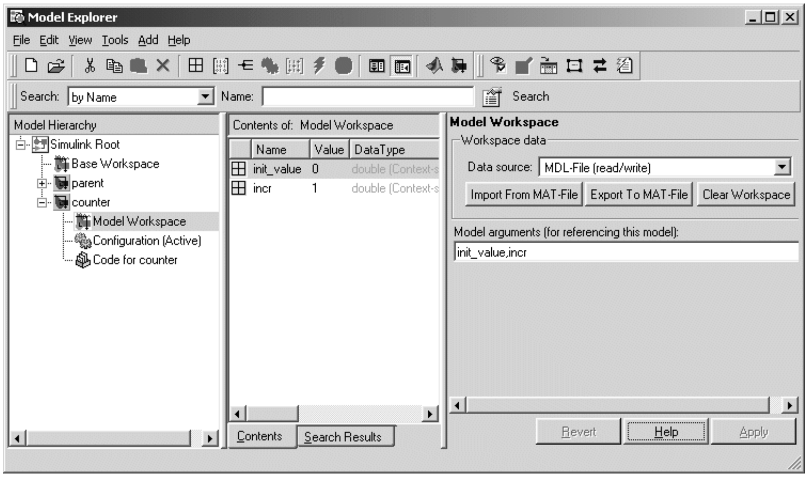


Name	Value	DataType
init_value	0	double (Context-s
incr	1	double (Context-s

Data source: MDL-File (read/write)

Model arguments (for referencing this model):

5 Enter the names of the workspace variables that you want to declare as model arguments as a comma-separated list in the **Model arguments** field in the model workspace's dialog box.



Name	Value	DataType
init_value	0	double (Context-s
incr	1	double (Context-s

Data source: MDL-File (read/write)

Model arguments (for referencing this model): init_value,incr

6 Click the **Apply** button on the dialog box to confirm the entered names.

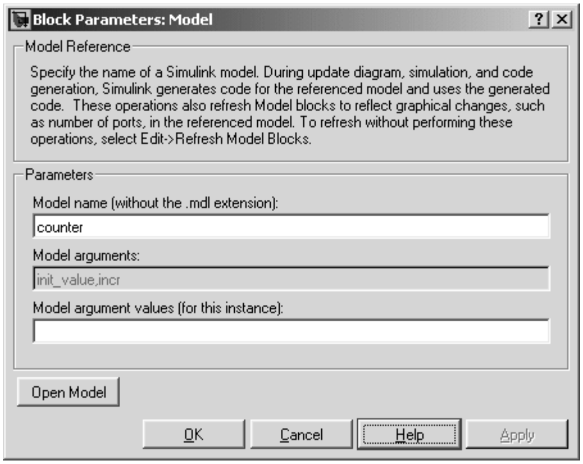
Note If a model does not declare a variable in its workspace as a model argument, the variable has the same value in every reference to the model and cannot be tuned from a parent model. For example, suppose that a model defines a variable `k` in its workspace but does not declare it as a model argument. Further, suppose that the model assigns a value of `5` to `k` in its workspace. Then the value of `k` will be `5` in every reference to the model in other models.

Assigning Values to Model Arguments

If a model declares model arguments, you must assign values to the model arguments in each reference to the model, i.e., in each Model block that references the model.

To assign values to a model's model arguments in a Model block that references the model:

1 Open the Model block's parameter dialog box.



Block Parameters: Model

Model Reference

Specify the name of a Simulink model. During update diagram, simulation, and code generation, Simulink generates code for the referenced model and uses the generated code. These operations also refresh Model blocks to reflect graphical changes, such as number of ports, in the referenced model. To refresh without performing these operations, select Edit->Refresh Model Blocks.

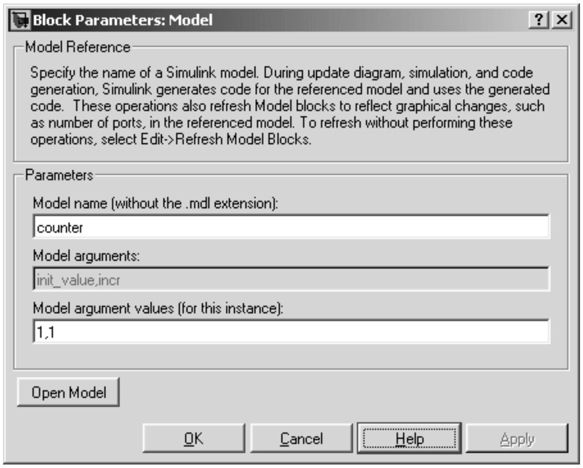
Parameters

Model name (without the .mdl extension): counter

Model arguments: init_value,incr

Model argument values (for this instance):

2 Enter a comma-delimited list of values for the parameter arguments in the **Model argument values** field in the same order in which the arguments appear in the **Model arguments** field.



Block Parameters: Model

Model Reference

Specify the name of a Simulink model. During update diagram, simulation, and code generation, Simulink generates code for the referenced model and uses the generated code. These operations also refresh Model blocks to reflect graphical changes, such as number of ports, in the referenced model. To refresh without performing these operations, select Edit->Refresh Model Blocks.

Parameters

Model name (without the .mdl extension): counter

Model arguments: init_value,incr

Model argument values (for this instance): 1,1

You can enter the values as literal values, variable names, MATLAB expressions, and Simulink parameter objects. The value for a particular argument must have the same dimensions and data and numeric type as the model workspace variable that defines the argument.

Model Block Sample Times

The sample times of a Model block are the sample times of the model that it references. If the referenced model needs to run at specific rates, the referenced model's simulation target specifies the required rates. Otherwise, the target specifies that the referenced model inherits its sample time from the parent model. Specifically, a referenced model inherits its sample time if all the following conditions are true:

- None of its blocks specify sample times (other than inherited and constant).
- It does not have any continuous states.
- It does not contain any blocks that use absolute time.
- It specifies a fixed-step solver but not a fixed step size.

- After sample time propagation, it has only one sample time (not counting constant and triggered sample time).
- It does not contain any blocks that preclude sample time inheritance (see "Blocks That Preclude Sample-Time Inheritance" on page 4-62)

You can use a referenced model that inherits its sample time anywhere in a parent model. By contrast, you cannot use a referenced model that has intrinsic sample times in a triggered, function call, or for an iterator subsystem. Further, to avoid rate transition errors, you must ensure that blocks connected to a referenced model with intrinsic samples times operate at the same rates as the referenced model.

To determine whether a referenced model inherits its sample time, set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane to `Ensure sample time independent` (see "Periodic sample time constraint" on page 10-36). If the model is unable to inherit sample times, this setting causes Simulink to display an error message when generating the referenced model's simulation (or Real-Time Workshop) target. To determine the intrinsic sample time of a referenced model (or the fastest intrinsic sample time for multirate referenced models), first update a model that references it. Then select a Model block that references the referenced model and enter the following command at the MATLAB command line:

```
get_param(gcb, 'CompiledSampleTime')
```

Blocks That Preclude Sample-Time Inheritance

Using a block whose output depends on an inherited sample time in a referenced model can cause a simulation to produce unexpected or erroneous results. For this reason, when building a simulation target for a model that does not need to run at a specified rate, Simulink checks whether the model contains any blocks, including any S-Function blocks, whose outputs are functions of the inherited simulation time. If so, Simulink generates a simulation target that specifies a default sample time and displays an error if you have set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane to `Ensure sample time independent` (see "Periodic sample time constraint" on page 10-36).

The outputs of the following built-in blocks depend on their inherited sample time and hence preclude a referenced model from inheriting its sample time from the parent model:

- Discrete-Time Integrator
- From Workspace (if it has input data that contains time)
- Probe (if probing sample time)
- Rate Limiter
- Sine Wave

Simulink assumes that the output of an S-function does not depend on inherited sample time unless the S-function explicitly declares the contrary (see “Writing S-Functions” for information on how to create S-functions that declare whether their output depends on their inherited sample time). Thus, to avoid simulation errors with referenced models that inherit their sample time, you need to take care not to include S-functions in the referenced models that fail to declare whether their output depends on their inherited sample time. Simulink by default warns you if your model contains such blocks when you update or simulate the model (see “Unspecified inheritability of sample time” on page 10-50).

Referenced Model I/O

Simulink imposes the following restrictions on connecting signals to the inputs and outputs of Model blocks.

Bus I/O Limitations

A parent model can reference a model with bus input or output ports only if each bus port meets the following conditions:

- The port is defined by a bus object, i.e., an instance of `Simulink.Bus` class specified as the value of the port block’s **Bus object** parameter.
- The bus object is defined in a workspace that is visible from both the parent and the referenced model, e.g., the MATLAB workspace for a model referenced by a root model.

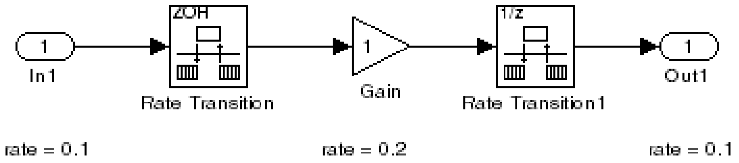
Similarly, the bus connected to a bus input port of a referenced model must be defined by the same bus object that defines the bus input, i.e., the bus must be created by a Bus Creator block whose Bus object parameter is set to the bus object as is the Inport of the referenced model. This explains why the bus object must be visible to both the parent and the referenced model.

Index I/O Limitations

In some circumstances, Simulink does not propagate 0- or 1-based indexing information to the root-level ports connected to blocks in the referenced model that accept indices, e.g., the Assignment block, or produce indices, e.g., the For Iterator block. In particular, if a root-level input port is connected to index inputs in the referenced model whose 0- or 1-based indexing properties differ, Simulink does not set the 0- or 1-based indexing property of the input port. Similarly, if a root-level output port of the referenced model is connected to index outputs in the model that have different 0- or 1-based indexing settings, Simulink does not set the 0- or 1-based indexing property of the root-level output port. This can cause Simulink to miss incompatible index connections when the model is referenced by another model.

Matching I/O Rates

In a referenced model, the first nonvirtual block connected downstream from a root-level Inport of the referenced model and the first nonvirtual block connected upstream from a root-level Outport must have the sample time as the Inport or Outport block. If the rates do not match when you update or start a simulation of the referencing model, Simulink halts and displays an error. You can use Rate Transition blocks to match the root-level input and output sample times as illustrated in the following diagram.



Model Interfaces

A referenced model's interface consists of its input and output ports and its parameter arguments. Model block instances depict the interfaces of the models they reference.

Incremental Loading

Simulink takes advantage of this fact to defer loading of referenced models until you update or simulate the model that references them. This feature, called incremental loading, allows you to begin editing a model before it is completely loaded, a useful capability when you need to make changes to large, complex models.

Note To take advantage of incremental loading, models referenced by Model blocks must have been opened and saved at least once in Release 14 (or a later version) of Simulink.

Refreshing Model Blocks

Refreshing Model blocks refers to the process of updating them to reflect graphical changes in the interfaces of the models they reference. To refresh all of a model's Model blocks, select **Refresh Model Blocks** from the model's **Edit** menu. To update a specific Model block, select **Refresh** from the block's context (pop-up) menu.

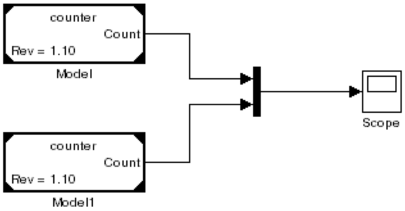
You should refresh a Model block instance if the model that it references has changed since the block was created or since it was last refreshed and the changes affect the block's graphical appearance, for example, the referenced model gained or lost a port. Simulink provides diagnostics that enable you to detect changes in the interfaces of referenced models that could require refreshing the Model blocks that reference them. The diagnostics include

- Model block version mismatch (see "Model block version mismatch" on page 10-60)
- I/O port and parameter mismatch (see "Port and parameter mismatch" on page 10-61)

Displaying Referenced Model Version Numbers

To display the version numbers of the models referenced by a model (see "Managing Model Versions" on page 4-103), select Model block version from the

Block displays submenu of the parent model’s Format menu. Simulink displays the version numbers in the icons of the corresponding Model block instances.



The version number displayed on a Model block’s icon refers to the version of the model used to create the block or refresh the block when it was last refreshed.

Building Simulation Targets

A simulation target is an S-function that computes the outputs of a referenced model during simulation of the model’s parent. You can command Simulink to generate simulation targets for model references at any time by updating the model’s diagram or by executing the `slbuild` command at the MATLAB command line or you can let Simulink determine whether and when to build the simulation targets. If the simulation target for a referenced model does not exist at the start of a simulation, Simulink generates the target. Subsequently, if the files or workspace variables used to build the target change, it may be necessary to rebuild the target to reflect the changes, depending on whether the changes affect target outputs. You can let Simulink determine whether to rebuild existing targets or specify that Simulink always or never rebuild targets at the beginning of a simulation (see “Rebuild options for all referenced models” on page 10-67).

While generating a target, Simulink displays status messages at the MATLAB command line to enable you to monitor the target generation process, which entails generating and compiling code and linking the compiled target code with compiled code from standard code libraries to create an executable file.

Simulink creates simulation targets in the current working directory. It stores intermediate files used to generate the simulation targets in separate subdirectories of a subdirectory of the working directory named `slprj`. If the `slprj` directory does not exist, Simulink creates it. The Simulink Accelerator and Real-Time Workshop also use the slprj subdirectory of the current working directory to store intermediate files used to build acceleration targets and stand-alone targets, respectively.

Project Directories

The policy of having all Simulink-related products store generated files in the same subdirectory of the current work directory makes it easy for you to keep all the generated files for a given project together and separate from generated files belonging to other projects. All that is required is that you create a separate directory for each project and make the directory for a given project the current working directory when you are working on the project.

Converting Subsystems to Model References

Converting an existing model to use model referencing can be a time-consuming and error-prone task if done by hand. Execute

```
mdlref_conversion
```

at the MATLAB command line for a demonstration of a way to automate this task.

Model Discretizer

The Model Discretizer selectively replaces continuous Simulink blocks with discrete equivalents. Discretization is a critical step in digital controller design and for hardware in-the-loop simulations. You can use this tool to prepare continuous models for use with the Real-Time Workshop Embedded Coder, which supports only discrete blocks.

The Model Discretizer enables you to

- Identify a model's continuous blocks.
- Change a block's parameters from continuous to discrete.
- Apply discretization settings to all continuous blocks in the model or to selected blocks.
- Create configurable subsystems that contain multiple discretization candidates along with the original continuous block(s).
- Switch among the different discretization candidates and evaluate the resulting model simulations.

Requirements

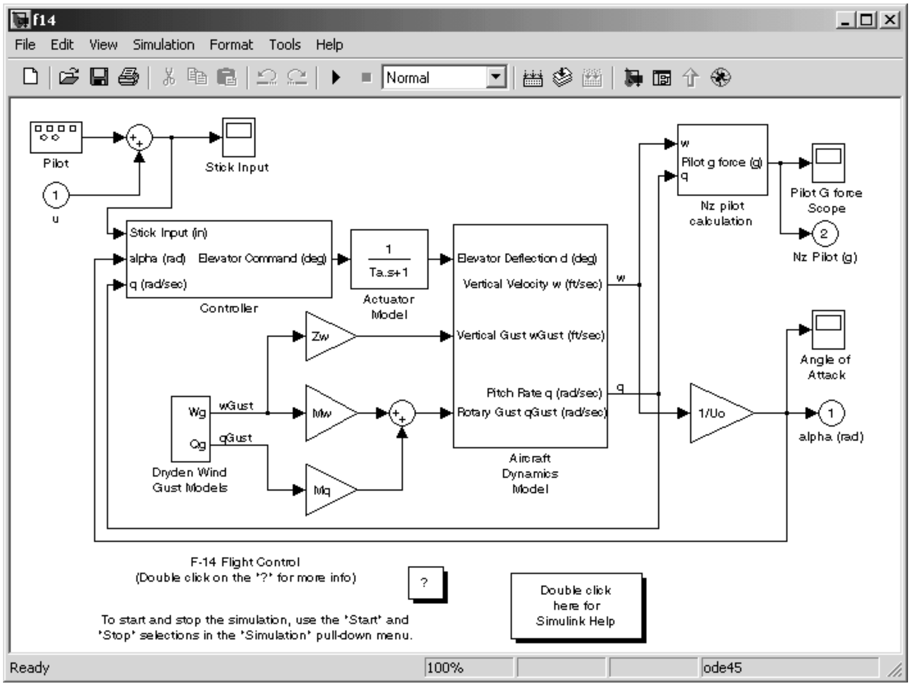
To use the Model Discretizer, you must have the Control System Toolbox, Version 5.2, installed.

Discretizing a Model from the Model Discretizer GUI

To discretize a model, follow these steps:

- “Start the Model Discretizer” on page 4-70
- “Specify the Transform Method” on page 4-70
- “Specify the Sample Time” on page 4-71
- “Specify the Discretization Method” on page 4-71
- “Discretize the Blocks” on page 4-75

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

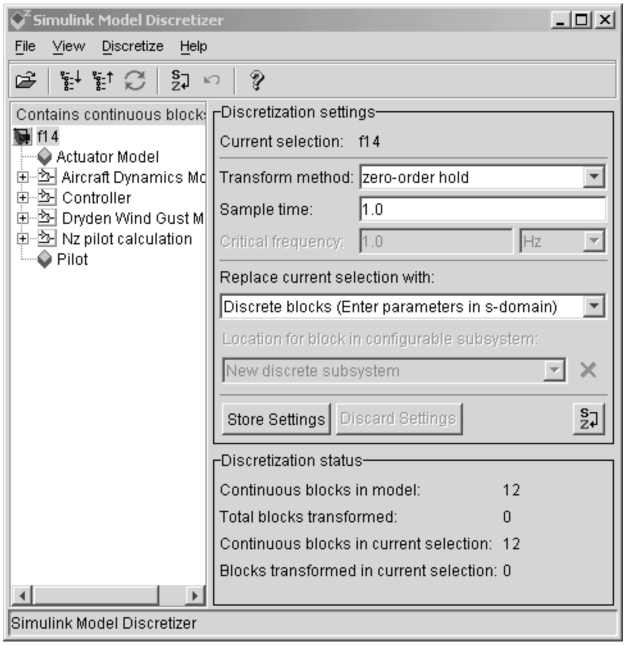


F-14 Flight Control
(Double click on the '?' for more info)

To start and stop the simulation, use the 'Start' and 'Stop' selections in the 'Simulation' pull-down menu.

Start the Model Discretizer

To open the tool, select **Model Discretizer** from the **Tools** menu in a Simulink model. This displays the **Simulink Model Discretizer** window.



Discretization settings

Current selection: f14

Transform method: zero-order hold

Sample time: 1.0

Critical frequency: 1.0 Hz

Replace current selection with:

Discrete blocks (Enter parameters in s-domain)

Location for block in configurable subsystem:

New discrete subsystem

Discretization status

Continuous blocks in model: 12

Total blocks transformed: 0

Continuous blocks in current selection: 12

Blocks transformed in current selection: 0

Alternatively, you can open the Model Discretizer from the MATLAB command window using the `slmdldiscui` function.

The following command opens the **Simulink Model Discretizer** window with the `f14` model.

```
slmdldiscui('f14')
```

To open a new Simulink model or library from the Model Discretizer, select **Load model** from the **File** menu.

Specify the Transform Method

The transform method specifies the type of algorithms used in the discretization. For more information on the different transform methods, see

Continuous/Discrete Conversions of LTI Models in the Control Systems Toolbox documentation.

The **Transform method** drop-down list contains the following options:

- `zero-order hold`
  Zero-order hold on the inputs.
- `first-order hold`
  Linear interpolation of inputs.
- `tustin`
  Bilinear (Tustin) approximation.
- `tustin with prewarping`
  Tustin approximation with frequency prewarping.
- `matched pole-zero`

  Matched pole-zero method (for SISO systems only).

Specify the Sample Time

Enter the sample time in the **Sample time** field.

You can specify an offset time by entering a two-element vector for discrete blocks or configurable subsystems. The first element is the sample time and the second element is the offset time. For example, an entry of [1.0 0.1] would specify a 1.0 second sample time with a 0.1 second offset. If no offset is specified, the default is zero.

You can enter workspace variables when discretizing blocks in the s-domain. See "Discrete blocks (Enter parameters in s-domain)" on page 4-72.

Specify the Discretization Method

Specify the discretization method in the **Replace current selection with** field. The options are

- `Discrete blocks (Enter parameters in s-domain)`
  Creates a discrete block whose parameters are retained from the corresponding continuous block.
- `Discrete blocks (Enter parameters in z-domain)`
  Creates a discrete block whose parameters are "hard-coded" values placed directly into the block's dialog.

- `Configurable subsystem (Enter parameters in s-domain)`

  Create multiple discretization candidates using s-domain values for the current selection. A configurable subsystem can consist of one or more blocks.
- `Configurable subsystem (Enter parameters in z-domain)`

  Create multiple discretization candidates in z-domain for the current selection. A configurable subsystem can consist of one or more blocks.

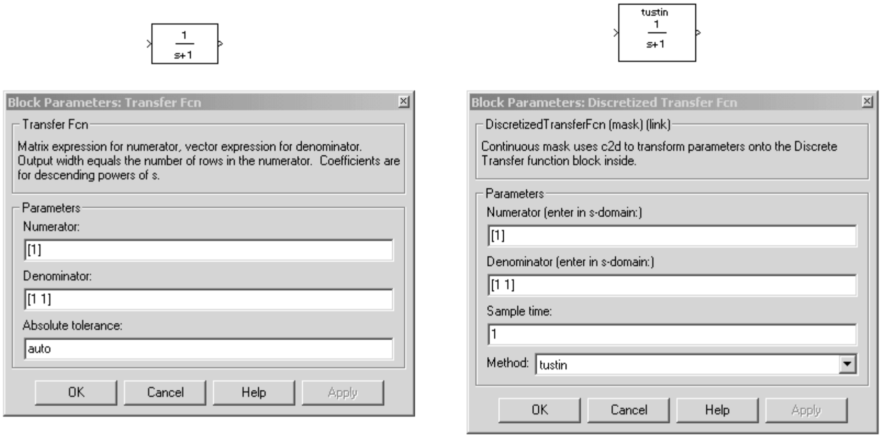
Discrete blocks (Enter parameters in s-domain). Creates a discrete block whose parameters are retained from the corresponding continuous block. The sample time and the discretization parameters are also on the block's parameter dialog.

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

These blocks have the unique capability of reverting to continuous behavior if the sample time is changed to zero. Entering the sample time as a workspace variable (`'Ts'`, for example) allows for easy changeover from continuous to discrete and back again. See “Specify the Sample Time” on page 4-71.

Note Parameters are not tunable when **Inline parameters** is selected in the model's **Configuration Parameters** dialog box.

The figure below shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the s-domain. The **Block Parameters** dialog box for each block is shown below the block.



Block Parameters: Transfer Fcn

Transfer Fcn

Matrix expression for numerator, vector expression for denominator. Output width equals the number of rows in the numerator. Coefficients are for descending powers of s.

Parameters

Numerator: [1]

Denominator: [1 1]

Absolute tolerance: auto

Block Parameters: Discretized Transfer Fcn

DiscretizedTransferFcn (mask) (link)

Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside.

Parameters

Numerator (enter in s-domain:) [1]

Denominator (enter in s-domain:) [1 1]

Sample time: 1

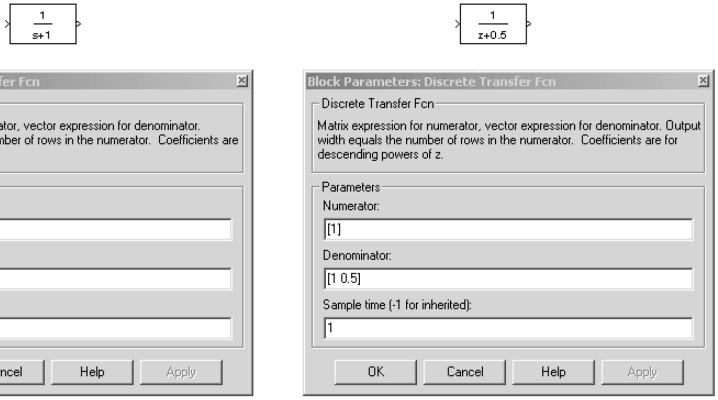
Method: tustin

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

For more help on the `c2d` function, type the following in the Command Window:

```
help c2d
```

The figure below shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the z-domain. The **Block Parameters** dialog box for each block is shown below the block.



Block Parameters: Transfer Fcn

Transfer Fcn

Matrix expression for numerator, vector expression for denominator. Output width equals the number of rows in the numerator. Coefficients are for descending powers of s.

Parameters

Numerator: [1]

Denominator: [1 1]

Absolute tolerance: auto

Block Parameters: Discrete Transfer Fcn

Discrete Transfer Fcn

Matrix expression for numerator, vector expression for denominator. Output width equals the number of rows in the numerator. Coefficients are for descending powers of z.

Parameters

Numerator: [1]

Denominator: [1 0.5]

Sample time (-1 for inherited): 1

Note If you want to recover exactly the original continuous parameter values after the Model Discretization session, you should enter parameters in the s-domain.

Configurable subsystem (Enter parameters in s-domain). Create multiple discretization candidates using s-domain values for the current selection. A configurable subsystem can consist of one or more blocks.

The **Location for block in configurable subsystem** field becomes active when this option is selected. This option allows you to either create a new configurable subsystem or overwrite an existing one.

Note The current directory must be writable in order to save the library or libraries for the configurable subsystem option.

Configurable subsystem (Enter parameters in z-domain). Create multiple discretization candidates in z-domain for the current selection. A configurable subsystem can consist of one or more blocks.

The **Location for block in configurable subsystem** field becomes active when this option is selected. This option allows you to either create a new configurable subsystem or overwrite an existing one.

Note The current directory must be writable in order to save the library or libraries for the configurable subsystem option.

Configurable subsystems are stored in a library containing the discretization candidates and the original continuous block. The library will be named `<model name>_disc_lib` and it will be stored in the current directory. For example a library containing a configurable subsystem created from the `f14` model will be named f`14_disc_lib`.

If multiple libraries are created from the same model, then the filenames will increment accordingly. For example, the second configurable subsystem library created from the `f14` model will be named `f14_disc_lib2`.

You can open a configurable subsystem library by right-clicking on the subsystem in the Simulink model and selecting **Link options -> Go to library block** from the pop-up menu.

Discretize the Blocks

To discretize blocks that are linked to a library, you must either discretize the blocks in the library itself or disable the library links in the model window.

You can open the library from the Model Discretizer by selecting **Load model** from the **File** menu.

You can disable the library links by right-clicking on the block and selecting **Link options -> Disable link** from the pop-up menu.

There are two methods for discretizing blocks.

Select Blocks and Discretize.

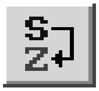
1. Select a block or blocks in the Model Discretizer tree view pane.

   To choose multiple blocks, press and hold the **Ctrl** button on the keyboard while selecting the blocks.

Note You must select blocks from the Model Discretizer tree view. Clicking on blocks in the Simulink editor does not select them for discretization.

2 Select **Discretize current block** from the **Discretize** menu if a single block is selected or select **Discretize selected blocks** from the **Discretize** menu if multiple blocks are selected.

   You can also discretize the current block by clicking the Discretize button, shown below.



Store the Discretization Settings and Apply Them to Selected Blocks in the Model.

1 Enter the discretization settings for the current block.

2 Click **Store Settings**.

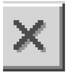
   This adds the current block with its discretization settings to the group of preset blocks.

3 Repeat steps 1 and 2, as necessary.

4 Select **Discretize preset blocks** from the **Discretize** menu.

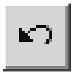
Deleting a Discretization Candidate from a Configurable Subsystem

You can delete a discretization candidate from a configurable subsystem by selecting it in the **Location for block in configurable subsystem** field and clicking the Delete button, shown below.



Undoing a Discretization

To undo a discretization, click the Undo discretization button, shown below.



Alternatively, you can select **Undo discretization** from the **Discretize** menu.

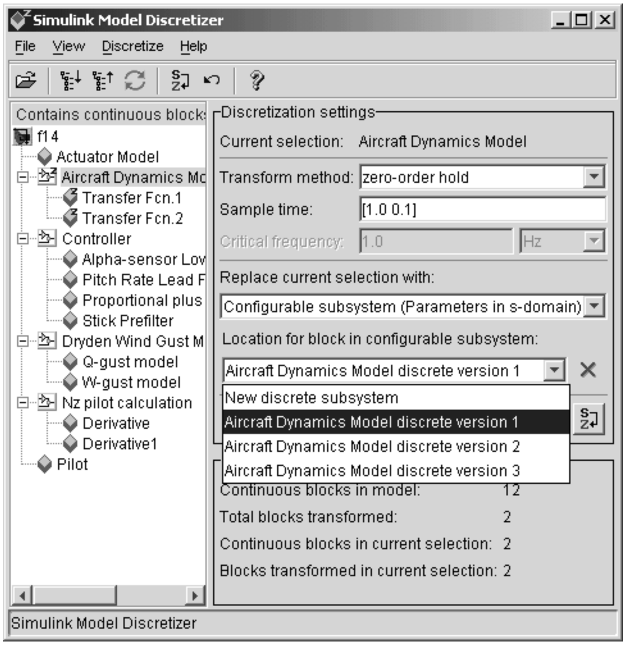
This operation undoes discretizations in the current selection and its children. For example, performing the undo operation on a subsystem will remove discretization from all blocks in all levels of the subsystem's hierarchy.

Viewing the Discretized Model

The Model Discretizer displays the model in a hierarchical tree view.

Viewing Discretized Blocks

The block's icon in the tree view becomes highlighted with a "**z**" when the block has been discretized. The figure below shows that the Aircraft Dynamics Model subsystem has been discretized into a configurable subsystem with three discretization candidates. The other blocks in this `f14` model have not been discretized.



Current selection: Aircraft Dynamics Model

Transform method: zero-order hold

Sample time: [1.0 0.1]

Critical frequency: 1.0 Hz

Replace current selection with: Configurable subsystem (Parameters in s-domain)

Location for block in configurable subsystem: Aircraft Dynamics Model discrete version 1

- New discrete subsystem
- Aircraft Dynamics Model discrete version 1
- Aircraft Dynamics Model discrete version 2
- Aircraft Dynamics Model discrete version 3

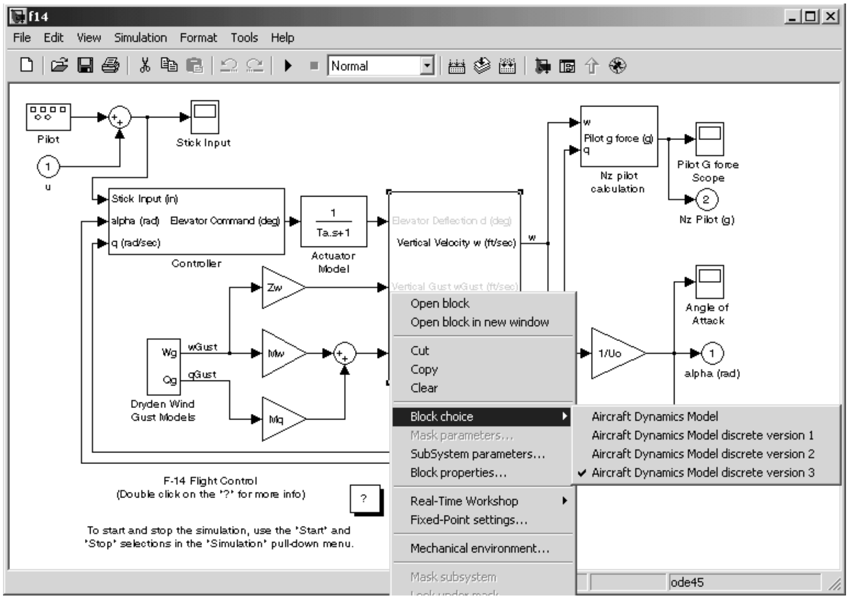
Continuous blocks in model: 12

Total blocks transformed: 2

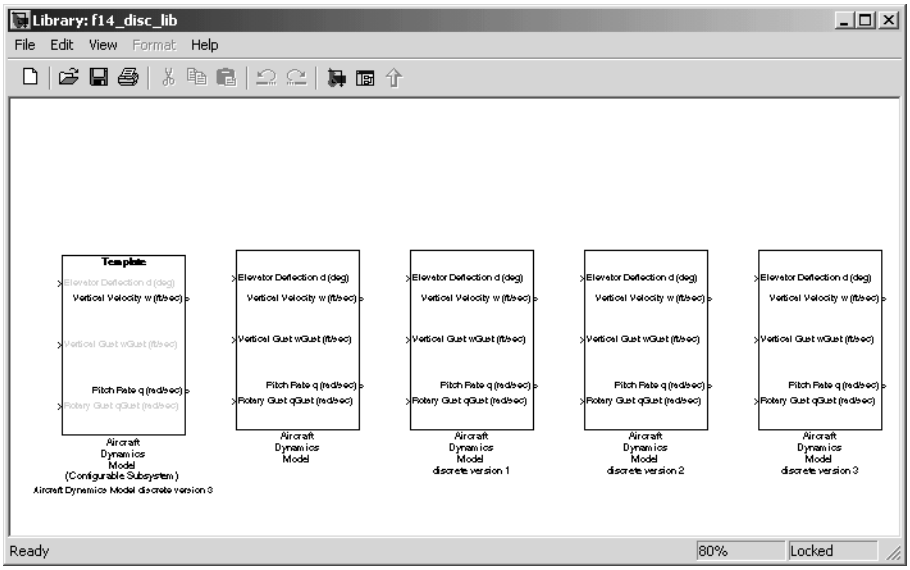
Continuous blocks in current selection: 2

Blocks transformed in current selection: 2

The following figure shows the Aircraft Dynamics Model subsystem of the `f14` demo model after discretization into a configurable subsystem containing the original continuous model and three discretization candidates.

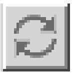


The following figure shows the library containing the Aircraft Dynamics Model configurable subsystem with the original continuous model and three discretization candidates.


Library: f14_disc_lib
File Edit View Format Help
Template
Elevator Deflection d (deg)
Vertical Velocity w (ft/sec)
Vertical Gust wGust (ft/sec)
Pitch Rate q (rad/sec)
Rotary Gust qGust (rad/sec)
Aircraft Dynamics Model (Configurable Subsystem)
Aircraft Dynamics Model discrete version 3
Aircraft Dynamics Model
Aircraft Dynamics Model discrete version 1
Aircraft Dynamics Model discrete version 2
Aircraft Dynamics Model discrete version 3
Ready
80%
Locked

Refreshing Model Discretizer View of the Model

To refresh the Model Discretizer's tree view of the model when the model has been changed, click the Refresh button, shown below.



Alternatively, you can select **Refresh** from the **View** menu.

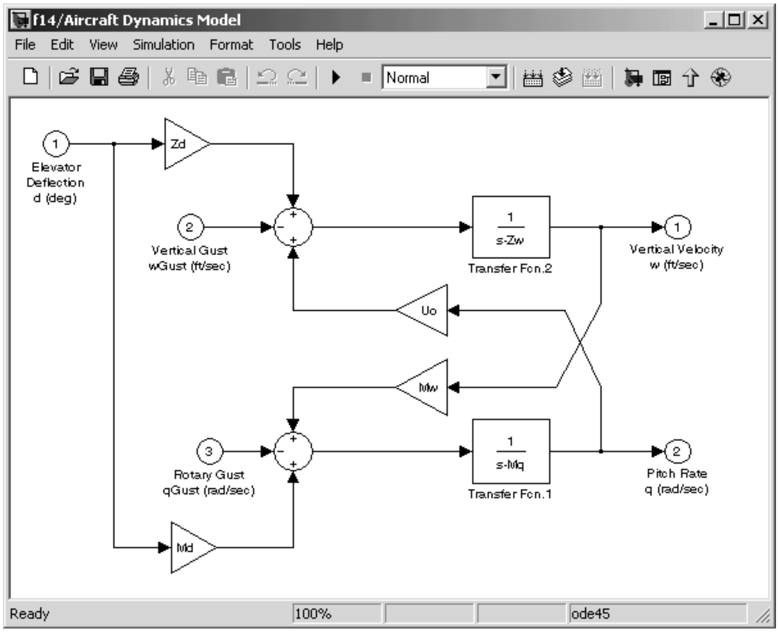
Discretizing Blocks from the Simulink Model

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

The procedure below shows how to replace a continuous Transfer Fcn block in the Aircraft Dynamics Model subsystem of the `f14` model with a discretized Transfer Fcn block from the Discretizing Library. The block is discretized in the s-domain with a zero-order hold transform method and a 2 second sample time.

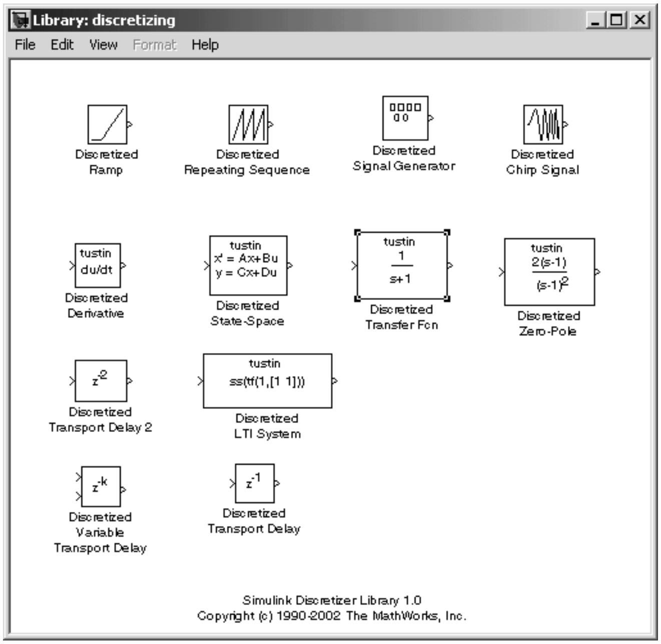
1 Open the `f14` model.

2 Open the Aircraft Dynamics Model subsystem in the `f14` model.


f14/Aircraft Dynamics Model
File Edit View Simulation Format Tools Help
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
1
s-Mq
Transfer Fcn.1
2
Pitch Rate q (rad/sec)
Md
Ready
100%
ode45

3 Open the Discretizing library window.

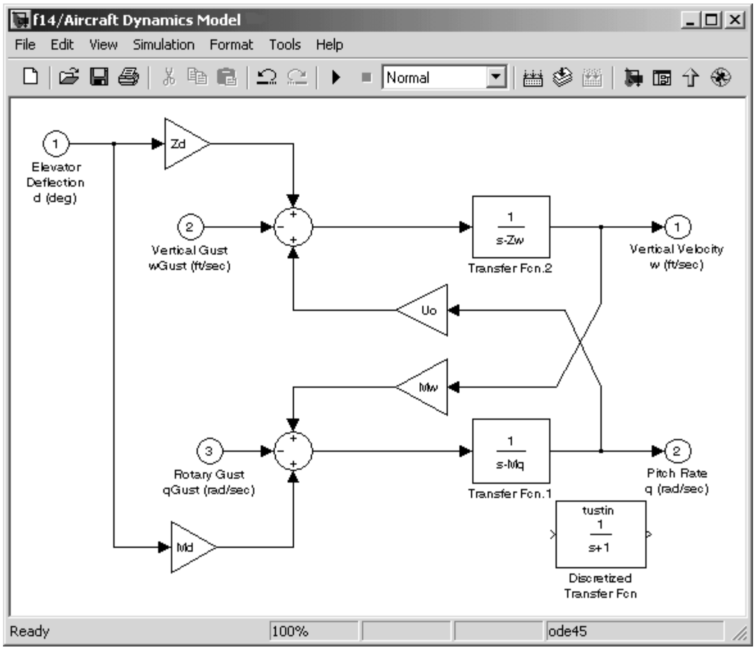
Enter `discretizing` at the MATLAB command prompt. The **Library: discretizing** window opens. This library contains s-domain discretized blocks.


Library: discretizing
File Edit View Format Help
Discretized Ramp
Discretized Repeating Sequence
Discretized Signal Generator
Discretized Chirp Signal
tustin du/dt
Discretized Derivative
tustin x' = Ax+Bu y = Cx+Du
Discretized State-Space
tustin 1/(s+1)
Discretized Transfer Fcn
tustin 2(s-1)/(s-1)^2
Discretized Zero-Pole
z^-2
Discretized Transport Delay 2
tustin ss(tf(1,[1 1]))
Discretized LTI System
z^-k
Discretized Variable Transport Delay
z^-1
Discretized Transport Delay
Simulink Discretizer Library 1.0
Copyright (c) 1990-2002 The MathWorks, Inc.

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

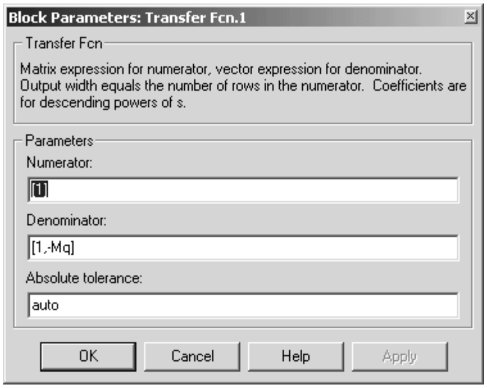
a Click the Discretized Transfer Fcn block in **Library: discretizing** window.

b Drag it into the **f14/Aircraft Dynamics Model** window.


f14/Aircraft Dynamics Model
File Edit View Simulation Format Tools Help
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
1
s-Mq
Transfer Fcn.1
2
Pitch Rate q (rad/sec)
tustin
1
s+1
Discretized Transfer Fcn
Md
Ready
100%
ode45

5 Open the parameter dialog box for the Transfer Fcn.1 block.

Double-click the Transfer Fcn.1 block in the **f14/Aircraft Dynamics Model** window. The **Block Parameters: Transfer Fcn.1** dialog box opens.


Block Parameters: Transfer Fcn.1

Transfer Fcn

Matrix expression for numerator, vector expression for denominator. Output width equals the number of rows in the numerator. Coefficients are for descending powers of s.

Parameters

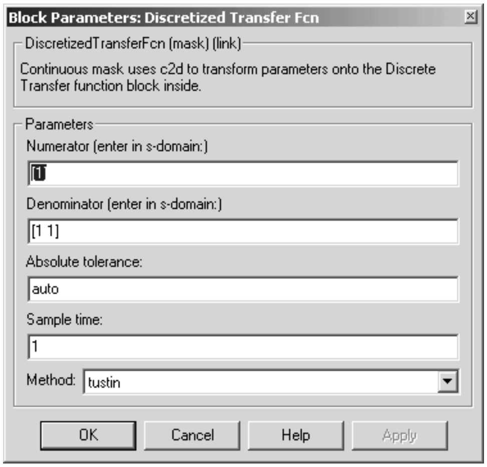
Numerator:
[1]

Denominator:
[1,-Mq]

Absolute tolerance:
auto

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

Double-click the Discretized Transfer Fcn block in the **f14/Aircraft Dynamics Model** window. The **Block Parameters: Discretized Transfer Fcn** dialog box opens.



Block Parameters: Discretized Transfer Fcn

DiscretizedTransferFcn (mask) (link)

Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside.

Parameters

Numerator (enter in s-domain:) [1]

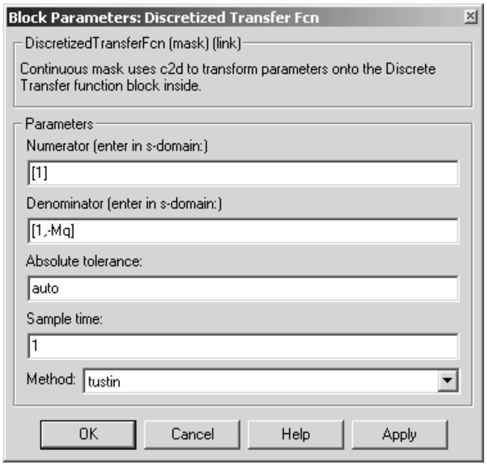
Denominator (enter in s-domain:) [1 1]

Absolute tolerance: auto

Sample time: 1

Method: tustin

Copy the parameter information from the Transfer Fcn.1 block's dialog box to the Discretized Transfer Fcn block's dialog box.



Block Parameters: Discretized Transfer Fcn

DiscretizedTransferFcn (mask) (link)

Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside.

Parameters

Numerator (enter in s-domain:) [1]

Denominator (enter in s-domain:) [1,-Mq]

Absolute tolerance: auto

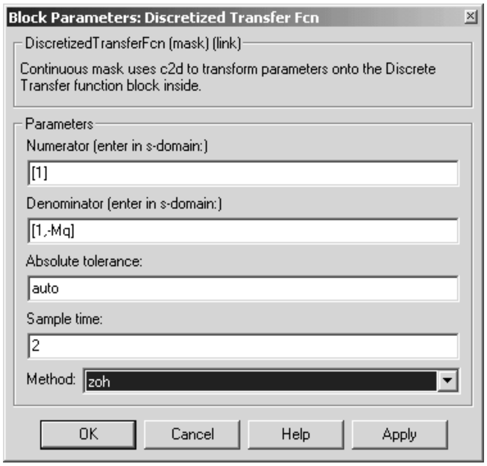
Sample time: 1

Method: tustin

7 Enter `2` in the **Sample time** field.

8 Select `zoh` from the **Method** drop-down list.

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



Block Parameters: Discretized Transfer Fcn

DiscretizedTransferFcn (mask) (link)

Continuous mask uses c2d to transform parameters onto the Discrete Transfer function block inside.

Parameters

Numerator (enter in s-domain:) [1]

Denominator (enter in s-domain:) [1,-Mq]

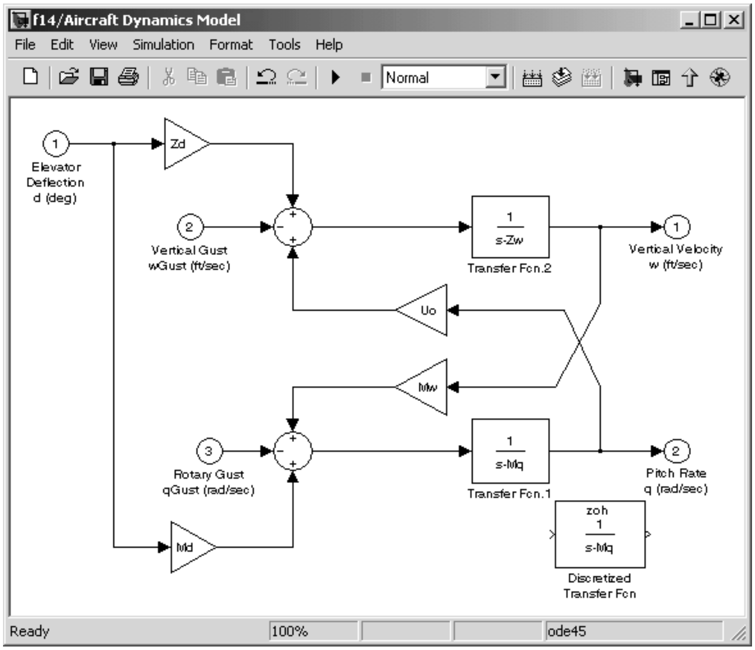
Absolute tolerance: auto

Sample time: 2

Method: zoh

9 Click **OK**.

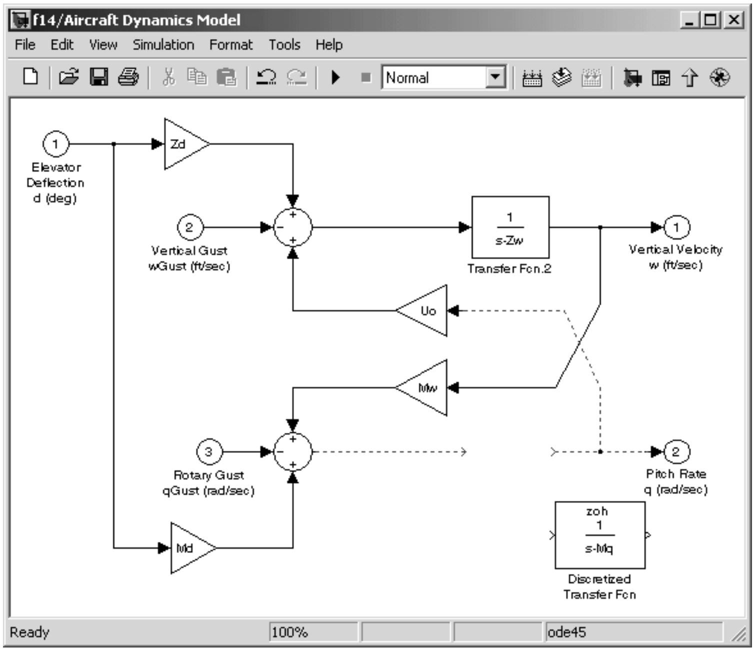
The **f14/Aircraft Dynamics Model** window now looks like this.


f14/Aircraft Dynamics Model
File Edit View Simulation Format Tools Help
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
1
s-Mq
Transfer Fcn.1
2
Pitch Rate q (rad/sec)
zoh
1
s-Mq
Discretized Transfer Fcn
Md
Ready
100%
ode45

10 Delete the original Transfer Fcn.1 block.

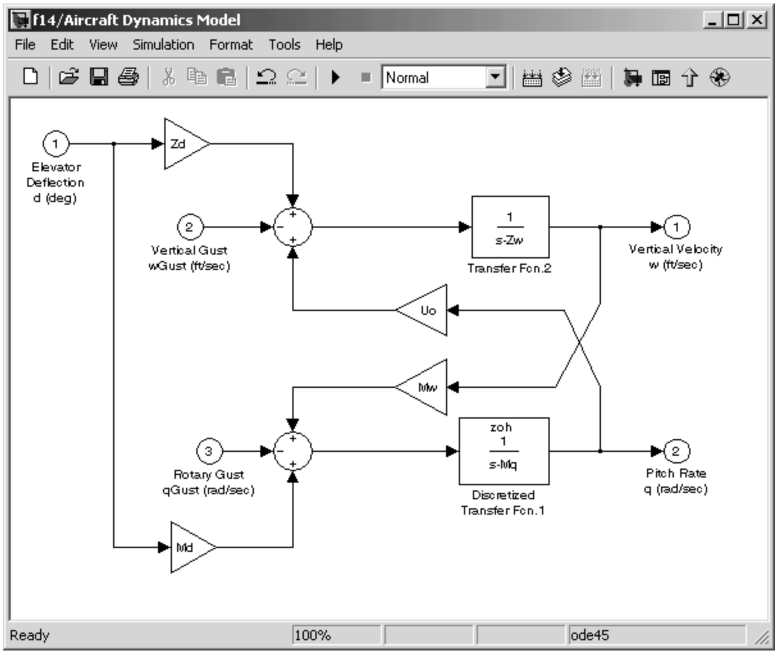
a Click the Transfer Fcn.1 block.

b Press the **Delete** key. The **f14/Aircraft Dynamics Model** window now looks like this.


f14/Aircraft Dynamics Model
File Edit View Simulation Format Tools Help
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
2
Pitch Rate q (rad/sec)
zoh
1
s-Mq
Discretized Transfer Fcn
Md
Ready
100%
ode45

11 Add the Discretized Transfer Fcn block to the model.

- **a** Click the Discretized Transfer Fcn block.
- **b** Drag the Discretized Transfer Fcn block into position to complete the model. The **f14/Aircraft Dynamics Model** window now looks like this.


f14/Aircraft Dynamics Model
File
Edit
View
Simulation
Format
Tools
Help
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
zoh
1
s-Mq
Discretized Transfer Fcn.1
2
Pitch Rate q (rad/sec)
Md
Ready
100%
ode45

Discretizing a Model from the MATLAB Command Window

Use the sldiscmdl function to discretize Simulink models from the MATLAB Command Window. You can specify the transform method, the sample time, and the discretization method with the sldiscmdl function.

For example, the following command discretizes the f14 model in the s-domain with a 1 second sample time using a zero-order hold transform method.

```
sldiscmdl('f14',1.0,'zoh')
```

For more information on the sldiscmdl function, see the reference pages in Simulink Model Construction Commands.

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, port, 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 the path changes.

Tracing Callbacks

Callback tracing allows you to determine the callbacks Simulink invokes and in what order Simulink invokes them when you open or simulate a model. To enable callback tracing, select the **Callback tracing** option on the Simulink **Preferences** dialog box (see "Setting Simulink Preferences" on page 1-18) or execute `set_param(0, 'CallbackTracing', 'on')`. This option causes Simulink to list callbacks in the MATLAB Command Window as they are invoked.

Creating Model Callback Functions

You can create model callback functions interactively or programmatically. Use the **Callbacks** pane of the model's **Model Properties** dialog box (see "Callbacks Pane" on page 4-106)to create model callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the model parameter corresponding to the callback (see "Model Callback Parameters" on page 4-91).

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

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

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

Model Callback Parameters

The following table lists the model parameters used to specify model callback routines and indicates when the corresponding callback routines are executed.

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.

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

Creating Block Callback Functions

You can create model callback functions interactively or programmatically. Use the **Callbacks** pane of the model's **Block Properties** dialog box (see "Callbacks Pane" on page 5-11) to create model callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the block parameter corresponding to the callback (see "Block Callback Parameters" on page 4-92).

Note A callback for a masked subsystem cannot directly reference the parameters of the masked subsystem (see "About Masks" on page 12-2). The reason? Simulink evaluates block callbacks in a model's base workspace whereas the mask parameters reside in the masked subsystem's private workspace. A block callback, however, can use `get_param` to obtain the value of a mask parameter, e.g., `get_param(gcb, 'gain')`, where `gain` is the name of a mask parameter of the current block.

Block Callback Parameters

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

Parameter	When Executed
`ClipboardFcn`	When the block is copied or cut to the system clipboard.
`CloseFcn`	When the block is closed using the `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.

Parameter	When Executed
DeleteChildFcn	After a block is deleted from a subsystem.
DeleteFcn	Before a block is deleted, e.g., when the user deletes the block or closes the model containing the block. This callback is recursive for Subsystem blocks.
DestroyFcn	When the 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 the 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.
OpenFcn	When the block is opened. This parameter is generally used with Subsystem blocks. The routine is executed when you double-click the block or when an `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` in the "Model Creation Commands" section of the Simulink online Help).

Parameter	When Executed
`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 "S-Function Callback Methods" in *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 "S-Function Callback Methods" in *Writing S-Functions* for more information.
`UndoDeleteFcn`	When a block delete is undone.

Port Callback Parameters

Block input and output ports have a single callback parameter, ConnectionCallback. This parameter allows you to set callbacks on ports that are triggered every time the connectivity of those ports changes. Examples of connectivity changes include deletion of blocks connected to the port and deletion, disconnection, or connection of branches or lines to the port.

Use get_param to get the port handle of a port and set_param to set the callback on the port. For example, suppose the currently selected block has a single input port. The following code fragment sets foo as the connection callback on the input port.

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

The first argument of the callback function must be a port handle. The callback function can have other arguments (and a return value) as well. For example, the following is a valid callback function signature.

```
function foo(port, otherArg1, otherArg2)
```

Working with Model Workspaces

Simulink provides each model with its own workspace for storing variable values. The model workspace is similar to the base MATLAB workspace except that

- Variables in a model's workspace are visible only in the scope of the model.

  If both the MATLAB workspace and a model workspace define a variable of the same name (and the variable does not appear in any intervening masked subsystem or referenced model workspaces), Simulink uses the value of the variable in the model workspace. A model's workspace effectively provides it with its own name space, allowing you to create variables for the model without risk of conflict with other models.
- When the model is loaded, the workspace is initialized from a data repository.

  The data repository can be a MAT-file or data or M-code stored in the model file itself (see "Data source" on page 4-99 for more information).
- You can interactively reload and save the current data repository.
- The only kind of Simulink data object that a model workspace can contain is `Simulink.Parameter`.
- In general, parameter variables in a model workspace are not tunable.

  However, you can declare variables in a model's workspace as model arguments. You can tune the arguments of a model referenced by another model in the base workspace of the parent model.

Note When resolving references to variables used in a referenced model, i.e., a model referenced by a Model block (see "Referencing Models" on page 4-53), Simulink resolves the referenced model's variables as if the parent model does not exist. For example, suppose a referenced model references a variable that is defined in both the parent model's workspace and in the MATLAB workspace but not in the referenced model's workspace. In this case, Simulink uses the variable defined in the MATLAB workspace.

Changing a Model Workspace

The procedure for modifying a workspace depends on the workspace's data source. See the following sections for more information.

- "Changing a Workspace Whose Source is the Model File" on page 4-97
- "Changinging a Workspace Whose Source Is a MAT-File" on page 4-98
- "Changing a Workspace Whose Source Is M-Code" on page 4-98

Changing a Workspace Whose Source is the Model File

If a model workspace's data source is data stored in the model, you can use the Model Explorer (see "The Model Explorer" on page 9-2) or MATLAB commands to change the model's workspace.

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

To change the value of a model workspace variable, select the workspace, then select the variable in the Model Explorer's **Contents** pane and edit the value displayed in the **Contents** pane or in the Model Explorer's object **Dialog** pane. To delete a model workspace variable, select the variable in the **Contents** pane and select **Delete** from the Model Explorer's **Edit** menu or toolbar. To save the changes, save the model.

To use MATLAB commands to change a model workspace, first get the workspace for the currently selected model:

```
hws = get_param(bdroot, 'modelworkspace');
```

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

- `whos(hws)`
- `assignin(hws, 'varname', value)`
- `evalin(hws, 'expression')`
- `clear(hws)`

- `clear(hws, 'var1', 'var2', ...)`
- `save(hws)`
- `reload(hws)`

Note You can use dot notation to invoke these methods on a workspace object, i.e., `hws.assignin('varname', value)` is equivalent to `assignin(hws, 'varname', value).`

The following example uses MATLAB commands to create, save, and load variables in the currently selected model's workspace.

```
a = get_param(bdroot, 'modelworkspace');
a.DataSource = 'MAT-File';
a.RelativePath = 'on';
a.FileName = 'params’;
a.assignin(a,'pitch’, -10);
a.assignin('roll', 30);
a.assignin('yaw’, -2);
a.save;
a.reload;
```

Changinging a Workspace Whose Source Is a MAT-File

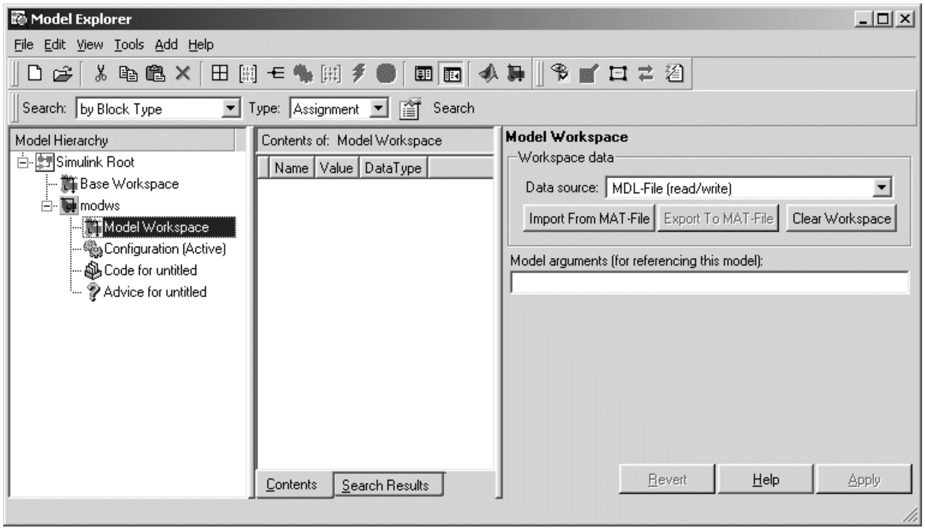
You cannot use the Model Explorer or model workspace commands to modify a model workspace whose source is a MAT-file. To change a model workspace whose source is a MAT-file, you must recreate the MAT-file and reload it. An easy way to do this is to use the model workspace's dialog box (see "Model Workspace Dialog Box" on page 4-99) to change the workspace's source temporarily to the model, make your changes, using the Model Explorer's object creation and editing controls, export the changes to the MAT-file, and then change the workspace's source back to the MAT-file.

Changing a Workspace Whose Source Is M-Code

Simulink does not allow you to use the Model Explorer's object creation and editing controls or model workspace commands to change a model workspace whose source is M-code. To change such a workspace, you must use the workspace's dialog box to edit the M-code and reinitialize the workspace (see "M-Code Source Controls" on page 4-101).

Model Workspace Dialog Box

The Model Workspace Dialog Box enables you to specify a model workspace's source and model reference arguments. To display the dialog box, select the model workspace in the Model Explorer's Model Hierarchy pane.



Search: by Block Type

Type: Assignment

Contents of: Model Workspace

Data source: MDL-File (read/write)

The dialog box contains the following controls.

Data source

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

- `Mdl-File`

  Specifies that the data source is the model itself. Selecting this option causes additional controls to appear (see "MDL-File Source Controls" on page 4-100).

- `MAT-File`

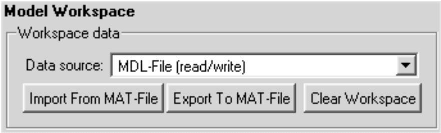
  Specifies that the data source is a MAT file. Selecting this option causes additional controls to appear (see "MAT-File Source Controls" on page 4-100).

- `M-code`

  Specifies that the data source is M code stored in the model file. Selecting this option causes additional controls to appear (see “M-Code Source Controls” on page 4-101).

MDL-File Source Controls

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



Data source: MDL-File (read/write)

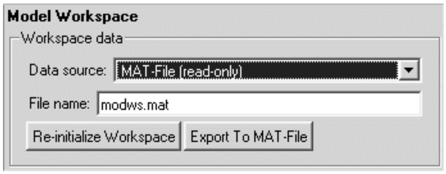
Import From MAT-File. This button lets you import data from a MAT-file. Selecting the button causes Simulink to display a file selection dialog box. Use the dialog box to select the MAT file that contains the data you want to import.

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

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

MAT-File Source Controls

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



Data source: MAT-File (read-only)

File name: modws.mat

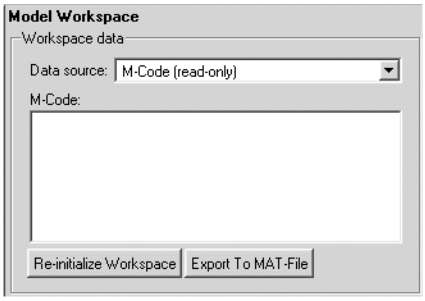
File name. File name or path name of the MAT file that is the data source for the selected workspace. If a file name, the name must reside on the MATLAB path.

Re-initialize Workspace. Clears the workspace and reloads it from the specified MAT file.

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

M-Code Source Controls

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



Data source: M-Code (read-only)

M-Code. Specifies M code that initializes the selected workspace in this field. To change the initialization code, edit this field and select the **Accept** button on the dialog box to confirm the changes and execute the modified code.

Re-initialize Workspace. Clears the workspace and executes the contents of the **M-Code** field.

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

Model Arguments

This field allows you to specify arguments that can be passed to instances of this model referenced by another model. See “Using Model Arguments” on page 4-57 for more information.

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 Box” on page 4-105).
- 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.
- 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 source control system. See “Interfacing with Source Control Systems” in the MATLAB documentation for more information.

Specifying the Current User

When you create or update 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, might 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')
```

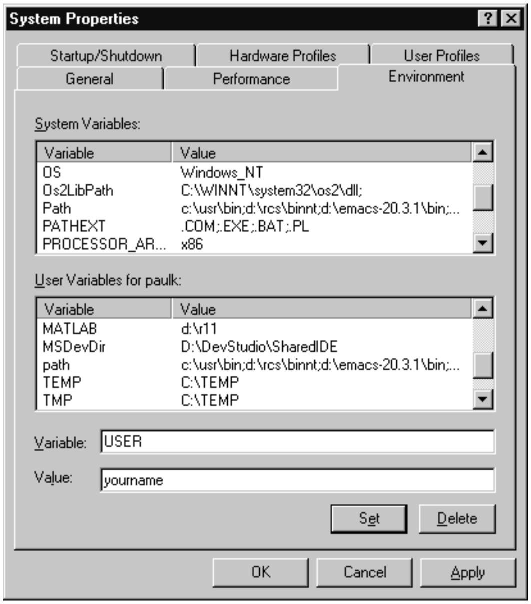
at the MATLAB command line to determine whether the USER environment variable exists on your Windows system. If not, you must set it yourself. On Windows 98, set the value by entering the following line

```
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 and 2000, use the **Environment** variables pane of the **System Properties** dialog box to set the USER environment variable (if it is not already defined).



System Variables:

Variable	Value
OS	Windows_NT
Os2LibPath	C:\WINNT\system32\os2\dll;
Path	c:\usr\bin;d:\rcs\binnt;d:\emacs-20.3.1\bin;...
PATHEXT	.COM;.EXE;.BAT;.PL
PROCESSOR_AR...	x86

User Variables for paulk:

Variable	Value
MATLAB	d:\r11
MSDevDir	D:\DevStudio\SharedIDE
path	c:\usr\bin;d:\rcs\binnt;d:\emacs-20.3.1\bin;...
TEMP	C:\TEMP
TMP	C:\TEMP

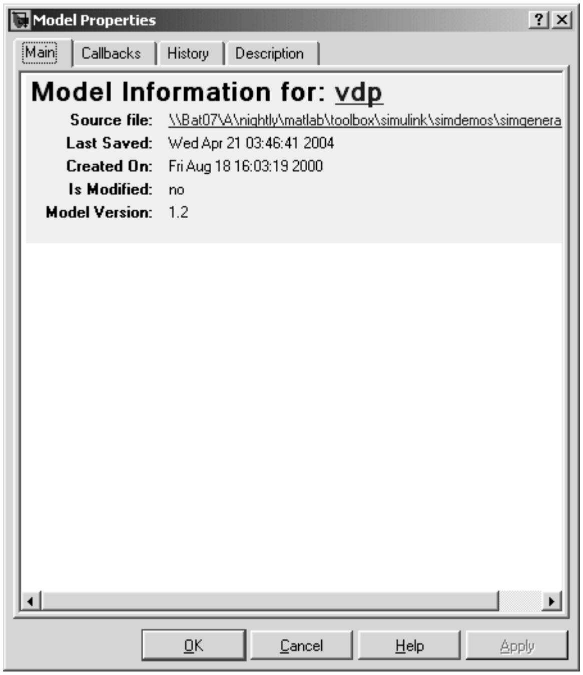
Variable: USER

Value: yourname

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

Model Properties Dialog Box

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



Model Information for: vdp

Source file: \\Bat07\A\nightly\matlab\toolbox\simulink\simdemos\simgenera
Last Saved: Wed Apr 21 03:46:41 2004
Created On: Fri Aug 18 16:03:19 2000
Is Modified: no
Model Version: 1.2

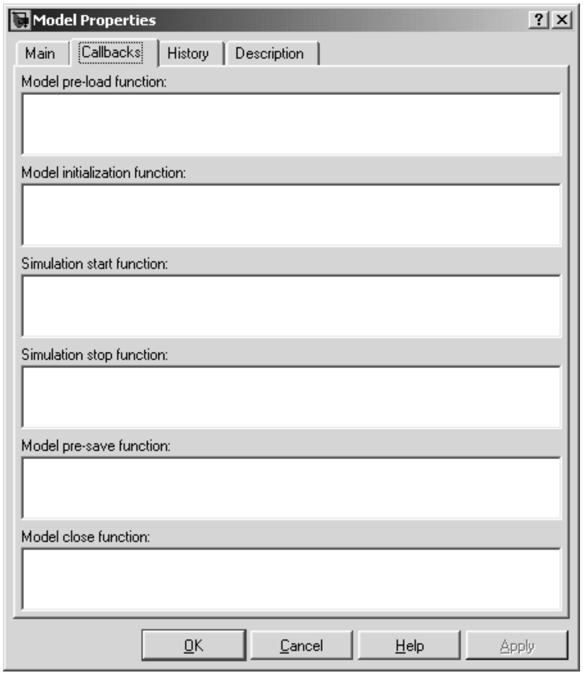
The dialog box includes the following panes.

Main Pane

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

Callbacks Pane

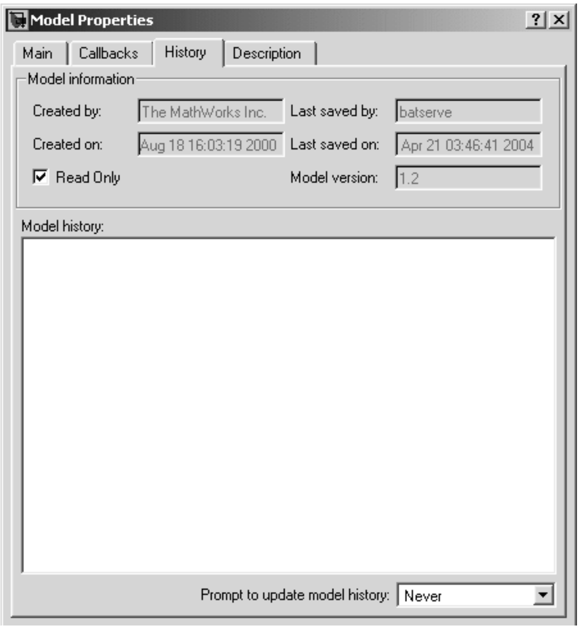
The **Callbacks** pane lets you specify functions to be invoked by Simulink at specific points in the simulation of the model.



Enter the names of any callback functions you want to be invoked in the appropriate fields. See “Creating Model Callback Functions” on page 4-90 for information on the callback functions listed on this pane.

History Pane

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



Created by: The MathWorks Inc.
Last saved by: batserve
Created on: Aug 18 16:03:19 2000
Last saved on: Apr 21 03:46:41 2004
☑ Read Only
Model version: 1.2
Prompt to update model history: Never

The **History** pane has two panels: the **Model information** panel and the **Model History** panel.

Version Information Panel

The contents of the **Version information** panel depend on the item selected in the list at the top of the panel. When `View current values` is selected, the panel shows the following fields.

Created by. 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.

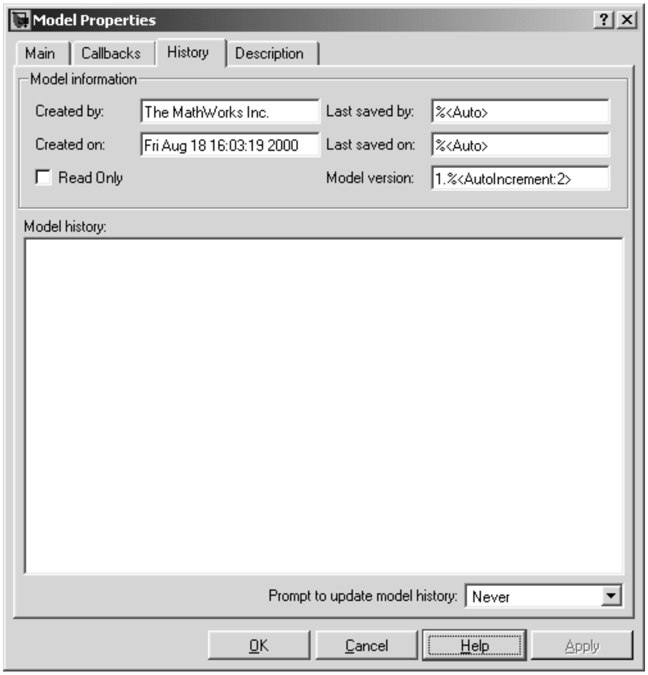
Created on. Date and time this model was created.

Model version. Version number for this model. You cannot edit this field.

Last saved by. Name of the person who last saved this model. Simulink sets the value of this parameter to the value of the `USER` environment variable when you save a model.

Last saved date. Date that this model was last saved. Simulink sets the value of this parameter to the system date and time whenever you save a model.

Read Only. Deselecting this option shows the format strings for each of the fields listed when `the` option is selected.



Model information

Created by: The MathWorks Inc.

Last saved by: %<Auto>

Created on: Fri Aug 18 16:03:19 2000

Last saved on: %<Auto>

□ Read Only

Model version: 1.%<AutoIncrement:2>

Model history:

Prompt to update model history: Never

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

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

as

```
1.2
```

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

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

becomes

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

and Simulink reports the model version number as `1.3`.

Last saved by. Enter a format string describing the format used to display the **Last saved by** value in the **History** pane and the **ModifiedBy** entry in the history log and Model Info blocks. The value of this field can be any string. The string can include the tag `%<Auto>`. Simulink replaces occurrences of this tag with the current value of the `USER` environment variable.

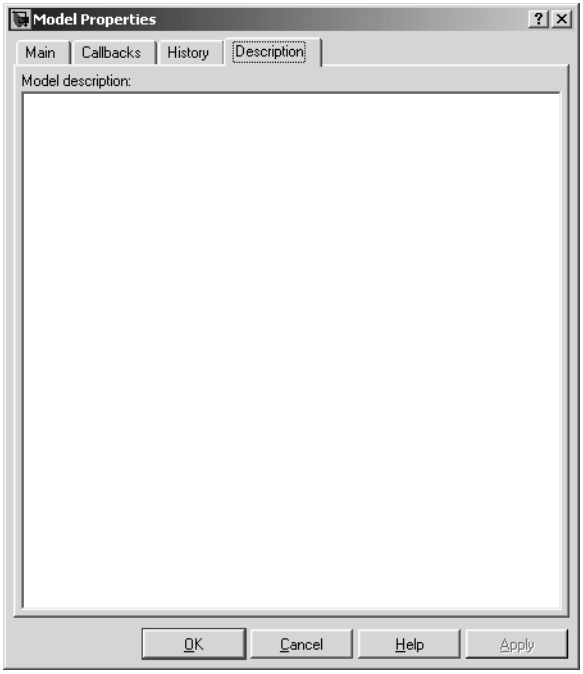
Last saved on. Enter a format string describing the format used to display the **Last saved on** date in the **History** pane and the **ModifiedOn** entry in the history log and the in Model Info blocks. The value of this field can be any string. The string can contain the tag `%<Auto>`. Simulink replaces occurrences of this tag with the current date and time.

Model History Panel

The model history panel contains a scrollable text field and an option list. The text field displays the history for the model in a scrollable text field. To change the model history, edit the contents of this field. The option list allows you to enable or disable the Simulink model history feature. To enable the history feature, select `When saving model` from the **Prompt to update model history** list. This causes Simulink to prompt you to enter a comment when saving the model. Typically you would enter any changes that you have made to the model since the last time you saved it. Simulink stores this information in the model's change history log. See "Creating a Model Change History" on page 4-110 for more information. To disable the change history feature, select `Never` from the **Prompt to update model history** list.

Model Description Pane

This pane allows you to enter a description of the model.

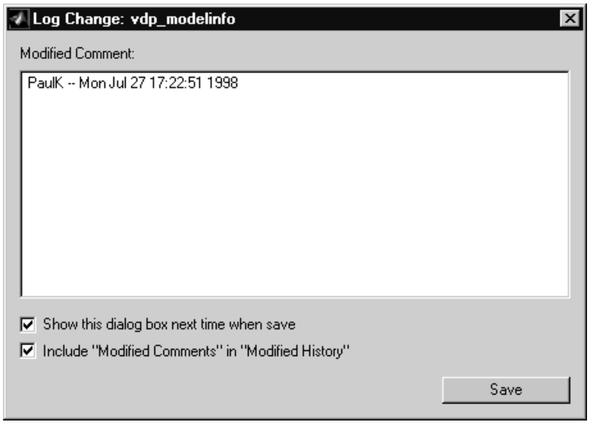


Creating a Model Change History

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 `When saving model` from the **Prompt to update model history** list on the **History** pane on the Simulink **Model Properties** dialog box. The next time you save the model, Simulink displays a **Log Change** dialog box.



PaulK -- Mon Jul 27 17:22:51 1998

☑ Show this dialog box next time when save

☑ Include "Modified Comments" in "Modified History"

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

Version Control Properties

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.

Property	Description
`ModifiedByFormat`	Format of the `ModifiedBy` parameter. Value can be any string. The string can include the tag `%<Auto>`. Simulink replaces the tag with the current value of the `USER` environment variable.
`ModifiedDate`	Date modified.
`ModifiedDateFormat`	Format of the `ModifiedDate` parameter. Value can be any string. The string can include the tag `%<Auto>`. 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.

Working with Blocks

This section explores the following block-related topics.

About 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.

Block Name	Condition Under Which Block Is Virtual
Bus Selector	Always virtual.
Demux	Always virtual.
Enable	Virtual unless connected directly to an Outport block.
From	Always virtual.
Goto	Always virtual.
Goto Tag Visibility	Always virtual.
Ground	Always virtual.

Block Name	**Condition Under Which Block Is 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.
Signal Specification	Always virtual.
Subsystem	Virtual unless the block is conditionally executed and/or the block's **Treat as Atomic Unit** option is selected.
Terminator	Always virtual.
Trigger	Virtual when the outport port is *not* present.

Editing Blocks

The Simulink Editor allows you to cut and paste blocks in and between models.

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, 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 5-36 for more information.

Note Simulink hides the names of Sum, Mux, Demux, Bus Creator, 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 indicate their respective functions.)

You can also copy blocks by using the **Copy** and **Paste** commands from the **Edit** menu:

1. Select the block you want to copy.
2. Choose **Copy** from the **Edit** menu.
3. Make the target model window the active window.
4. Choose **Paste** from the **Edit** menu.

Simulink assigns a name to each copied block. If it is the first block of its type in the model, its name is the same as its name in the source window. For example, if you copy the Gain block from the Math library into your model window, the name of the new block is Gain. If your model already contains a block named Gain, Simulink adds a sequence number to the block name (for example, Gain1, Gain2). You can rename blocks; see “Manipulating Block Names” on page 5-15.

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, enter

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

For example, to change the grid spacing to 20 pixels, enter

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

For either of the above commands, you can also select the model, then enter `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 4-3.

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.

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.

Setting Block Parameters

All Simulink blocks have a common set of parameters, called block properties, that you can set (see “Common Block Parameters” in the online Simulink Help). See “Block Properties Dialog Box” on page 5-9 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” in the online Simulink reference). 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 the “Simulink Blocks” in the online Simulink Help.

You can set any block parameter, using the Simulink `set_param` command. See `set_param` in the online Simulink Help 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.Parameter` data objects as parameters (see “Working with Data Objects” on page 7-10).

Tuning Parameters

Simulink lets you change the values of block parameters during simulation. You can use a block's parameter dialog box or the MATLAB Command Line to tune block parameters. To use the block's parameter dialog box, open the block's parameter dialog box, change the value displayed in the dialog box, and click the dialog box's **OK** or **Apply** button. You can use the `set_param` command to change the value of a variable at the MATLAB Command Line during simulation. Or, if the model uses a MATLAB workspace variable to specify the parameter's value, you can change the parameter's value by assigning a new value to the variable. In either case, you must update the model's block diagram for the change to take effect.

Inlining Parameters

You can declare some or all of a model's parameters as nontunable. Declaring a parameter as nontunable allows the Real-Time Workshop to include the parameter as a constant in code generated from the model, an optimization known as inlining a parameter. To inline all of a model's parameters, select the **Inline parameters** option on the **Optimization** pane of the model's active configuration set.

To inline some but not all of a model's parameters, you must first declare all of the parameters as inlined by selecting the **Inline parameters** option on the active set's **Optimization** pane. You must then specify the exceptions, using either the **Model Parameter Configuration** dialog box (see "Model Parameter Configuration Dialog Box" on page 10-47) or `Simulink.Parameter` objects.

Using Parameter Objects to Specify Parameter Tunability

To declare a parameter to be tunable even when the **Inline parameters** option is set, use an instance of `Simulink.Parameter` class to specify the parameter's value and set the parameter object's `RTWInfo.StorageClass` property to any value but `'Auto'` (the default).

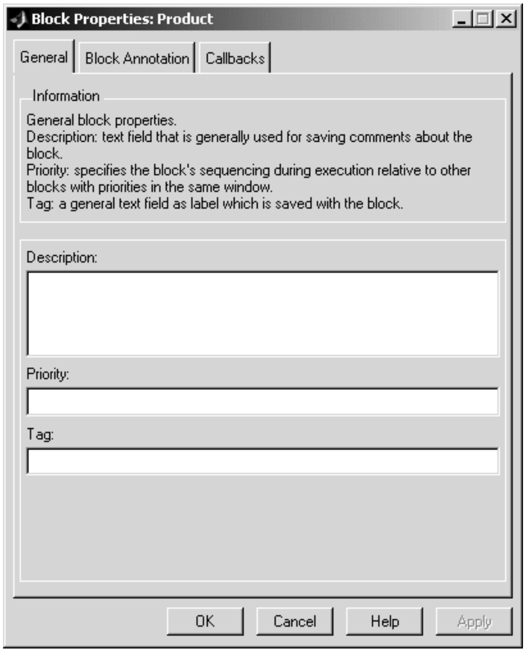
```
gain.RTWInfo.StorageClass = 'SimulinkGlobal';
```

If you set the `RTWInfo.StorageClass` property to any value other than `Auto`, you should not include the parameter in the tunable parameters table in the model's **Model Parameter Configuration** dialog box.

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.

Block Properties Dialog Box

This dialog box lets you set a block's properties. To display this dialog, select the block in the model window and then select **Block Properties** from the **Edit** menu.



Information

General block properties.
Description: text field that is generally used for saving comments about the block.
Priority: specifies the block's sequencing during execution relative to other blocks with priorities in the same window.
Tag: a general text field as label which is saved with the block.

Description:

Priority:

Tag:

The dialog box contains the following tabbed panes.

General Pane

This pane allows you to set the following properties.

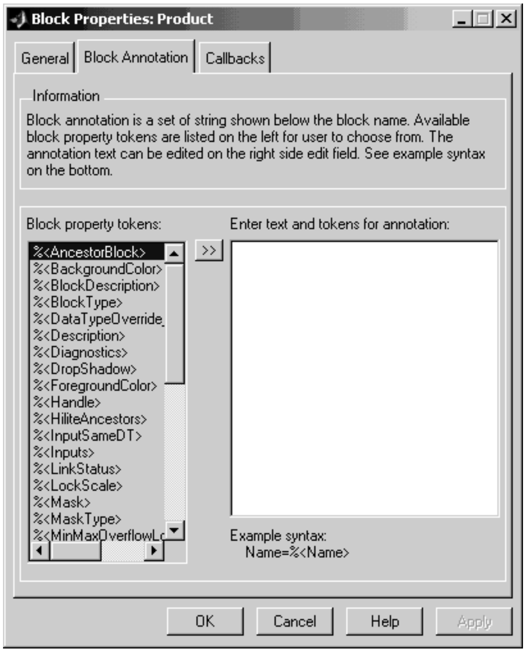
Description. Brief description of the block's purpose.

Priority. Execution priority of this block relative to other blocks in the model. See "Assigning Block Priorities" on page 5-20 for more information.

Tag. Text that is assigned to the block's Tag parameter and saved with the block in the model. You can use the tag to create your own block-specific label for a block.

Block Annotation Pane

The block annotation pane allows you to display the values of selected parameters of a block in an annotation that appears beneath the block's icon.



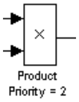
Block annotation is a set of string shown below the block name. Available block property tokens are listed on the left for user to choose from. The annotation text can be edited on the right side edit field. See example syntax on the bottom.

Example syntax:
  Name=%<Name>

Enter the text of the annotation in the text field that appears on the right side of the pane. The text can include block property tokens, for example

```
%<Name>
Priority = %<priority>
```

of the form `%<param>` where `param` is the name of a parameter of the block. When displaying the annotation, Simulink replaces the tokens with the values of the corresponding parameters, e.g.,

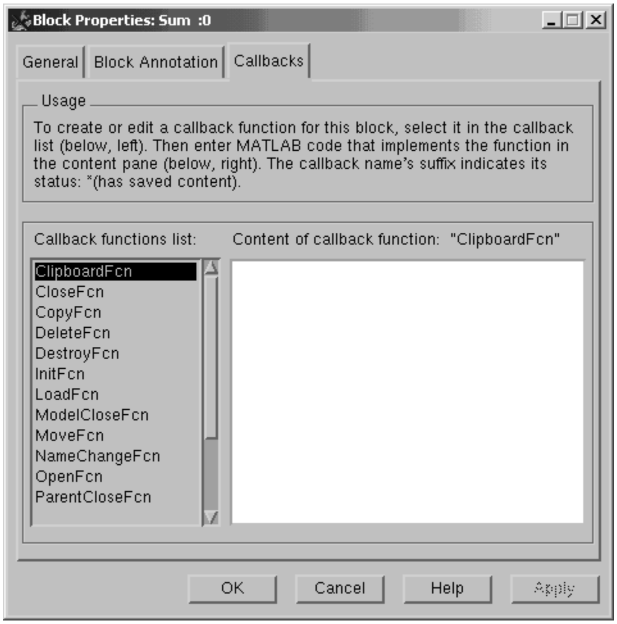


The block property tag list on the left side of the pane lists all the tags that are valid for the currently selected block. To include one of the listed tags in the annotation, select the tag and then click the button between the tag list and the annotation field.

You can also create block annotations programmatically. See “Creating Block Annotations Programmatically” on page 5-12.

Callbacks Pane

The **Callbacks Pane** allows you to specify implementations for a block’s callbacks (see “Using Callback Routines” on page 4-90).



To create or edit a callback function for this block, select it in the callback list (below, left). Then enter MATLAB code that implements the function in the content pane (below, right). The callback name’s suffix indicates its status: *(has saved content).

Callback functions list:

Content of callback function: "ClipboardFcn"

To specify an implementation for a callback, select the callback in the callback list on the left side of the pane. Then enter MATLAB commands that implement the callback in the right-hand field. Click **OK** or **Append** to save the change. Simulink appends an asterisk to the name of the saved callback to indicate that it has been implemented.

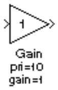
Creating Block Annotations Programmatically

You can use a block's `AttributesFormatString` parameter to display selected parameters of a block beneath the block as an "attributes format string," i.e. a string that specifies values of the block's attributes (parameters). The "Model and Block Parameters" section in the online Simulink reference describes the parameters that a block can have. Use the Simulink `set_param` command to set this parameter to the desired attributes format string.

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

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

for a Gain block displays



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

State Properties Dialog Box

The **State Properties** dialog box allows you to specify code generation options for certain blocks with discrete states. To get help on using this dialog box, you must install the Real-Time Workshop documentation. See "Block States:

Storing and Interfacing” in the online documentation for Real-Time Workshop for more information.

Changing a Block's Appearance

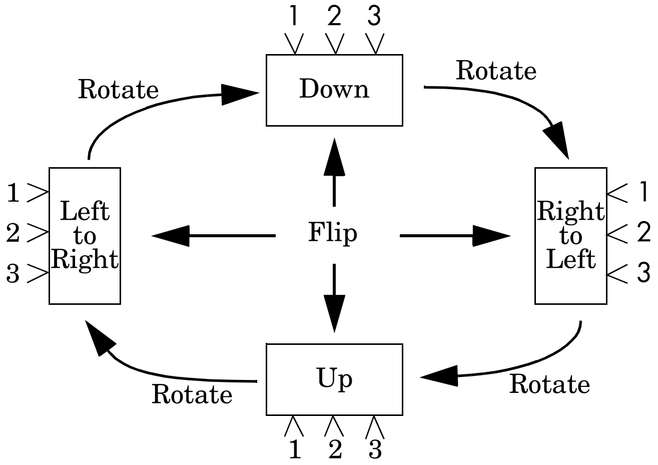
The Simulink Editor allows you to change the size, orientation, color, and label location of a block in a block diagram.

Changing the Orientation of a Block

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.

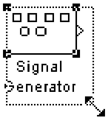


Resizing a Block

To change the size of a block, select it, then drag any of its selection handles. While you hold down the mouse button, a dotted rectangle shows the new block size. When you release the mouse button, the block is resized.

For example, the 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.



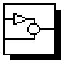
Displaying Parameters Beneath a Block

You can cause Simulink to display one or more of a block's parameters beneath the block. You specify the parameters to be displayed in the following ways:

- By entering an attributes format string in the **Attributes format string** field of the block's **Block Properties** dialog box (see "Block Properties Dialog Box" on page 5-9)
- By setting the value of the block's `AttributesFormatString` property to the format string, using `set_param`

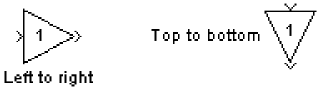
Using Drop Shadows

You can add a drop shadow to a block by selecting the block, then choosing **Show Drop Shadow** from the **Format** menu. When you select a block with a drop shadow, the menu item changes to **Hide Drop Shadow**. The figure below shows a Subsystem block with a drop shadow.



Manipulating Block Names

All block names in a model must be unique and must contain at least one character. By default, block names appear below blocks whose ports are on the sides, and to the left of blocks whose ports are on the top and bottom, as the following figure shows.



Note Simulink commands interprets a forward slash, i.e., /, as a block path delimiter. For example, the path `vdp/Mu` designates a block named `Mu` in the model named `vdp`. Therefore, avoid using forward slashes (/) in block names to avoid causing Simulink to interpret the names as paths.

Changing Block Names

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

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

When you click the pointer anywhere else in the model or take any other action, the name is accepted or rejected. If you try to change the name of a block to a name that already exists or to a name with no characters, Simulink displays an error message.

You can modify the font used in a block name by selecting the block, then choosing the **Font** menu item from the **Format** menu. Select a font from the **Set Font** dialog box. This procedure also changes the font of any text that appears inside the block.

You can cancel edits to a block name by choosing **Undo** from the **Edit** menu.

Note If you change the name of a library block, all links to that block become unresolved.

Changing the Location of a Block Name

You can change the location of the name of a selected block in two ways:

- By dragging the block name to the opposite side of the block.
- By choosing the **Flip Name** command from the **Format** menu. This command changes the location of the block name to the opposite side of the block.

For more information about block orientation, see "Changing the Orientation of a Block" on page 5-14.

Changing Whether a Block Name Appears

To change whether the name of a selected block is displayed, choose a menu item from the **Format** menu:

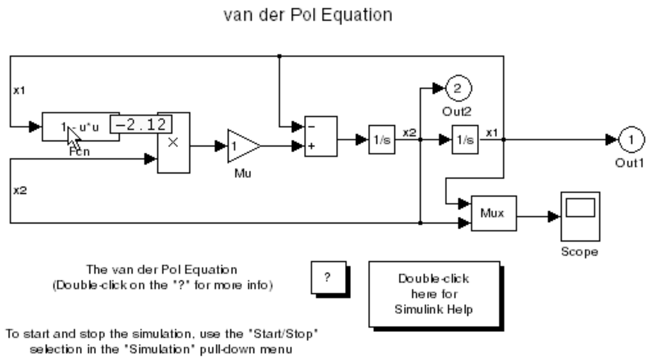
- The **Hide Name** menu item hides a visible block name. When you select **Hide Name**, it changes to **Show Name** when that block is selected.
- The **Show Name** menu item shows a hidden block name.

Specifying a Block's Color

See "Specifying Block Diagram Colors" on page 4-5 for information on how to set the color of a block.

Displaying Block Outputs

Simulink can display block outputs as data tips on the block diagram while a simulation is running.



van der Pol Equation

The van der Pol Equation
(Double-click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop"
selection in the "Simulation" pull-down menu

You can specify whether and when to display block outputs (see “Enabling Port Values Display” on page 5-18) and the size and format of the output displays and the rate at which Simulink updates them during a simulation (see “Port Values Display Options” on page 5-19).

Enabling Port Values Display

To turn display of port output values on or off, select **Port Values** from the model editor’s **View** menu. A menu of display options appears. Select one of the following display options from the menu:

- **Show none**

  Turns port value displaying off.

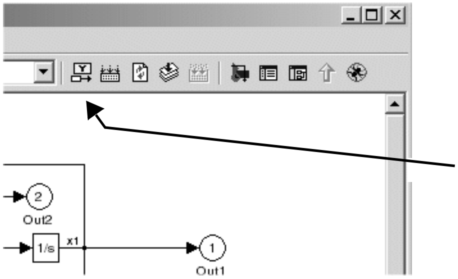
- **Show when hovering**

  Displays output port values for the block under the mouse cursor.

- **Toggle when selected**

  Selecting a block displays its outputs. Reselecting the block turns the display off.

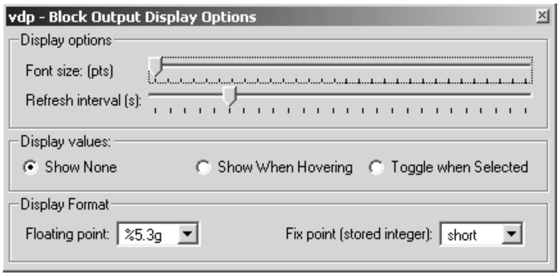
When using the Microsoft Windows version of Simulink, you can turn block output display when hovering on or off from the model editor's toolbar. To do this, select the block output display button on the toobar.



Click to show/hide block output when hovering

Port Values Display Options

To specify other display options, select **Port Values -> Options** from the model editor's **View** menu. The **Block Output Display Options** dialog box appears.



vdp - Block Output Display Options

Display options

Font size: (pts)

Refresh interval (s):

Display values:

● Show None ○ Show When Hovering ○ Toggle when Selected

Display Format

Floating point: %5.3g

Fix point (stored integer): short

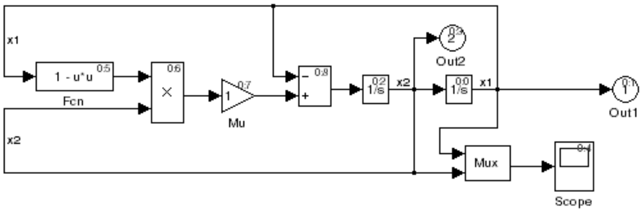
To increase the size of the output display text, move the **Font size** slider to the right. To increase the rate at which Simulink updates the displays, move the **Refresh interval slider** to the left.

Controlling and Displaying the Sorted Order

The sorted order is an ordering of the blocks in the model that Simulink uses as a starting point for determining the order in which to invoke the blocks' methods during simulation. Simulink allows you to display the sorted order for a model and to assign priorities to blocks that can influence where they appear in the sorted order.

Displaying the Sorted Order

To display the sorted order, select **Sorted order** from the Simulink **Format** menu. Selecting this option causes Simulink to display a notation in the top right corner of each block in a block diagram.



The notation for most blocks has the format `s:b`, where `s` specifies the index of the subsystem to whos execution context (see "Conditional Execution Behavior" on page 4-38) the block belongs and `b` specifies the block's position in the sorted order for that execution context. The sorted order of a Function-Call Subsystem cannot be determined at compile time. Therefore, for these subsystems, Simulink uses either the notation `s:F`, if the system has one initiator, where s is the index of the subsystem that contains the initiator, or the notation M, if the subsystem has more than one initiator.

Assigning Block Priorities

You can assign priorities to nonvirtual blocks or virtual subsystem blocks in a model (see "Virtual Blocks" on page 5-2). Higher priority blocks appear before lower priority blocks in the sorted order, 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 5-9).

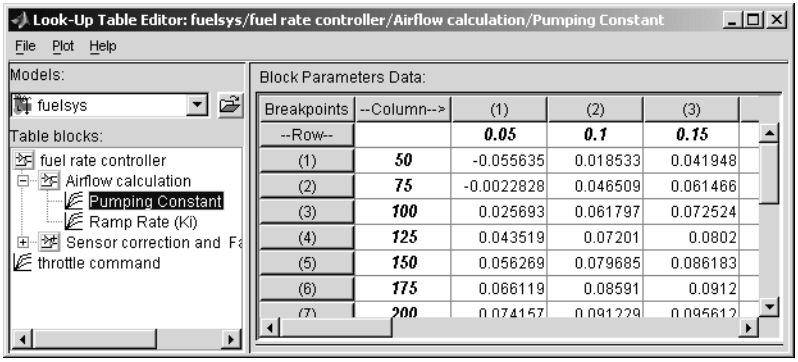
Simulink honors the block priorities that you specify only if they are consistent with the Simulink block sorting algorithm. If Simulink is unable to honor a block priority, it displays a `Block Priority Violation` diagnostic message (see "The Diagnostics Pane" on page 10-48).

Lookup Table Editor

The Lookup Table Editor allows you to inspect and change the table elements of any lookup table (LUT) block in a model (see “Lookup Tables” in the online Simulink documentation), including custom LUT blocks that you have created, using the Simulink Mask Editor (see “Editing Custom LUT Blocks” on page 5-27). You can also use a block’s parameter dialog to edit its table. However, that requires you to open the subsystem containing the block first and than its parameter dialog box first The LUT editor allows you to skip these steps. This section explains how to open and use the LUT editor to edit LUT blocks.

Note You cannot use the LUT Editor to change the dimensions of a lookup table. You must use the block’s parameter dialog box for this purpose.

To open the editor, select **Lookup table editor** from the Simulink **Tools** menu. The editor appears.

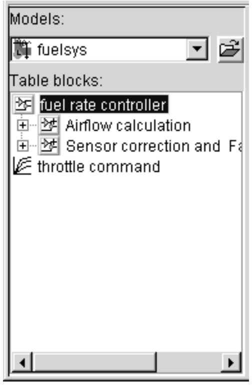


Breakpoints	--Column-->	(1)	(2)	(3)
--Row--		0.05	0.1	0.15
(1)	50	-0.055635	0.018533	0.041948
(2)	75	-0.0022828	0.046509	0.061466
(3)	100	0.025693	0.061797	0.072524
(4)	125	0.043519	0.07201	0.0802
(5)	150	0.056269	0.079685	0.086183
(6)	175	0.066119	0.08591	0.0912
(7)	200	0.074157	0.091229	0.095612

The editor contains two panes. The pane on the left is a LUT block browser. It allows you to browse and select LUT blocks in any open model (see “Browsing LUT Blocks” on page 5-23). The pane on the right allows you to edit the selected block’s lookup table (“Editing Table Values” on page 5-24).

Browsing LUT Blocks

The **Models** list in the upper left corner of the LUT Editor lists the names of all models open in the current MATLAB session.To browse any open model's LUT table blocks, select the model's name from the list. A tree-structured view of the selected model's LUT blocks appears in the **Table blocks** field beneath the **Models** list.



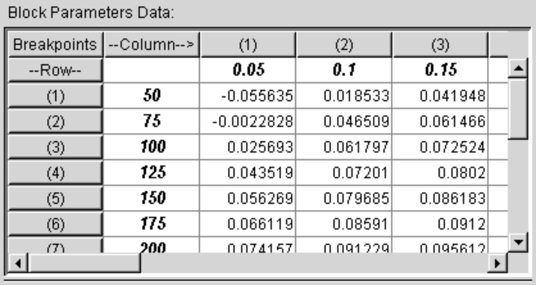
The tree view initially lists all the LUT blocks that reside at the model's root level. It also displays any subsystems that contain LUT blocks. Clicking the expand button (+) to the left of the subsystem's name expands the tree to show the LUT blocks in that subsystem. The expanded view also shows any subsystems in the expanded subsystem. You can continue expanding subsystem nodes in this manner to display LUT blocks at any level in the model hierarchy.

Clicking any LUT block in the LUT block tree view displays the block's lookup table in the right hand pane, allowing you to edit the table (see“Editing Table Values” on page 5-24).

Note If you want to browse the LUT blocks in a model that is not currently open, you can command the LUT Editor to open the model. To do this, select **Open** from the LUT Editor's **File** menu.

Editing Table Values

The **Block parameters data** table view of the LUT Editor allows you to edit the lookup table of the LUT block currently selected in the adjacent tree view.



Block Parameters Data:

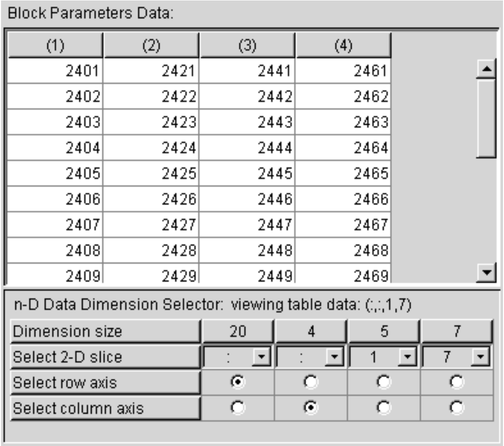
Breakpoints	--Column-->	(1)	(2)	(3)
--Row--		***0.05***	***0.1***	***0.15***
(1)	***50***	-0.055635	0.018533	0.041948
(2)	***75***	-0.0022828	0.046509	0.061466
(3)	***100***	0.025693	0.061797	0.072524
(4)	***125***	0.043519	0.07201	0.0802
(5)	***150***	0.056269	0.079685	0.086183
(6)	***175***	0.066119	0.08591	0.0912
(7)	***200***	0.074157	0.091229	0.095612

The table view displays the entire table if it is one- or two-dimensional or a two-dimensional slice of the table if the table has more than two dimensions (see “Displaying N-D Tables” on page 5-25). To change any of the displayed values, double-click the value. The LUT Editor replaces the value with an edit field containing the value. Edit the value, then press **Enter** or click outside the field to confirm the change.

The LUT Editor records your changes in a copy of the table that it maintains. To update the copy maintained by the LUT block itself, select **Update block data** from the LUT Editor’s **File** menu. To restore the LUT Editor ‘s copy to the values stored in the block, select **Reload block data** from the **File** menu.

Displaying N-D Tables

If the lookup table of the LUT block currently selected in the LUT Editor's tree view has more than two dimensions, the editor's table view displays a two-dimensional slice of the table.



Block Parameters Data:

(1)	(2)	(3)	(4)
2401	2421	2441	2461
2402	2422	2442	2462
2403	2423	2443	2463
2404	2424	2444	2464
2405	2425	2445	2465
2406	2426	2446	2466
2407	2427	2447	2467
2408	2428	2448	2468
2409	2429	2449	2469

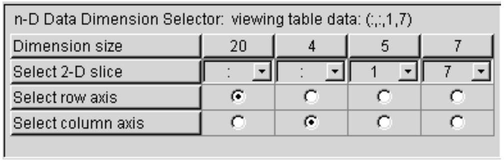
n-D Data Dimension Selector: viewing table data: (:,:,1,7)

Dimension size	20	4	5	7
Select 2-D slice	:	:	1	7
Select row axis	●	○	○	○
Select column axis	○	●	○	○

The **n-D Data Dimension Selector** beneath the table specifies which slice currently appears and allows you to select another slice. The selector consists of a 4-by-N array of controls where N is the number of dimensions in the lookup table. Each column corresponds to a dimension of the lookup table. The first column corresponds to the first dimension of the table, the second column to the second dimension of the table, and so on. The top row of the selector array displays the size of each dimension. The remaining rows specify which dimensions of the table correspond to the row and column axes of the slice and the indices that select the slice from the remaining dimensions.

To select another slice of the table, click the **Select row axis** and **Select column axis** radio buttons in the columns that correspond to the dimensions that you want to view. Then select the indexes of the slice from the pop-up index lists in the remaining columns.

For example, the following selector displays slice (:,: ,1,7) of a 4-D table.

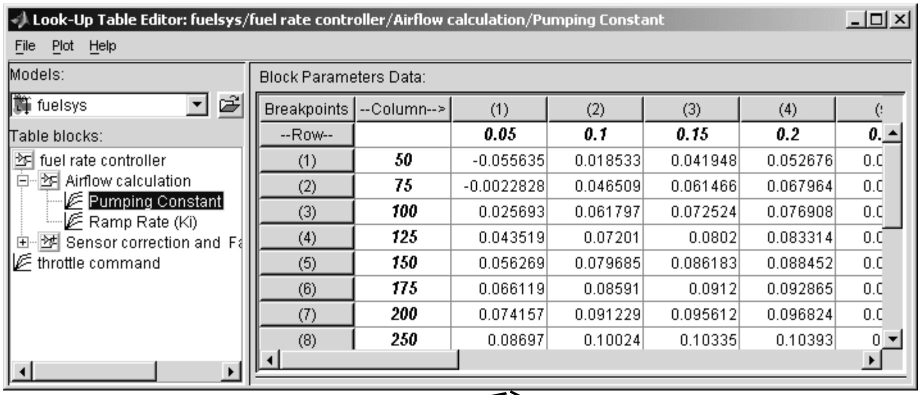


n-D Data Dimension Selector: viewing table data: (:,:,1,7)

Dimension size	20	4	5	7
Select 2-D slice	:	:	1	7
Select row axis	●	○	○	○
Select column axis	○	●	○	○

Plotting LUT Tables

Select **Linear** or **Mesh** from the **Plot** menu of the LUT Editor to display a linear or mesh plot of the table or table slice currently displayed in the editor's table view.



Look-Up Table Editor: fuelsys/fuel rate controller/Airflow calculation/Pumping Constant

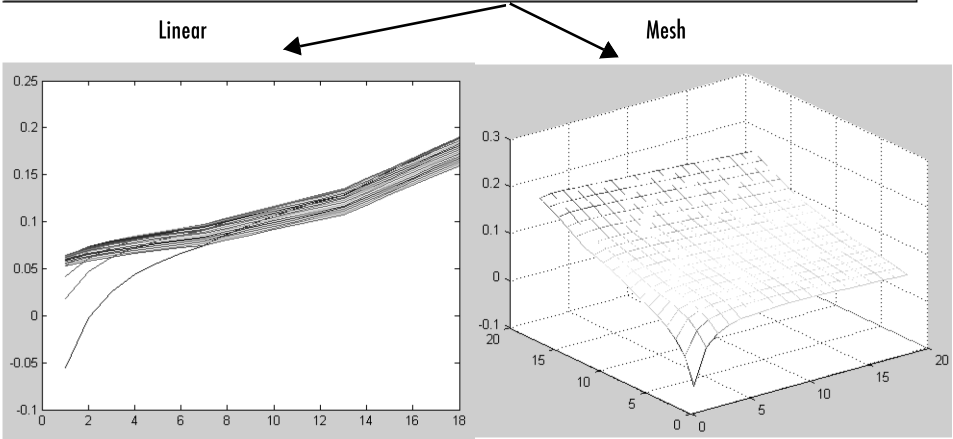
Models: fuelsys

Table blocks:

- fuel rate controller
  - Airflow calculation
    - Pumping Constant
    - Ramp Rate (Ki)
  - Sensor correction and Fa
- throttle command

Block Parameters Data:

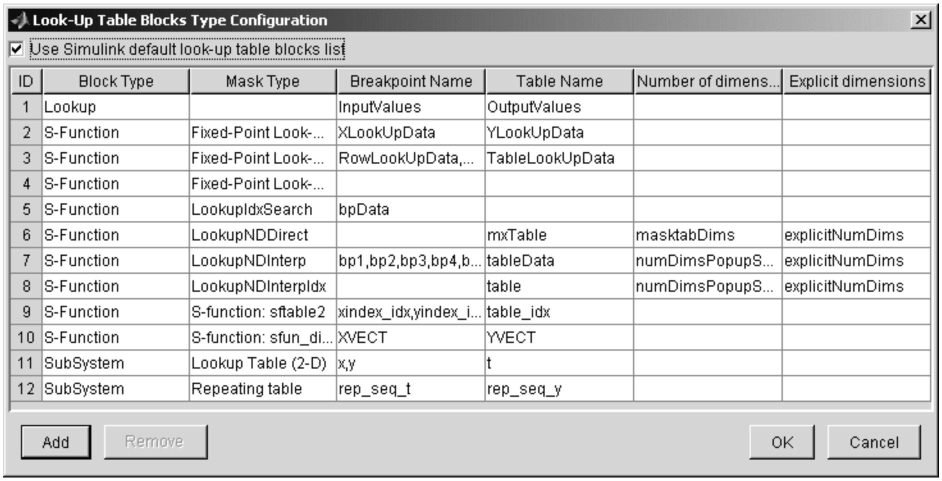
Breakpoints	--Column-->	(1)	(2)	(3)	(4)	(5
--Row--		***0.05***	***0.1***	***0.15***	***0.2***	***0.***
(1)	***50***	-0.055635	0.018533	0.041948	0.052676	0.0
(2)	***75***	-0.0022828	0.046509	0.061466	0.067964	0.0
(3)	***100***	0.025693	0.061797	0.072524	0.076908	0.0
(4)	***125***	0.043519	0.07201	0.0802	0.083314	0.0
(5)	***150***	0.056269	0.079685	0.086183	0.088452	0.0
(6)	***175***	0.066119	0.08591	0.0912	0.092865	0.0
(7)	***200***	0.074157	0.091229	0.095612	0.096824	0.0
(8)	***250***	0.08697	0.10024	0.10335	0.10393	0



Editing Custom LUT Blocks

You can use the LUT Editor to edit custom lookup table blocks that you or others have created. To do this, you must first configure the LUT Editor to recognize the custom LUT blocks in your model. Once you have configured the LUT Editor to recognize the custom blocks, you can edit them as if they were standard blocks.

To configure the LUT editor to recognize custom LUT blocks, select **Configure** from the editor's **File** menu. The **Look-Up Table Blocks Type Configuration** dialog box appears.



☑ Use Simulink default look-up table blocks list

ID	Block Type	Mask Type	Breakpoint Name	Table Name	Number of dimens...	Explicit dimensions
1	Lookup		InputValues	OutputValues		
2	S-Function	Fixed-Point Look-...	XLookUpData	YLookUpData		
3	S-Function	Fixed-Point Look-...	RowLookUpData,...	TableLookUpData		
4	S-Function	Fixed-Point Look-...				
5	S-Function	LookupIdxSearch	bpData			
6	S-Function	LookupNDDirect		mxTable	masktabDims	explicitNumDims
7	S-Function	LookupNDInterp	bp1,bp2,bp3,bp4,b...	tableData	numDimsPopupS...	explicitNumDims
8	S-Function	LookupNDInterpIdx		table	numDimsPopupS...	explicitNumDims
9	S-Function	S-function: sftable2	xindex_idx,yindex_i...	table_idx		
10	S-Function	S-function: sfun_di...	XVECT	YVECT		
11	SubSystem	Lookup Table (2-D)	x,y	t		
12	SubSystem	Repeating table	rep_seq_t	rep_seq_y		

By default the dialog box displays a table of the types of LUT blocks that the LUT Editor currently recognizes. By default these are the standard Simulink LUT blocks. Each row of the table displays key attributes of a LUT block type.

Adding a Custom LUT Type

To add a custom block to the list of recognized types,

1 Select the **Add** button on the dialog box.

A new row appears at the bottom of the block type table.

2 Enter information for the custom block in the new row under the following headings.

Field Name	Description
Block Type	Block type of the custom LUT block. The block type is the value of the block's BlockType parameter.
Mask Type	Mask type in this field. The mask type is the value of the block's MaskType parameter.
Breakpoint Name	Names of the custom LUT block's parameters that store its breakpoints.
Table Name	Name of the block parameter that stores the custom block's lookup table.
Number of dimensions	Leave empty.
Explicit Dimensions	Leave empty.

3 Select **OK**.

Removing Custom LUT Types

To remove a custom LUT type from the list of types recognized by the LUT Editor, select the custom type's entry in the table in the **Look-Up Table Blocks Type Configuration** dialog box. Then select **Remove**. To remove all custom LUT types, check the check box labeled **Use Simulink default look-up table blocks list** at the top of the dialog box.

Working with Block 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.

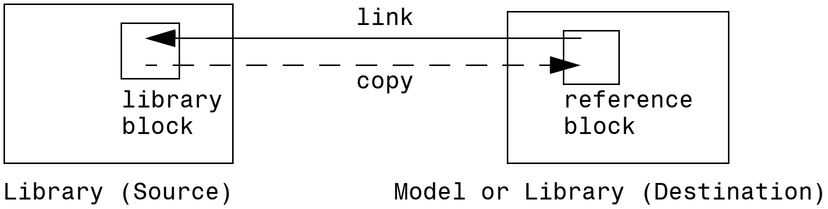
Library block – A block in a library.

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.



Simulink Block Library

Simulink comes with a library of standard blocks called the Simulink block library. See “Starting Simulink” on page 3-2 for information on displaying and using this library.

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 “Model Construction Commands” in the online Simulink reference.

The library must be named (saved) before you can copy blocks from it. See “Adding Libraries to the Library Browser” on page 5-38 for information on how to point the Library Browser to your new library.

Modifying a Library

When you open a library, it is automatically locked and you cannot modify its contents. To unlock the library, select **Unlock Library** from the **Edit** menu. Closing the library window locks the library.

Creating a Library Link

To create a link to a library block in a model, copy the block from the library to the model (see “Copying and Moving Blocks from One Window to Another” on page 5-4) or by dragging the block from the Library Browser (see “Browsing Block Libraries” on page 5-36) into the model window.

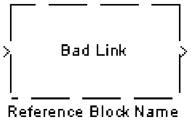
When you copy a library block into a model or another library, Simulink creates a link to the library block. The reference block is a copy of the library block. You can change the values of the reference block’s parameters but you cannot mask the block or, if it is masked, edit the mask. Also, you cannot 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 5-31).

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 do one of the following:

- Delete the unlinked reference block and copy the library block back into your model.
- Add the directory that contains the required library to the MATLAB path and select **Update Diagram** from the **Edit** menu.
- Double-click the reference block. On the dialog box that appears, correct the pathname and click **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 changing 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 propagate 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 5-34)
- When you use the `find_system` command

Updating Links to Reflect Block Path Changes

Library forwarding tables enable Simulink to update models to reflect changes in the names or locations of the library blocks that they reference. For example, suppose that you rename a block in a library. You can use a forwarding table for that library to enable Simulink to update models that reference the block under its old name to reference it under its new name.

Simulink allows you to associate a forwarding table with any library. The forwarding table for a library specifies the old locations and new locations of blocks that have moved within the library or to another library. You associate a forwarding table with a library by setting its ForwardingTable parameter to a cell array of two-element cell arrays, each of which specifies the old and new path of a block that has moved. For example, the following command creates a forwarding table and assigns it to a library named Lib1.

```
set_param('Lib1', 'ForwardingTable', {{'Lib1/A', 'Lib2/A'}
{'Lib1/B', 'Lib1/C'}});
```

The forwarding table specifies that block A has moved from Lib1 to Lib2. and that block B is now named C. Suppose that you opensa model that contains links to Lib1/A and Lib1/B. Simulink updates the link to Lib1/A to refer to Lib2/A and the link to Lib1/B to refer to Lib1/C. The changes become permanent when you subsequently save the model.

Breaking a Link to a Library Block

You can break the link between a reference block and its library block to cause the reference block to become a simple copy of the library block, unlinked to the library block. Changes to the library block no longer affect the block. Breaking links to library blocks may enable you to transport a 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')
```

Note Breaking library links in a model does not guarantee that you can run the model stand-alone, especially if the model includes blocks from third-party libraries or optional Simulink blocksets. It is possible that a library block invokes functions supplied with the library and hence can run only if the library is installed on the system running the model. Further, breaking a link can cause a model to fail when you install a new version of the library on a system. For example, suppose a block invokes a function that is supplied with the library. Now suppose that a new version of the library eliminates the function. Running a model with an unlinked copy of the block results in invocation of a now nonexistent function, causing the simulation to fail. To avoid such problems, you should generally avoid breaking links to third-party libraries and optional Simulink blocksets.

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 and highlights the library block and makes the source library the active window. If the library is not open, Simulink opens it and selects the library block.

Library Link Status

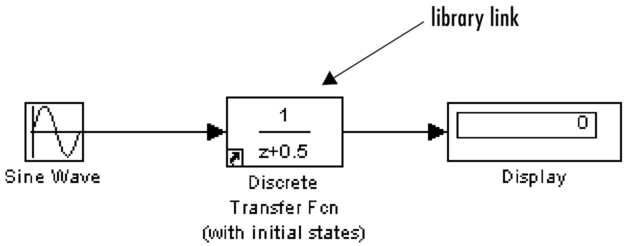
All blocks have a `LinkStatus` parameter that indicates whether the block is a reference block. The parameter can have these values.

Status	Description
`none`	Block is not a reference block.
`resolved`	Link is resolved.
`unresolved`	Link is unresolved.

Status	Description
`implicit`	Block resides in library block and is itself not a link to a library block. For example, suppose that A is a link to a subsystem in a library that contains a Gain block. Further, suppose that you open A and select the Gain block. Then, `get_param(gcb, 'LinkStatus')` returns `implicit`.
`inactive`	Link is disabled.
`restore`	Restores a broken link to a library block and discards any changes made to the local copy of the library block. For example, set_param(gcb, 'LinkStatus', 'restore') replaces the selected block with a link to a library block of the same type, discarding any changes in the local copy of the library block. Note that this parameter is a "write-only" parameter, i.e., it is usable only with `set_param`. You cannot use `get_param` to get it.
`propagate`	Restores a broken link to a library block and propagates any changes made to the local copy to the library.

Displaying Library Links

Simulink optionally displays an arrow in the bottom left corner of each block that represents a library link in a model.



This arrow allows you to tell at a glance whether a block 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

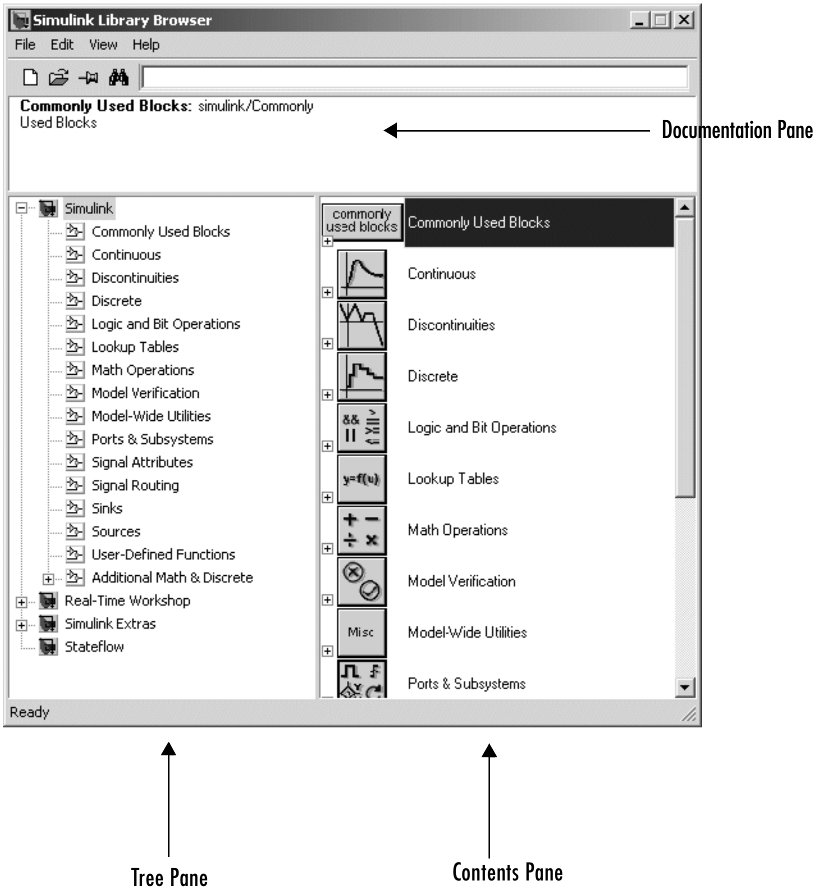
Use the `libinfo` command to get information about reference blocks in a system

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 enter `simulink` at the MATLAB command line.

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

The Library Browser contains three panes.



The tree pane displays all the block libraries installed on your system. The contents pane displays the blocks that reside in the library currently selected in the tree pane. The documentation pane displays documentation for the block selected in the contents 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, 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.

Accessing Block Data During Simulation

Simulink provides an application programming interface (API) that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to develop MATLAB programs capable of accessing block data while a simulation is running or to access the data from the MATLAB command line.

Note You can use this interface even when the model is paused or is running or stopped in the debugger.

To obtain the interface for a particular block, select the block and execute

```
br = get_param(gcb,'RuntimeObject');
```

while the model containing the block is running. The syntax for accessing the value and attributes of the run-time data is:

```
get(br.InputPort(portIdx))
get(br.OutputPort(portIdx))
get(br.Dwork(dworkIdx))
get(br.ContStates(stateIdx))
get(br.RuntimePrm(prmIdx))
```

For example, to display the signal values at the first input port of a block when stopped in the debugger, you could type:

```
br = get_param(gcb,'RuntimeObject');
br.InputPort(1).Data
```

The execution interface also supports an event-listener mechanism that allows you to listen for specific events during model simulation. In particular, you can request that Simulink throw events whenever a specific method (e.g., Outputs, Derivatives, or Update) of a given block is executed during a simulation. For instance, to determine when block a in model b executes its Outputs method, execute the following commands after you have started simulating the model:

```
br = get_param('b/a','RuntimeObject');
add_exec_event_listener('b/a','PostOutputs','disp(''here'')')
br.ExecutionEvents = 1;
```

Simulink now displays `here` after executing the output method of block `b/a`.

Note The final release of R14 will provide mechanisms for adding events before a simulation starts.

The `add_exec_event_listener` command supports the following events:

- `PreOutputs`
- `PostOutputs`
- `PreUpdate`
- `PostUpdate`
- `PreDerivs`
- `PostDerivs`

The listener can be any M callback you choose to register. The callback can also be a function handle for a function that accepts two input arguments:

- the block run-time object

  i.e., the object returned by `get_param(gcb,'RuntimeObject')`
- an event data object

Note You must install the events at the start of each simulation of a model, even in the same Simulink session.

Working with Signals

This section describes how to create and use Simulink signals.

Signal Basics

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

About Signals

Simulink defines signals as the outputs of dynamic systems represented by blocks in a Simulink diagram and by the diagram itself. The lines in a block diagram represent mathematical relationships among the signals defined by the block diagram. For example, a line connecting the output of block A to the input of block B indicates that the signal output by B depends on the signal output by A.

Note It is tempting but misleading to think of Simulink signals as traveling along the lines that connect blocks the way electrical signals travel along a telephone wire. This analogy is misleading because it suggests that a block diagram represents physical connections between blocks, which is not the case. Simulink signals are mathematical, not physical, entities and the lines in a block diagram represent mathematical, not physical, relationships among signals.

Creating Signals

You can create signals by creating source blocks in your model. For example, you can create a signal that varies sinusoidally with time by dragging an instance of the Sine block from the Simulink Sources library into the model. See “Sources” in the online “Block Libraries” reference for information on blocks that you can use to create signals in a model. You can also use the Signal & Scope Manager to create signals in your model without using blocks. See “The Signal & Scope Manager” on page 6-16 for more information.

Signal Labels

A signal label is text that appears next to the line that represents a signal that has a name. The signal label displays the signal’s name. In addition, if the signal is a virtual signal (see “Virtual Signals” on page 6-4) and its **Show propagated signals** property is on (see “Show propagated signals” on page 6-31), the label displays the names of the signals that make up the virtual signal.

Simulink creates a label for a signal when you assign it a name in the **Signal Properties** dialog box (see “Signal Properties Dialog Box” on page 6-30). You can change the signal’s name by editing its label on the block diagram. To edit the label, left-click the label. Simulink replaces the label with an edit field. Edit the name in the edit field, the press **Enter** or click outside the label to confirm the change.

Displaying Signal Values

As with creating signals, you can use either blocks or the Signal & Scope Manager to display the values of signals during a simulation. For example, you can use either the Scope block or the Signal & Scope Manager to graph time-varying signals on an oscilloscope-like display during simulation. See “Sinks” in the online “Block Libraries” reference for information on blocks that you can use to display signals in a model.

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 7-2 for more information.

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 refer 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 "Simulink Blocks" in the online Simulink Help. See "Determining Output Signal Dimensions" on page 6-12 for information on what determines the dimensions of output signals for blocks that can output nonscalar signals.

Complex Signals

The values of Simulink signals can be complex numbers. A signal whose values are complex numbers is called a complex signal. You can introduce a complex-valued signal into a model in the following ways:

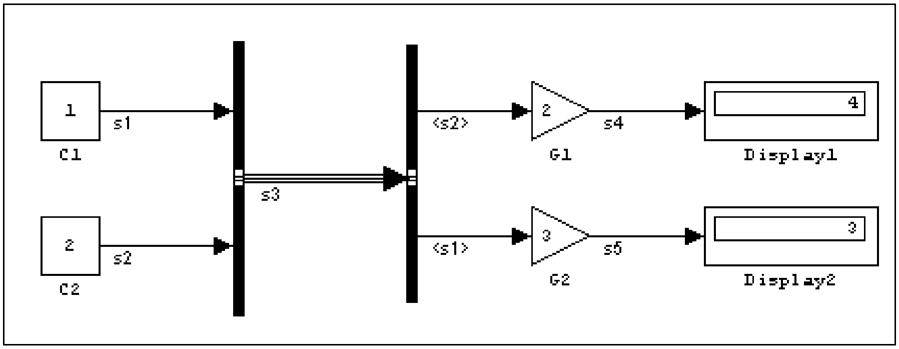
- Load complex-valued signal data from the MATLAB workspace into the model via a root-level inport.
- Create a Constant block in your model and set its value to a complex number.
- Create real signals corresponding to the real and imaginary parts of a complex signal, then combine the parts into a complex signal, using the Real-Imag to Complex conversion block.

You can manipulate complex signals via blocks that accept them. If you are not sure whether a block accepts complex signals, see the documentation for the block in the "Simulink Blocks" section of the Simulink online documentation.

Virtual Signals

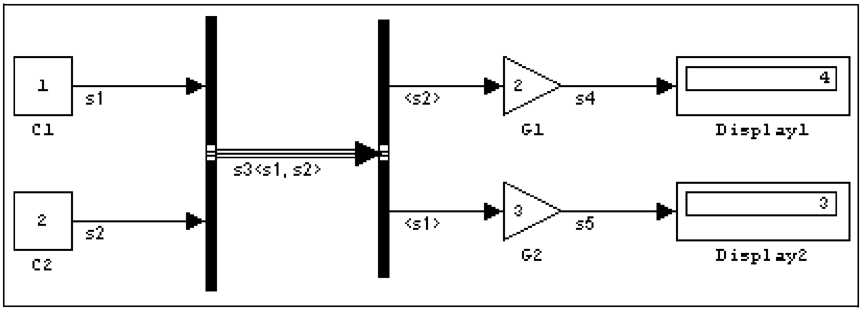
A *virtual signal* is a signal that represents another signal graphically. Some blocks, such as Bus Creator, Inport, and Outport blocks (see "Virtual Blocks" on page 5-2), generate virtual signals either exclusively or optionally (see "Virtual Versus Nonvirtual Buses" on page 6-7). Virtual signals are 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. Consider, for example, the following model.



The signals driving Gain blocks G1 and G2 are virtual signals corresponding to signals s2 and s1, respectively. Simulink determines this automatically whenever you update or simulate the model.

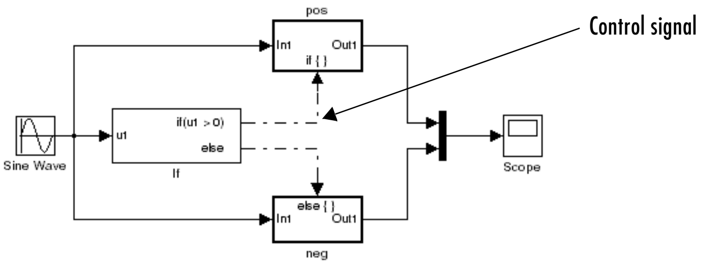
The **Show propagated signals** option (see “Signal Properties Dialog Box” on page 6-30) displays the nonvirtual signals represented by virtual signals in the labels of the virtual signals.



Note Virtual signals can represent virtual as well as nonvirtual signals. For example, you can use a Bus Creator block to combine multiple virtual and nonvirtual signals into a single virtual signal. If during signal propagation Simulink determines that a component of a virtual signal is itself virtual, Simulink determines its nonvirtual components using signal propagation. This process continues until Simulink has determined all nonvirtual components of a virtual signal.

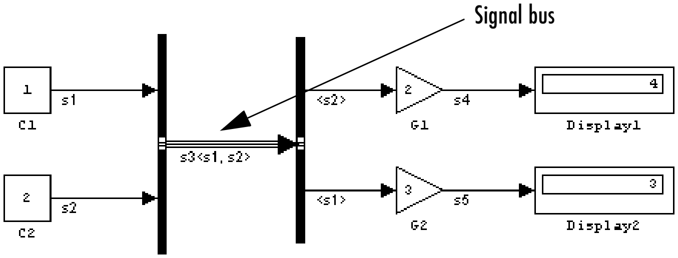
Control Signals

A *control signal* is a signal used by one block to initiate execution of another block, e.g., a function-call or action subsystem. When you update or start simulation of a block diagram, Simulink uses a dash-dot pattern to redraw lines representing the diagram's control signals as illustrated in the following example.



Signal Buses

A bus is a composite signal comprising a set of signals represented graphically by a bundle of lines. It is analogous to a bundle of wires held together by tie wraps. The components of a bus can have different data types and can themselves be composite signals (i.e., buses or muxed signals). You can use Bus Creator and Inport blocks to create signal buses and Bus Selector blocks to access a bus's components.



Selecting a bus and then **Signal Dimensions** from the model editor's **Format** menu displays the number of signal components carried by the bus.

Virtual Versus Nonvirtual Buses

Buses may be either virtual or nonvirtual. During simulation, blocks connected to a virtual bus read their inputs from memory allocated to the component signals, which may reside in noncontiguous areas of memory. By contrast, blocks connected to a nonvirtual bus read their inputs from a copy of the component signals maintained by Simulink in a contiguous area of memory allocated to the bus.

Some Simulink features, such as model referencing (see “Referencing Models” on page 4-53), require use of nonvirtual signals. Others require virtual buses. Nonvirtual buses also facilitate code generation by enabling buses to be represented by data structures. On the other hand, nonvirtual buses can save memory where nonvirtual buses are not required.

The Bus Creator and Inport blocks output virtual buses by default. To cause them to output a nonvirtual bus, select the **Output as structure** option on their parameter dialog boxes. You can also use the Signal Conversion block to convert nonvirtual to virtual buses, and vice versa.

Bus-Capable Blocks

A *bus-capable block* is a block through which both virtual and nonvirtual buses can pass. All virtual blocks are bus capable. Further, the following nonvirtual blocks are also bus-capable:

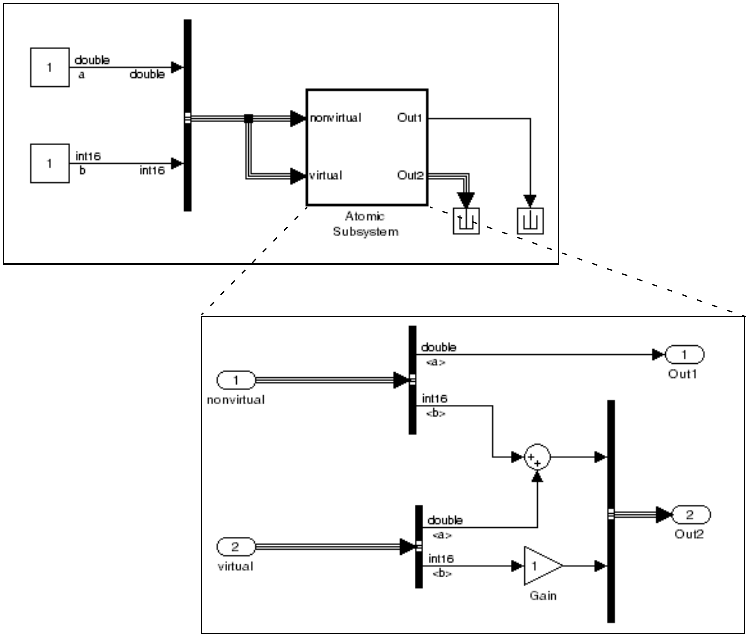
- Memory
- Merge
- Switch
- Multiport Switch
- Rate Transition
- Unit Delay
- Zero-Order Hold

Some bus-capable blocks impose constraints on bus propagation through them. See the documentation for the individual blocks for more information.

Connecting Buses to Subsystem Inports

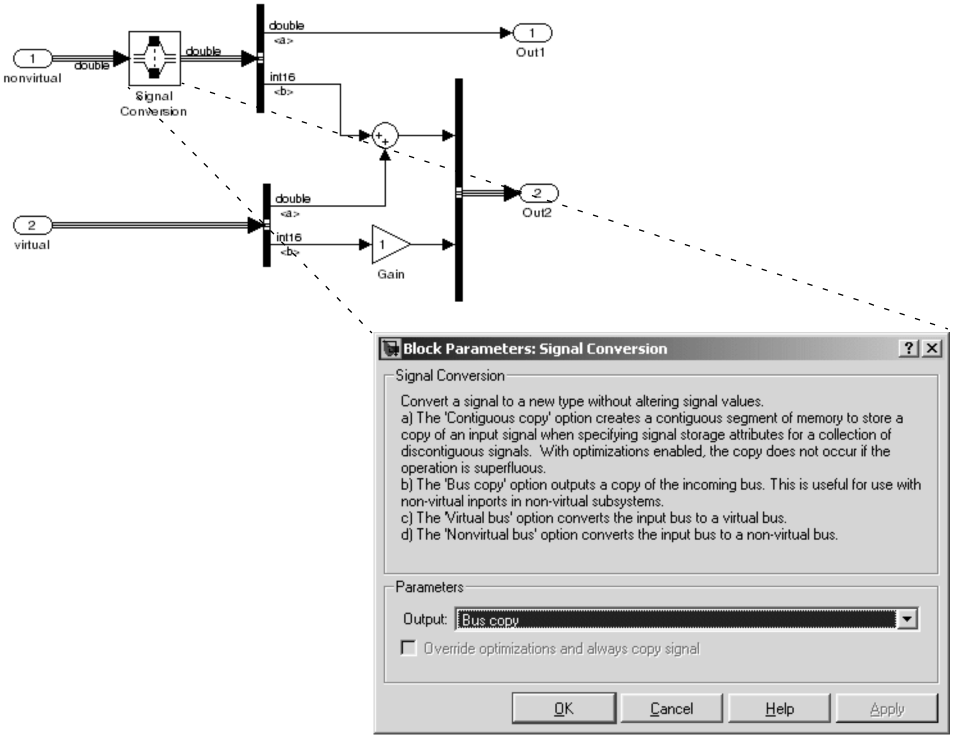
Generally, an Inport block is a virtual block and hence accepts a bus as input. However, an Inport block is nonvirtual if it resides in a conditionally executed or atomic subsystem and it or any of its components is directly connected to an output of the subsystem. In such a case, the Inport block can accept a bus only if its components have the same data type. If the components are of differing data types, attempting to simulate the model causes Simulink to halt the simulation and display an error message. You can avoid this problem, without changing the semantics of your model, by inserting a Signal Conversion block between the Inport block and the Outport block to which it was originally connected.

Consider, for example, the following model.



In this model, the Inport labeled `nonvirtual` is nonvirtual because it resides in an atomic subsystem and one of its components (labeled `a`) is directly connected to one of the subsystem's outputs. Further, the bus connected to the subsystem's inputs has components of differing data types. As a result, Simulink cannot simulate this model.

Inserting a Signal Conversion block with the `bus copy` option selected breaks the direct connection to the subsystem's output and hence enables Simulink to simulate the model.



Block Parameters: Signal Conversion

Signal Conversion

Convert a signal to a new type without altering signal values.
a) The 'Contiguous copy' option creates a contiguous segment of memory to store a copy of an input signal when specifying signal storage attributes for a collection of discontiguous signals. With optimizations enabled, the copy does not occur if the operation is superfluous.
b) The 'Bus copy' option outputs a copy of the incoming bus. This is useful for use with non-virtual inports in non-virtual subsystems.
c) The 'Virtual bus' option converts the input bus to a virtual bus.
d) The 'Nonvirtual bus' option converts the input bus to a non-virtual bus.

Parameters

Output: Bus copy

☐ Override optimizations and always copy signal

Connecting Buses to Model Inports

If you want a root level Inport of a model to be able to accept a bus signal, you must set the Inport's bus object parameter to the name of a bus object that defines the type of bus that the Inport accepts. See "Working with Data Objects" on page 7-10 and Simulink.Bus class for more information.

Checking Signal Connections

Many Simulink blocks have limitations on the types of signals they can accept. Before simulating a model, Simulink checks all blocks to ensure that they can accommodate the types of signals output by the ports to which they are connected. If any incompatibilities exist, Simulink reports an error and terminates the simulation. To detect 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.

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 7-2 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	A composite signal made up of other signals, including other buses. You can use Bus Creator, Mux, and Inport blocks to create signal buses.
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.

Term	Meaning
Test point	A signal that must be accessible during simulation (see “Signal Properties Dialog Box” on page 6-30).
Vector	One-dimensional signal array.
Virtual signal	Signal that represents another signal or set of signals.
Width	Size of a vector signal.

Determining Output Signal Dimensions

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

Determining the Output Dimensions of Source Blocks

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

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

Constant Value	Interpret Vector Parameters as 1-D	Output
2-D scalar	off	2-D scalar
2-D scalar	on	1-D scalar
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

Constant Value	Interpret Vector Parameters as 1-D	Output
M-by-N matrix	off	M-by-N matrix
M-by-N matrix	on	M-by-N matrix

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

Determining the Output Dimensions of Nonsource Blocks

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

All nonscalar inputs to a block must have the same dimensions.

A block can have a mix of scalar and nonscalar inputs as long as all the nonscalar inputs have the same dimensions. Simulink expands the scalar inputs to have the same dimensions as the nonscalar inputs (see "Scalar Expansion of Inputs" on page 6-14), thus preserving the general rule.

Block Parameter Dimension Rule

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

Two seeming exceptions exist to this general rule:

- A block can have scalar parameters corresponding to nonscalar inputs. In this case, Simulink expands a scalar parameter to have the same dimensions as the corresponding input (see "Scalar Expansion of Parameters" on page 6-15), thus preserving the general rule.

- If an input is a vector, the corresponding parameter can be either an N-by-1 or a 1-by-N matrix. In this case, Simulink applies the N matrix elements to the corresponding elements of the input vector. This exception allows use of MATLAB row or column vectors, which are actually 1-by-N or N-by-1 matrices, respectively, to specify parameters that apply to vector inputs.

Vector or Matrix Input Conversion Rules

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

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

Note You can configure Simulink to display a warning or error message if a vector or matrix conversion occurs during a simulation. See “Vector/matrix block input conversion” on page 10-54 for more information.

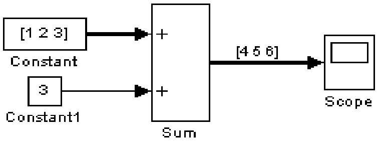
Scalar Expansion of Inputs and Parameters

Scalar expansion is the conversion of a scalar value into a nonscalar array of the same dimensions. Many Simulink blocks support scalar expansion of inputs and parameters. Block descriptions in the “Simulink Blocks” section in the online Simulink Help 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]`.

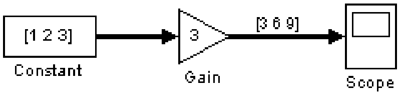


When a block's output is a function of a parameter and the parameter is nonscalar, Simulink expands a scalar input to match the dimensions of the parameter. For example, Simulink expands a scalar input to a Gain block to match the dimensions of a nonscalar gain parameter.

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.

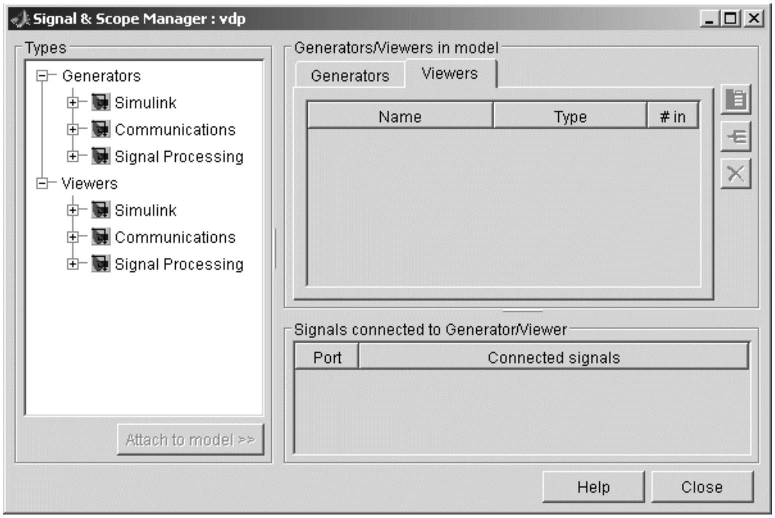


The Signal & Scope Manager

The Signal & Scope Manager lets you globally manage signal generators and viewers.

Note The Signal & Scope Manager requires that you start MATLAB with Java enabled (the default).

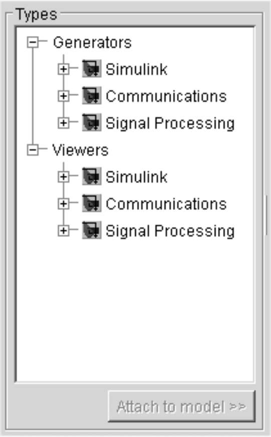
To display the Signal & Scope Manager, select **Signal & Scope Manager** from the model editor's **Tools** or context menu. The Signal & Scope Manager appears.



The Signal & Scope Manager contains the following groups of controls.

Generator and Viewer Types

This group of controls lets you create signal generators and viewers of various types and associate them with your model.



The tree control displays a list of the types of generators and viewers installed on your system. The tree's second-level nodes group the generators and viewers by the products that provides them (i.e., Simulink and any MathWorks blocksets installed on your system). Expand a product's nodes to see the generators and viewers that it provides.

For information on the attributes and usage of the generators and viewers, see the documentation for the identically named source (i.e., generator) and sink (i.e., viewer) blocks in the product's documentation. For example, for information on the generators and viewers provided with Simulink, see the documentation for the corresponding blocks in the Simulink Sources and Sinks libraries.

Note The Scope viewer, unlike the Scope block, cannot log signals to the MATLAB workspace. To log a signal displayed on the Scope viewer, select the signal logging option on the signal's **Signal Properties** dialog box (see "Logging Signals" on page 6-27).

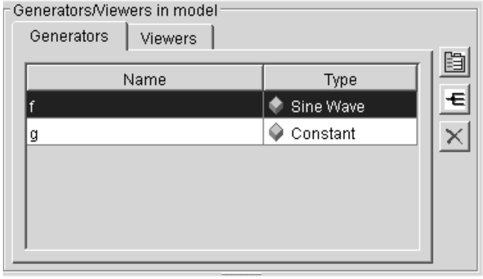
To create an instance of a generator or viewer and associate with the currently selected model, select its type in the type list and then click the **Attach to model** button beneath the list.

Generator and Viewer Objects

This group of controls lets you edit the sources and viewers already associated with your model. It contains the following controls.

Generators

The **Generators** pane displays a table listing the generators associated with your model.

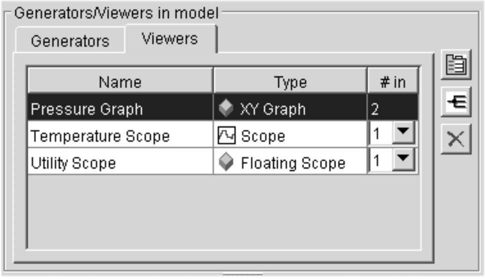


Name	Type
f	Sine Wave
g	Constant

Each row corresponds to a generator. The columns specify each generator's name and type.

Viewers

The **Viewers** pane displays a table listing the viewers associated with your model.



Name	Type	# in
Pressure Graph	XY Graph	2
Temperature Scope	Scope	1
Utility Scope	Floating Scope	1

Each row corresponds to a viewer. The columns specify each viewer's name, type, and number of inputs. If a viewer accepts a variable number of inputs, the **#in** entry for the viewer contains a pull-down list that displays the range of inputs that the viewer can accept. To change the number of inputs accepted by the viewer, pull down the list and select the desired value.

Edit Buttons

Selecting the table entry for a generator or viewer enables the following buttons.

Button	Description
▤	Opens the parameter dialog box for the selected generator or viewer. The parameter dialog box enables you to view and change the current settings of the selected object's parameters. See the documentation for the corresponding source or sink block for more information.
∈	Opens the Signal Selector for the selected generator or viewer. The Signal Selector lets you connect signal generators to your model's inputs and your model's signals to its signal viewers. **Note** You can also use port or signal context menus to connect signals to input ports and output ports to viewers. For example, to connect a signal to a new viewer, select **Create Viewer** from the signal or output port's context menu, then the type of viewer. To connect a signal to an existing viewer, select **Connect to Viewer**, then the axis to display the signal. Similarly, to connect a new signal generator to a block input port, select **Create Generator** from the input port's context menu, then the type of generator.
☒	Deletes the selected generator or viewer.

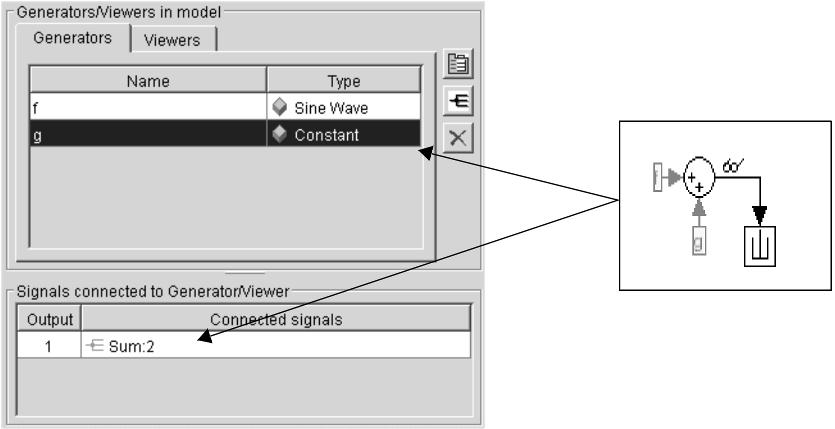
Edit Menu

Selecting a row in the generator or viewer table and pressing the right button on your mouse displays an edit menu containing entries corresponding to the edit buttons described in the preceding section. It also displays a **Rename** command for renaming the selected object (e.g., a viewer). Selecting this command causes Simulink to replace the selected object's name with an edit control. Use the edit control to rename the object.

Note You can also rename a signal generator on a model's block diagram. To do this, select **Edit Source Name** from the context menu of an input port to which the signal generator is connected. Simulink replaces the source's name with an edit field containing the source's name. Edit the name and then click outside the field or press **Enter** to confirm your changes.

Signals connected to Generator/Viewer

This table lists the signals connected to the generator or viewer selected in the Generator/Viewers control panel of the Signal & Scope Manager.



Generators/Viewers in model

Name	Type
f	Sine Wave
g	Constant

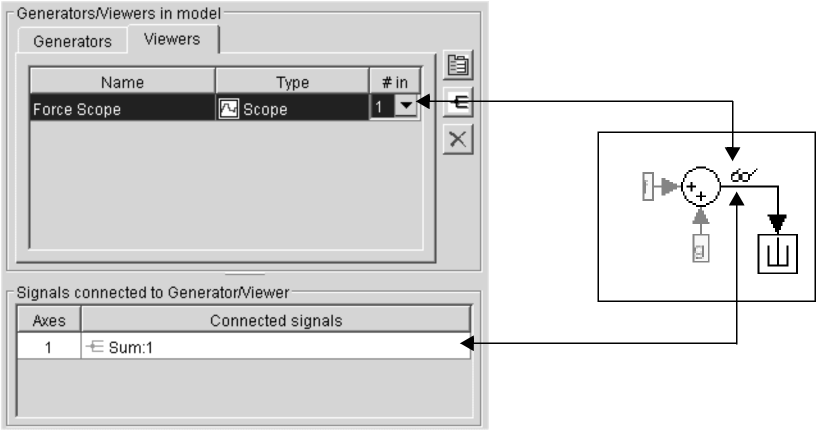
Signals connected to Generator/Viewer

Output	Connected signals
1	Sum:2

If the selected object is a signal generator, the table lists the block input ports to which each of the generator's outputs is connected. For each connection, the first column of the table specifies the number of the corresponding generator output. The second column specifies the number of the corresponding input port and the name of the block that owns the input port. For example, in the preceding figure, the **Signals connected to Generator/Viewer** table shows that the first (and only output) of the selected Constant generator is connected to the second input port of the block named Sum.

If the selected object is a signal viewer, the **Signals connected to Generator/Viewer** table lists the signals connected to the selected viewer. For each connection, the first column of the table specifies the number of the corresponding viewer axis. The second column specifies the number of the corresponding output port and the name of the block that owns the output port.

For example, in the next figure, the **Signals connected to Generator/Viewer** table shows that the first axis of the selected signal viewer is connected to the first output port of the block named Sum.


Generators/Viewers in model
Generators
Viewers
Signals connected to Generator/Viewer

Name	Type	# in
Force Scope	Scope	1

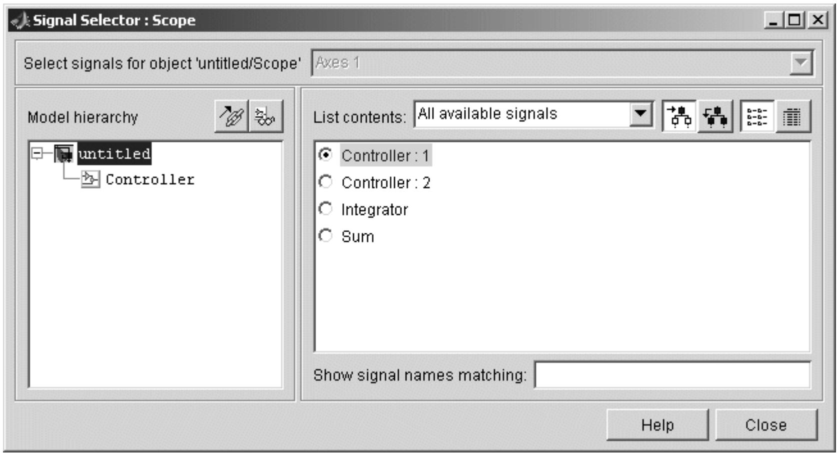
Axes	Connected signals
1	Sum:1

Connection Menu

Selecting a connection in the **Signals connected to Generator/Viewer** table and pressing the right button on your mouse displays a context menu. To highlight the block to which the object is connected, select **Hilight signal in model** from the menu. To open the Signal Selector, select **Edit Signal Connections** from the model.

The Signal Selector

The Signal Selector allows you to connect a signal or viewer object (see "The Signal & Scope Manager" on page 6-16) or the Floating Scope to block inputs and outputs. It appears when you click the signal selection button for a signal or viewer object in the Signal & Scope Manager or on the toolbar of the Floating Scope's window.



Signal Selector : Scope

Select signals for object 'untitled/Scope' Axes 1

Model hierarchy

- untitled
  - Controller

List contents: All available signals

- ● Controller : 1
- ○ Controller : 2
- ○ Integrator
- ○ Sum

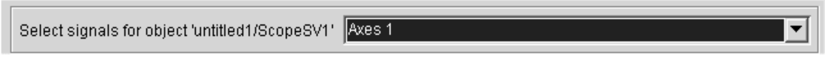
Show signal names matching:

The Signal Selector that appears when you click the signal selection button applies only to the currently selected signal or viewer object (or the Floating Scope). If you want to connect blocks to another signal or viewer object, you must select the object in the Signal & Scope Manager and launch another instance of the Signal Selector. The object used to launch a particular instance of the Signal Selector is called that instance's owner.

The Signal Selector includes the following control panels.

Port/Axis Selector

This list box allows you to select the owner output port (in the case of signal generators) or display axis (in the case of signal viewers) to which you want to connect blocks in your model.

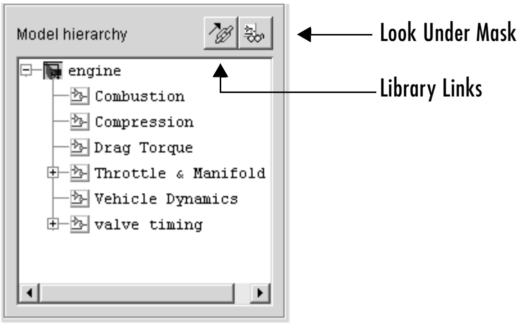


Select signals for object 'untitled1/ScopeSV1' Axes 1

The list box is enabled only if the signal generator has multiple outputs or the signal viewer has multiple axes.

Model Hierarchy

This tree-structured list lets you select any subsystem in your model.

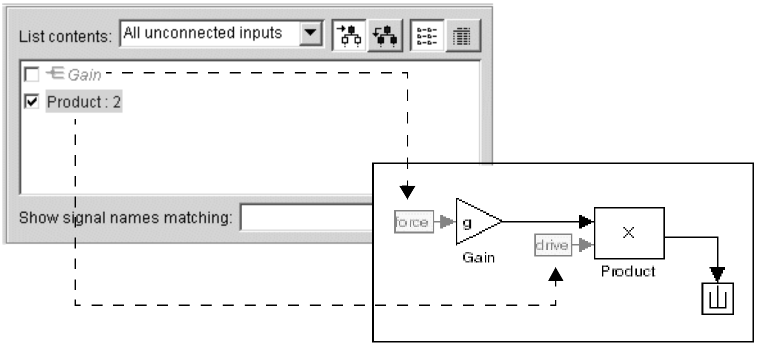


Selecting a subsystem causes the adjacent port list panel to display the ports available for connection in the selected subsystem. To display subsystems included as library links in your model, click the **Library Links** button at the top of the **Model hierarchy** panel. To display subsystems contained by masked subsystems, click the **Look Under Masks** button at the top of the panel.

Inputs/Signals List

The contents of this panel displays input ports available for connection to the Signal Selector's owner if the owner is a signal generator or signals available for connection to the owner if the owner is a signal viewer.

If the Signal Selector's owner is a signal generator, the inputs/signals list by default lists each input port in the system selected in the model hierarchy tree that is either unconnected or connected to a signal generator.



List contents: All unconnected inputs

□ Gain
☑ Product : 2

Show signal names matching:

The label for each entry indicates the name of the block of which the port is an input. If the block has more than one input, the label indicates the number of the displayed port. A greyed label indicates that the port is connected to a signal generator other than the Signal Selectors' owner. Checking the checkbox next to a port's entry in the list connects the Signal Selector's owner to the port, replacing, if necessary, the signal generator previously connected to the port.

To display more information on each signal, click the **Detailed view** button at the top of the pane. The detailed view shows the path and data type of each signal and whether the signal is a test point. The controls at the top and bottom of the panel let you restrict the amount of information shown in the ports list.

- To show named signals only, select `Named signals only` from the **List contents** control at the top of the pane.
- To show 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 the **Show signals matching** control at the bottom of the **Signals** pane and press the **Enter** key.
- To show the selected types of signals for all subsystems below the currently selected subsystem in the model hierarchy, click the **Current and Below** button at the top of the **Signals** pane.

To select or deselect a signal in the **Signals** pane, click its entry or use the arrow keys to move the selection highlight to the signal entry and press the **Enter** key. You can also move the selection highlight to a signal entry by typing the first few characters of its name (enough to uniquely identify it).

Note You can continue to select and deselect signals on the block diagram with the Signal Selector open. For example, shift-clicking a line in the block diagram adds the corresponding signal to the set of signals that you previously selected with the Signal Selector. Simulink updates the Signal Selector to reflect signal selection changes you have made on the block diagram. However, the changes do not appear until you select the Signal Selector window itself.

Logging Signals

Logging signals refers to the process of saving signal values to the MATLAB workspace during simulation for later retrieval and postprocessing. Simulink allows you to log a signal either by connecting the signal to a To Workspace or root-level Inport block or by setting the signal's signal logging properties. The first method allows you to document in the diagram itself the workspace variables used to store signal data. The second method reduces diagram clutter by eliminating the need to use blocks to log signals. Either method allows you to specify the names of the workspace variables used to save signal data and to limit the amount of data logged for a particular signal.

See the documentation for the To Workspace and Inport blocks for information on using these blocks to log and retrieve signal data. The remainder of this section explains how to use signal properties to log and access signal data.

Enabling Signal Logging

To enable signal logging for a signal, check the **Log signal data** option on the signal's **Signal Properties** dialog box (see "Signal Properties Dialog Box" on page 6-30).

Note Simulink does not support signal logging for nonvirtual buses. If you enable signal logging for a nonvirtual bus, Simulink displays an error message when you simulate the model.

Specifying a Logging Name

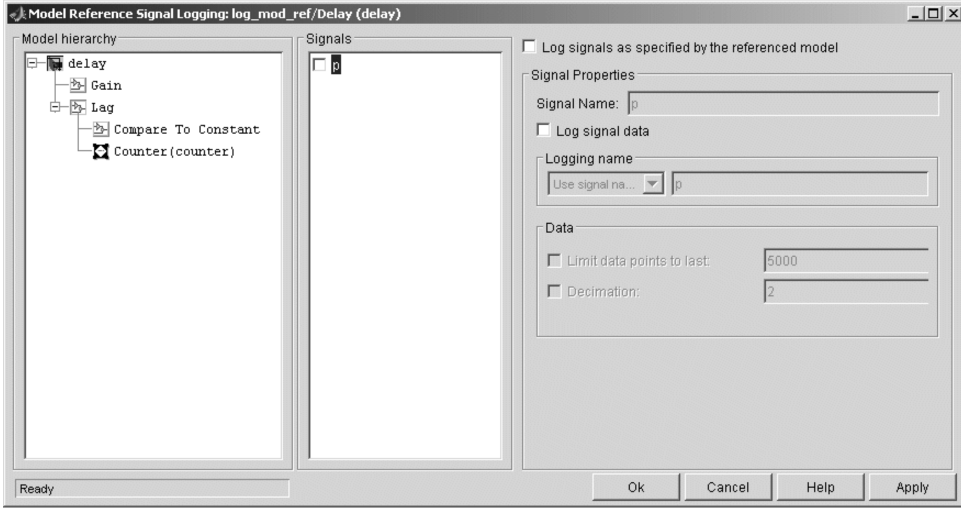
You can assign a name, called the logging name, to the object used to log data for a signal during simulation. If the signal to be logged does not have a name or is an element of a composite signal that has another element of the same name, you must specify a unique log name for the signal. To specify a log name for a signal, select `Custom` from the **Logging name** list on the signal's **Signal Properties** dialog box and enter the custom name in the adjacent text field. If a signal has a name you do not need to specify a logging name for the signal, Simulink uses the signal's name as its logging name.

Limiting the Data Logged for a Signal

The **Data** panel of the **Signal Properties** dialog box lets you limit the amount of data logged for a signal. For example, you can specify the maximum amount of data to be logged for a signal or a decimation factor that causes Simulink to skip a specified number of time steps before logging a signal value. See “Data” on page 6-33 for more information.

Logging Referenced Model Signals

To log signals in a model referenced by a Model block, select the Model block and then select **Log referenced signals** from the model editor’s **Edit** menu or from the block’s context menu. The **Model Reference Signal Logging** dialog box appears.



□ p

□ Log signals as specified by the referenced model

Signal Properties

Signal Name: p

□ Log signal data

Logging name: Use signal na... p

Data

□ Limit data points to last: 5000

□ Decimation: 2

The dialog box contains the following panes and controls.

Model Hierarchy

This pane displays the contents of the referenced model as a tree control with expandable nodes. The top-level node represents the referenced model. Expanding this node displays the subsystems that the referenced model contains and any models that it itself references. Expanding a subsystem node displays the subsystems that it contains and the models that it references.

Signals

This pane displays the test points of the model or subsystem selected in the **Model Hierarchy** pane (see “Working with Test Points” on page 6-35). Check the checkbox next to a test point’s name to specify that it should be logged.

Note Simulink does not support logging of virtual buses in referenced models. If you enable logging for such a bus, Simulink displays a warning message when you simulate the model.

Log signals as specified by the referenced model

Checking this checkbox causes Simulink to log the signals that the referenced model specifies should be logged.

Signal Properties

This pane is enabled if **Log signals as specified by the referenced model** is not selected. In this case, the controls on this pane allow you to specify the signal logging properties of the signal selected in the **Signals** pane. The values that you specify override for this instance of the referenced model those specified by the model itself. The controls correspond to the controls of the same name on the **Signal Properties dialog box**. See “Signal Properties Dialog Box” on page 6-30 for information on how to use them.

Accessing Logged Signal Data

Simulink saves signal data that it logs during simulation in a Simulink data object of type `Simulink.ModelDataLogs` that resides in the MATLAB workspace. The name of the object’s handle is `logsout` by default. The **Data Import/Export** configuration pane (see “Data Import/Export Pane” on page 10-39) allows you to specify another name for this object. See `Simulink.ModelDataLogs` in the “Data Object Classes” chapter of the *Simulink Reference* for information on extracting signal data from this object.

Note Logging data is available only at the end of a simulation. It is not available when the simulation is paused.

Signal Properties Dialog Box

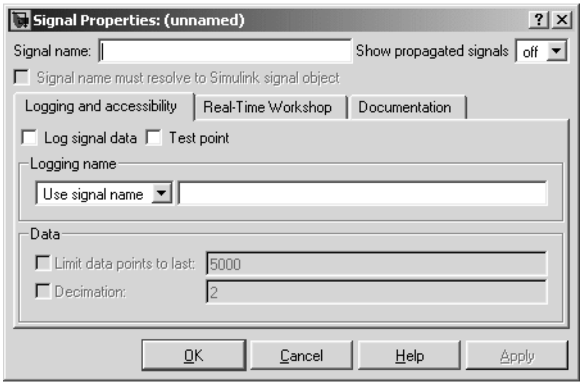
The **Signal Properties** dialog box lets you display and edit signal properties. To display the dialog box, either

- Select the line that represents the signal whose properties you want to set and then choose **Signal Properties** from the signal's context menu or from the Simulink **Edit** menu

  or

- Select the block that outputs the signal and select **Output Port Signal Properties** from the block's context menu

The dialog box appears.



Signal Properties: (unnamed)

Signal name:

Show propagated signals: off

□ Signal name must resolve to Simulink signal object

□ Log signal data □ Test point

Logging name: Use signal name

Data

□ Limit data points to last: 5000

□ Decimation: 2

The dialog box includes the follow controls.

Signal name

Name of signal.

Signal name must resolve to a Simulink signal object.

Specifies that the MATLAB workspace must contain a `Simulink.Signal` object with the same name as this signal. Simulink displays an error message if it cannot find such an object when you update or simulate the model containing this signal.

Show propagated signals

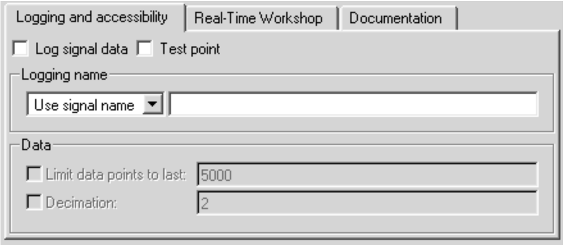
Note This option appears only for signals that originate from a virtual block other than a Bus Selector block.

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, the label for `s1` 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, the label for `s1` is `s1<s2,s4,s5>`.

Logging and Accessibility Options

Select the **Logging and accessibility** tab on the **Signal Properties** dialog box to display controls that enable you to specify signal logging and accessibility options for this signal.



□ Log signal data □ Test point

Logging name

Use signal name ▾

Data

□ Limit data points to last: 5000

□ Decimation: 2

Log signal data

Select this option to cause Simulink to save this signal's values to the MATLAB workspace during simulation (see "Logging Signals" on page 6-27).

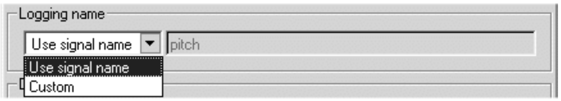
Test point

Select this option to designate this signal as a test point (see "Signal Properties Dialog Box" on page 6-30).

Note If you select the **Log signal data** option for this signal, Simulink selects and disables the **Test point** option so that you cannot deselect it. This is because a signal must be a test point to be logged.

Logging name

This pair of controls, consisting of a list box and an edit field, specifies the signal's logging name, i.e., the name under which to be used to retrieve the data that Simulink logs for this signal during simulation.



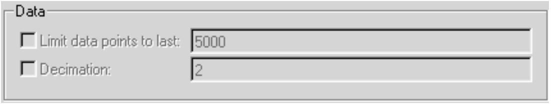
Logging name

Use signal name ▾ pitch

Simulink uses the signal's signal name as its logging name by default. To specify a custom logging name, select `Custom` from the list box and enter the custom name in the adjacent edit field.

Data

This group of controls enables you to limit the amount of data that Simulink logs for this signal.



Data

□ Limit data points to last: 5000

□ Decimation: 2

The options are

Limit data points to last. Discard all but the last `N` data points where `N` is the number entered in the adjacent edit field.

Decimation. Log every `N`th data point where `N` is the number entered in the adjacent edit field. For example, suppose that your model uses a fixed-step solver with a step size of `0.1 s`. if you select this option and accept the default decimation value (`2`), Simulink records data points for this signal at times `0.0`, `0.2`, `0.4`, etc.

Real-Time Workshop Options

The following controls set properties used by 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.

Documentation Options

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.

Working with Test Points

A *test point* is a signal that Simulink guarantees to be observable, for example, on a Floating Scope, during a simulation. Simulink allows you to designate any signal in a model as a test point. Designating a signal as a test point exempts the signal from model optimizations, such as signal storage reuse (see "Signal storage reuse" on page 10-46) and block reduction (see "Implement logic signals as boolean data (vs. double)" on page 10-45), that can render signals inaccessible and hence unobservable during simulation.

Designating a Signal as a Test Point

To designate a signal as a test point, check the **Test point** option on the signal's **Signal Properties** dialog box (see "Signal Properties Dialog Box" on page 6-30).

Note If you enable signal logging for a signal, Simulink designates the signal as a test point automatically. This is because a signal must be accessible to be logged (see "Enabling Signal Logging" on page 6-27 for more information).

Note If you set the test point property of a signal in a library that is referenced by a model that is itself referenced by another model, you must update the referenced model by opening and saving it. Otherwise, Simulink cannot log or display the referenced signal.

Using Signal Objects to Designate Test Points

You can use `Simulink.Signal` objects to designate test points from the MATLAB workspace.This allows you to designate test points in a model without having to modify the model itself. To use a `Simulink.Signal` object to control a signal's visibility, the following conditions must be true:

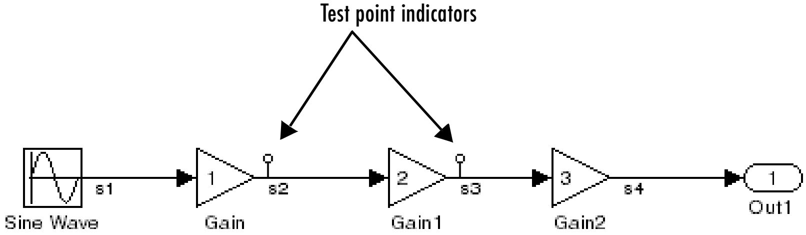
- The model does not specify the signal as a test point, i.e., the **Test point** option is unchecked in the **Signal Properties** dialog box.

- The model specifies the signal's storage class as `auto` (the default), i.e., the **Storage class** option in the signal's **Signal Properties** dialog box is set to `auto`.
- A `Simulink.Signal` object is associated with the signal, i.e., the MATLAB workspace contains a signal object having the same name as the signal.

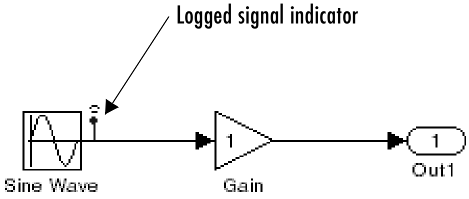
If all these conditions are true, you can designate the signal as a test point by setting the associated object's storage class property to any value but `auto`.

Displaying Test Point Indicators

By default, Simulink displays an indicator next to each signal that serves as a test point. These test point indicators enable you to find the test points in a model at a glance.



The appearance of the indicator changes slightly to indicate test points for which signal logging is enabled.



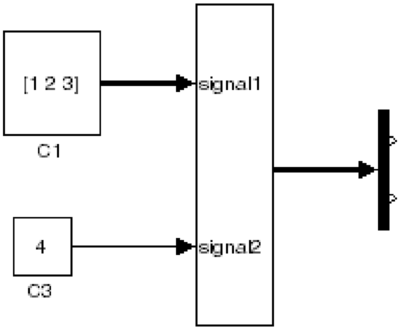
To turn display of test point indicators on or off, select **Port/Signal Displays -> Test Point Indicators** from the Simulink **Format** menu.

Displaying Signal Properties

A model window's **Format** menu and its model context (right-click) menu offer the following options for displaying signal properties on the block diagram.

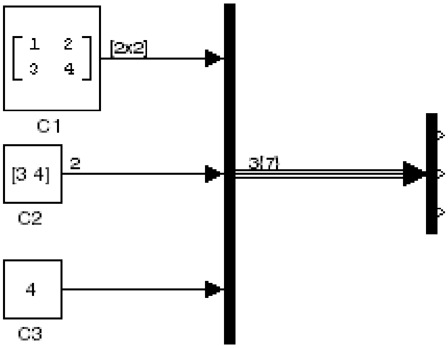
Wide nonscalar lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.



Signal dimensions

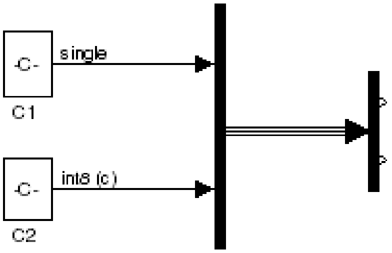
Display the dimensions of nonscalar signals next to the line that carries the signal.



The format of the display depends on whether the line represents a single signal or a bus. If the line represents a single vector signal, Simulink displays the width of the signal. If the line represents a single matrix signal, Simulink displays its dimensions as $[N_1xN_2]$ where N_i is the size of the ith dimension of the signal. If the line represents a bus carrying signals of the same data type, Simulink displays N{M} where N is the number of signals carried by the bus and M is the total number of signal elements carried by the bus. If the bus carries signals of different data types, Simulink displays only the total number of signal elements {M}.

Port data types

Displays the data type of a signal next to the output port that emits the signal.



The notation (c) following the data type of a signal indicates that the signal is complex.

Signal 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 6-30)
- 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 the line that represents the signal. The text cursor appears. Enter the name and click anywhere outside the label to exit label editing mode.

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

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

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

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

To edit an existing signal label, select it:

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

To delete all occurrences of a signal label, delete all the characters in the label. When you click outside the label, the labels are deleted. To delete a single occurrence of the label, hold down the **Shift** key while you select the label, then press the **Delete** or **Backspace** key.

To change the font of a signal label, select the signal, choose **Font** from the **Format** menu, then select a font from the **Set Font** dialog box.

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 6-30).

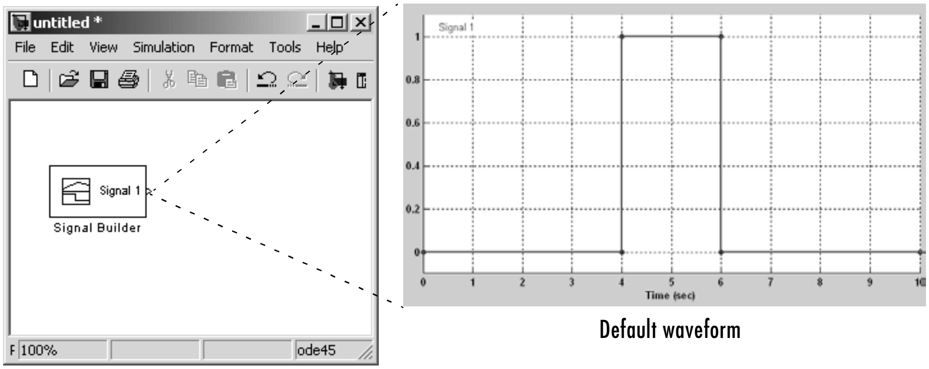
Working with Signal Groups

The Signal Builder block allows you to create interchangeable groups of signal sources and quickly switch the groups into and out of a model. Signal groups can greatly facilitate testing a model, especially when used in conjunction with Simulink assertion blocks and the optional Model Coverage Tool.

Creating a Signal Group Set

To create an interchangeable set of signal groups:

1 Drag an instance of the Signal Builder block from the Simulink Sources library and drop it into your model.



Default waveform

By default, the block represents a single signal group containing a single signal source that outputs a square wave pulse.

2 Use the block's signal editor (see "The Signal Builder Dialog Box" on page 6-42) to create additional signal groups, add signals to the signal groups, modify existing signals and signal groups, and select the signal group that the block outputs.

3 Connect the output of the block to your diagram.

The block displays an output port for each signal that the block can output.

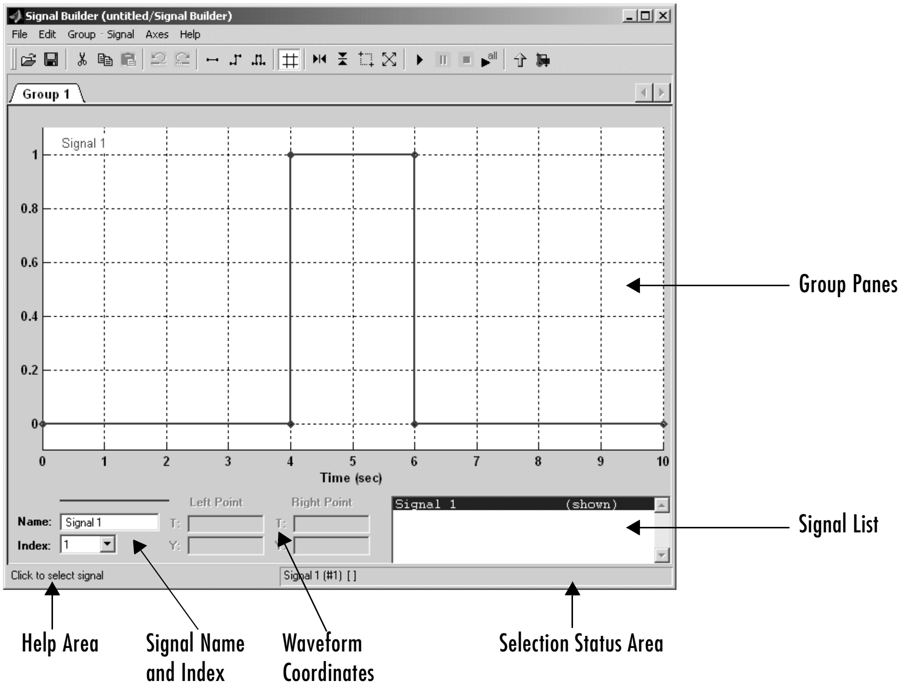
You can create as many Signal Builder blocks as you like in a model, each representing a distinct set of interchangeable groups of signal sources. See

"Simulating with Signal Groups" on page 6-51 for information on using signal groups in a model.

The Signal Builder Dialog Box

The Signal Builder block's dialog box allows you to define the waveforms of the signals output by the block. You can specify any waveform that is piecewise linear.

To open the dialog box, double-click the block. The **Signal Builder** dialog box appears.



Name: Signal 1

Index: 1

Signal 1 (shown)

Click to select signal

Signal 1 (#1) [ ]

The **Signal Builder** dialog box allows you to create and modify signal groups represented by a Signal Builder block. The **Signal Builder** dialog box includes the following controls.

Group Panes

Displays the set of interchangeable signal source groups represented by the block. The pane for each group displays an editable representation of the waveform of each signal that the group contains. The name of the group appears on the pane's tab. Only one pane is visible at a time. To display a group that is invisible, select the tab that contains its name. The block outputs the group of signals whose pane is currently visible.

Signal Axes

The signals appear on separate axes that share a common time range (see "Signal Builder Time Range" on page 6-50). This allows you to easily compare the relative timing of changes in each signal. The Signal Builder automatically scales the range of each axis to accommodate the signal that it displays. Use the Signal Builder's **Axes** menu to change the time (T) and amplitude (Y) ranges of the selected axis.

Signal List

Displays the names and visibility (see "Editing Signals" on page 6-44) of the signals that belong to the currently selected signal group. Clicking an entry in the list selects the signal. Double-clicking a signal's entry in the list hides or displays the signal's waveform on the group pane.

Selection Status Area

Displays the name of the currently selected signal and the index of the currently selected waveform segment or point.

Waveform Coordinates

Displays the coordinates of the currently selected waveform segment or point. You can change the coordinates by editing the displayed values (see "Editing Waveforms" on page 6-46).

Name

Name of the currently selected signal. You can change the name of a signal by editing this field (see "Renaming a Signal" on page 6-45).

Index

Index of the currently selected signal. The index indicates the output port at which the signal appears. An index of 1 indicates the topmost output port, 2 indicates the second port from the top, and so on. You can change the index of a signal by editing this field (see “Changing a Signal’s Index” on page 6-46).

Help Area

Displays context-sensitive tips on using **Signal Builder** dialog box features.

Editing Signal Groups

The Signal Builder dialog box allows you to create, rename, move, and delete signal groups from the set of groups represented by a Signal Builder block.

Creating and Deleting Signal Groups

To create a signal group, you must copy an existing signal group and then modify it to suit your needs. To copy an existing signal group, select its tab and then select **Copy** from the Signal Builder’s **Group** menu. To delete a group, select its tab and then select **Delete** from the **Group** menu.

Renaming Signal Groups

To rename a signal group, select the group’s tab and then select **Rename** from the Signal Builder’s **Group** menu. A dialog box appears. Edit the existing name in the dialog box or enter a new name. Click **OK**.

Moving Signal Groups

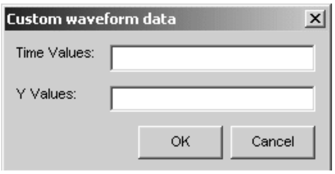
To reposition a group in the stack of group panes, select the pane and then select **Move right** from the Signal Builder’s **Group** menu to move the group lower in the stack or **Move left** to move the pane higher in the stack.

Editing Signals

The **Signal Builder** dialog box allows you to create, cut and paste, hide, and delete signals from signal groups.

Creating Signals

To create a signal in the currently selected signal group, select **New** from the Signal Builder’s **Signal** menu. A menu of waveforms appears. The menu includes a set of standard waveforms (**Constant**, **Step**, etc.) and a **Custom** waveform option. Select one of the waveforms. If you select a standard waveform, the Signal Builder adds a signal having that waveform to the currently selected group. If you select **Custom**, a custom waveform dialog box appears.



The dialog box allows you to specify a custom piecewise linear waveform to be added to the groups defined by the Signal Builder block. Enter the custom waveform's time coordinates in the **T Values** field and the corresponding signal amplitudes in the **Y Values** field. The entries in either field can be any MATLAB expression that evaluates to a vector. The resulting vectors must be of equal length. Select **OK**. The Signal Builder adds a signal having the specified waveform to the currently selected group.

Cutting and Pasting Signals

To cut or copy a signal from one group and paste it into another group:

1. Select the signal you want to cut or copy.
2. Select **Cut** or **Copy** from the Signal Builder's **Edit** menu or click the corresponding button from the toolbar.
3. Select the group into which you want to paste the signal.
4. Select **Paste** from the Signal Builder's **Edit** menu or click the corresponding button on the toolbar.

Renaming a Signal

To rename a signal, select the signal and choose **Rename** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field that displays the signal's current name. Edit or replace the current name with a new name. Click **OK**. Or edit the signal's name in the **Name** field in the lower left corner of the **Signal Builder** dialog box.

Changing a Signal's Index

To change a signal's index, select the signal and choose **Change Index** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field containing the signal's existing index. Edit the field and select **OK**. Or select an index from the **Index** list in the lower left corner of the Signal Builder window.

Hiding Signals

By default, the **Signal Builder** dialog box displays the waveforms of a group's signals in the group's tabbed pane. To hide a waveform, select the waveform and then select **Hide** from the Signal Builder's **Signal** menu. To redisplay a hidden waveform, select the signal's **Group** pane, then select **Show** from the Signal Builder's **Signal** menu to display a menu of hidden signals. Select the signal from the menu. Alternatively, you can hide and redisplay a hidden waveform by double-clicking its name in the Signal Builder's signal list (see "Signal List" on page 6-43).

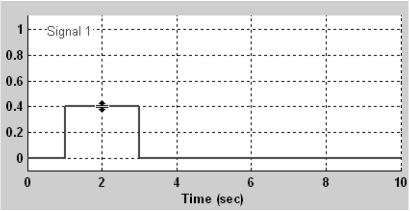
Editing Waveforms

The **Signal Builder** dialog box allows you to change the shape, color, and line style and thickness of the signal waveforms output by a signal group.

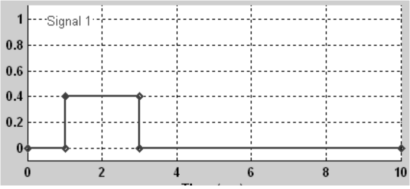
Reshaping a Waveform

The **Signal Builder** dialog box allows you to change the shape of a waveform by selecting and dragging its line segments and points with the mouse or arrow keys or by editing the coordinates of segments or points.

Selecting a Waveform. To select a waveform, left-click the mouse on any point on the waveform.

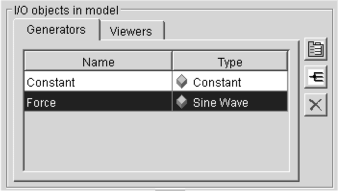


The Signal Builder displays the waveform's points to indicate that the waveform is selected.



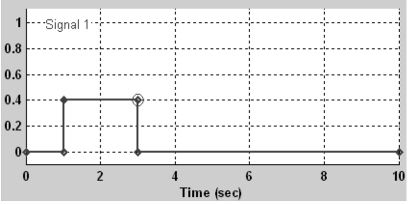
To deselect a waveform, left-click any point on the waveform graph that is not on the waveform itself or press the **Esc** key.

Selecting points. To select a point of a waveform, first select the waveform. Then position the mouse cursor over the point. The cursor changes shape to indicate that it is over a point.



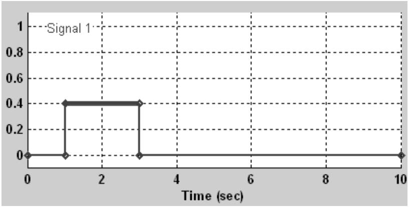
Name	Type
Constant	Constant
Force	Sine Wave

Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate that it is selected.



To deselect the point, press the **Esc** key.

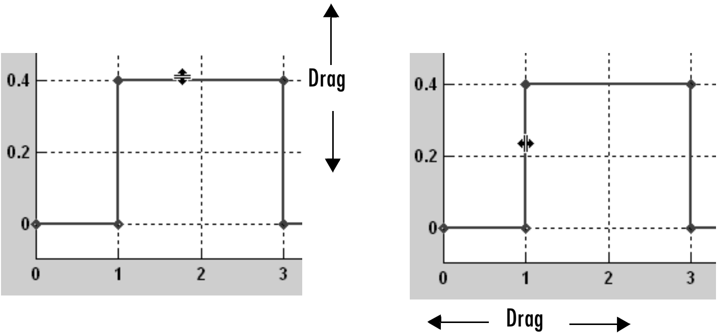
Selecting Segments. To select a line segment, first select the waveform that contains it. Then left-click the segment. The Signal Builder thickens the segment to indicate that it is selected.



To deselect the segment, press the **Esc** key.

Moving Waveforms. To move a waveform, select it and use the arrow keys on your keyboard to move the waveform in the desired direction. Each key stroke moves the waveform to the next location on the snap grid (see "Snap Grid" on page 6-49) or by `0.1` inches if the snap grid is not enabled.

Dragging Segments. To drag a line segment to a new location, position the mouse cursor over the line segment. The mouse cursor changes shape to show the direction in which you can drag the segment.



Press the left mouse button and drag the segment in the direction indicated to the desired location. You can also use the arrow keys on your keyboard to move the selected line segment.

Dragging points. To drag a point along the signal amplitude (vertical) axis, move the mouse cursor over the point. The cursor changes shape to a circle to indicate that you can drag the point. Drag the point parallel to the x-axis to the desired location. To drag the point along the time (horizontal) axis, press the **Shift** key while dragging the point. You can also use the arrow keys on your keyboard to move the selected point.

Snap Grid. Each waveform axis contains an invisible snap grid that facilitates precise positioning of waveform points. The origin of the snap grid coincides with the origin of the waveform axis. When you drop a point or segment that you have been dragging, the Signal Builder moves the point or the segment's points to the nearest point or points on the grid, respectively. The Signal Builder's **Axes** menu allows you to specify the grid's horizontal (time) axis and vertical (amplitude) axis spacing independently. The finer the spacing, the more freedom you have in placing points but the harder it is to position points precisely. By default, the grid spacing is 0, which means that you can place points anywhere on the grid; i.e., the grid is effectively off. Use the **Axes** menu to select the spacing that you prefer.

Inserting and Deleting points. To insert a point, first select the waveform. Then hold down the **Shift** key and left-click the waveform at the point where you want to insert the point. To delete a point, select the point and press the **Del** key.

Editing Point Coordinates. To change the coordinates of a point, first select the point. The Signal Builder displays the current coordinates of the point in the **Left Point** edit fields at the bottom of the **Signal Builder** dialog box. To change the amplitude of the selected point, edit or replace the value in the **y** field with the new value and press **Enter**. The Signal Builder moves the point to its new location. Similarly edit the value in the **t** field to change the time of the selected point.

Editing Segment Coordinates. To change the coordinates of a segment, first select the segment. The Signal Builder displays the current coordinates of the endpoints of the segment in the **Left Point** and **Right Point** edit fields at the bottom of the **Signal Builder** dialog box. To change a coordinate, edit the value in its corresponding edit field and press **Enter**.

Changing the Color of a Waveform

To change the color of a signal waveform, select the waveform and then select **Color** from the Signal Builder's **Signal** menu. The Signal Builder displays the MATLAB color chooser. Choose a new color for the waveform. Click **OK**.

Changing a Waveform's Line Style and Thickness

The Signal Builder can display a waveform as a solid, dashed, or dotted line. It uses a solid line by default. To change the line style of a waveform, select the waveform, then select **Line style** from the Signal Builder's **Signal** menu. A menu of line styles pops up. Select a line style from the menu.

To change the line thickness of a waveform, select the waveform, then select **Line width** from the **Signal** menu. A dialog box appears with the line's current thickness. Edit the thickness value and click **OK**.

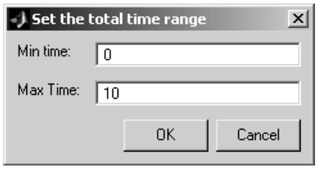
Signal Builder Time Range

The Signal Builder's time range determines the span of time over which its output is explicitly defined. By default, the time range runs from 0 to 10 seconds. You can change both the beginning and ending times of a block's time range (see "Changing a Signal Builder's Time Range" on page 6-50).

If the simulation starts before the start time of a block's time range, the block extrapolates its initial output from its first two defined outputs. If the simulation runs beyond the block's time range, the block by default outputs its final defined values for the remainder of the simulation. The Signal Builder's **Simulation Options** dialog box allows you to specify other final output options (see "Signal values after final time" on page 6-52 for more information).

Changing a Signal Builder's Time Range

To change the time range, select **Change time range** from the Signal Builder's **Axes** menu. A dialog box appears.



Set the total time range

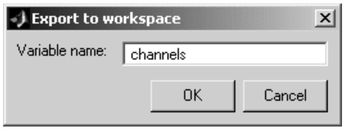
Min time: 0

Max Time: 10

Edit the **Min. time** and **Max. time** fields as necessary to reflect the beginning and ending times of the new time range, respectively. Click **OK**.

Exporting Signal Group Data

To export the data that define a Signal Builder block's signal groups to the MATLAB workspace, select **Export to workspace** from the block's **File** menu. A dialog box appears.



Variable name: channels

The Signal Builder exports the data by default to a workspace variable named `channels`. To export to a differently named variable, enter the variable's name in the **Variable name** field. Click **OK**. The Signal Builder exports the data to the workspace as the value of the specified variable. The exported data is an array of structures.

Simulating with Signal Groups

You can use standard simulation commands to run models containing Signal Builder blocks or you can use the Signal Builder's **Run all** command (see "Running All Signal Groups" on page 6-51).

Activating a Signal Group

During a simulation, a Signal Builder block always outputs the active signal group. The active signal group is the group selected in the **Signal Builder** dialog box for that block, if the dialog box is open, otherwise the group that was selected when the dialog box was last closed. To activate a group, open the group's **Signal Builder** dialog box and select the group.

Running Different Signal Groups in Succession

The Signal Builder's toolbar includes the standard Simulink buttons for running a simulation. This facilitates running several different signal groups in succession. For example, you can open the dialog box, select a group, run a simulation, select another group, run a simulation, etc., all from the Signal Builder's dialog box.

Running All Signal Groups

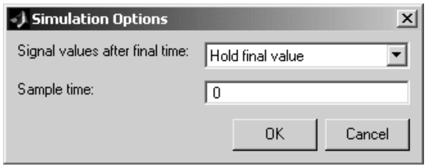
To run all the signal groups defined by a Signal Builder block, open the block's dialog box and select the **Run all** button from the Signal Builder's toolbar.

The **Run all** command runs a series of simulations, one for each signal group defined by the block. If you have installed the optional Model Coverage Tool on your system, the **Run all** command configures the tool to collect and save coverage data for each simulation in the MATLAB workspace and display a report of the combined coverage results at the end of the last simulation. This allows you to quickly determine how well a set of signal groups tests your model.

Note To stop a series of simulations started by the **Run all** command, enter **Control-c** at the MATLAB command line.

Simulation Options Dialog Box

The **Simulation Options** dialog box allows you to specify simulation options pertaining to the Signal Builder. To display the dialog box, select **Simulation Options** from the Signal Builder's **File** menu. The dialog box appears.


Signal values after final time: Hold final value
Sample time: 0

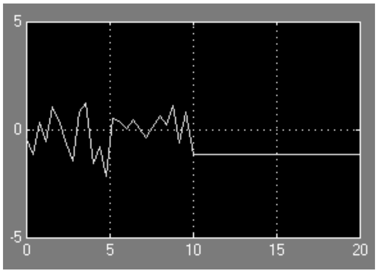
The dialog box allows you to specify the following options.

Signal values after final time

The setting of this control determines the output of the Signal Builder block if a simulation runs longer than the period defined by the block. The options are

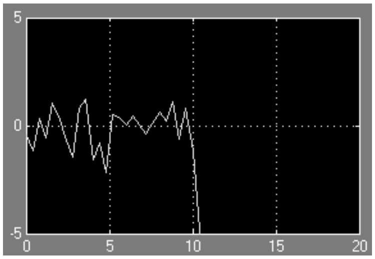
- `Hold final value`

  Selecting this option causes the Signal Builder block to output the last defined value of each signal in the currently active group for the remainder of the simulation.



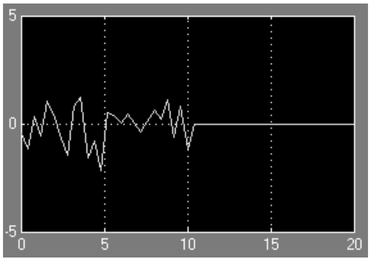
- `Extrapolate`

  Selecting this option causes the Signal Builder block to output values extrapolated from the last defined value of each signal in the currently active group for the remainder of the simulation.



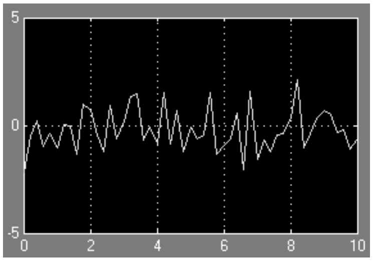
- `Set to zero`

  Selecting this option causes the Signal Builder block to output zero for the remainder of the simulation.

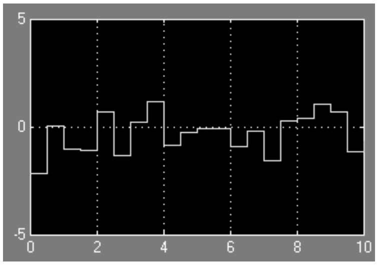


Sample time

Determines whether the Signal Builder block outputs a continuous (the default) or a discrete signal. If you want the block to output a continuous signal, enter 0 in this field. For example, the following display shows the output of a Signal Builder block set to output a continuous Gaussian waveform over a period of 10 seconds.



If you want the block to output a a discrete signal, enter the sample time of the signal in this field. The following example shows the output of a Signal Builder block set to emit a discrete Gaussian waveform having a `0.5` second sample time.

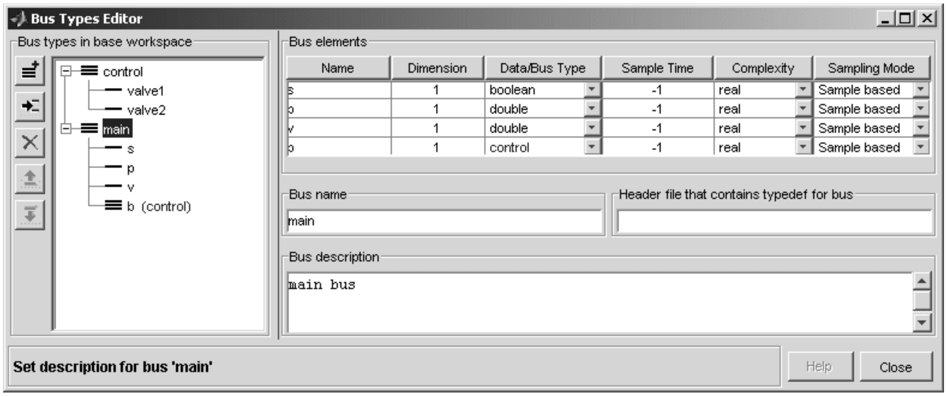


Bus Editor

The Simulink Bus Editor allows you to change the properties of bus type objects, i.e., instances of `Simulink.Bus` class. You can open the Bus Editor in any of the following ways:

- Select **Bus Editor** from the model editor's **Tools** menu.
- Select the **Launch Bus Editor** button on a bus object's dialog box in the Model Explorer.
- Enter `buseditor` at the MATLAB command line.

After you have performed any of these actions, the Bus Editor appears.



Bus elements

Name	Dimension	Data/Bus Type	Sample Time	Complexity	Sampling Mode
s	1	boolean	-1	real	Sample based
p	1	double	-1	real	Sample based
v	1	double	-1	real	Sample based
b	1	control	-1	real	Sample based

Bus name: main

Header file that contains typedef for bus:

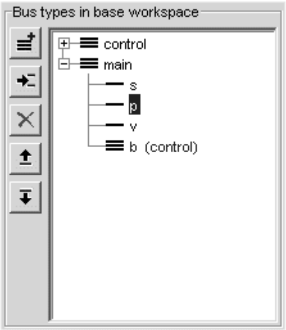
Bus description: main bus

Set description for bus 'main'

The Bus Editor contains the following groups of controls.

Bus types in base workspace

This group contains a bus object hierarchy pane and a column of editing command buttons.



Bus Object Hierarchy Pane

The bus object hierarchy pane displays the structure of bus objects in the Simulink base (i.e., MATLAB) workspace. The pane displays each object as an expandable tree control. The root node of the tree displays the name of the MATLAB variable that references the bus object and, if the bus contains any elements, a button for expanding and collapsing the node. Expanding a bus node displays nodes representing the bus's top-level elements. Each element node displays the element's name. If the element is itself a bus object, the element appears as a bus node that can itself be expanded and collapsed. Selecting any top-level bus object node displays the bus object's properties in the control groups to the right of the bus object hierarchy pane (see below). Selecting any element displays the element's properties in the Bus Editor's **Bus elements** table.

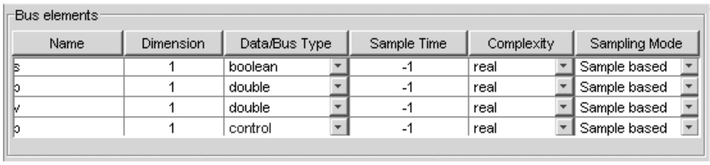
Editing Buttons

This group of buttons allows you to create and modify bus objects in the Simulink base (MATLAB) workspace. It includes the following buttons.

Command	Icon	Description
Create	≡+	Create a bus object in the Simulink base (MATLAB) workspace.
Insert	→≡	Insert an element in the bus object selected in the Bus Editor's bus object hierarchy pane.
Delete	×	Delete the bus or bus element selected in the Bus Editor's bus object hierarchy pane.
Move Up	⤒	Move the selected element up in the list of a bus object's elements.
Move Down	⤓	Move the selected element down in the list of a bus object's elements.

Bus elements

This table displays the properties of the top-level elements of the bus object selected in the bus object hierarchy pane or of the selected element.



Bus elements

Name	Dimension	Data/Bus Type	Sample Time	Complexity	Sampling Mode
s	1	boolean	-1	real	Sample based
b	1	double	-1	real	Sample based
v	1	double	-1	real	Sample based
b	1	control	-1	real	Sample based

The table's cells contain controls that enable you to change the displayed property values. See the documentation for `Simulink.BusElement` class for a description of the usage and valid values for each property.

Bus name

Specifies the name of the workspace variable that references the selected bus object.

Header file

Name of a C header file that defines the user-defined type corresponding to this bus. Simulink ignores this field, which is used by Real-Time Workshop.

Bus description

Description of this bus. Simulink ignores this field.

7

Working with Data

The following sections explain how to specify the data types of signals and parameters and how to create data objects.

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 types 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 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 performance and decrease the size of the code generated from the model.

Simulink performs extensive checking before and during a simulation to ensure that your model is *typesafe*, that is, that code generated from the model will not overflow or underflow and thus produce incorrect results. Simulink models that use the default data type (`double`) are inherently typesafe. Thus, if you never plan to generate code from your model or use a nondefault data type in your models, you can skip the remainder of this section.

On the other hand, if you plan to generate code from your models and use nondefault data types, read the remainder of this section carefully, especially the section on data type rules (see "Data Typing Rules" on page 7-6). In that way, you can avoid introducing data type errors that prevent your model from running to completion or simulating at all.

Data Types Supported by Simulink

Simulink supports all built-in MATLAB data types except `int64` and `uint64`. The term *built-in data type* refers to data types defined by MATLAB itself as opposed to data types defined by MATLAB users. Unless otherwise specified, the term data type in the Simulink documentation refers to built-in data types.

The following table lists the built-in MATLAB data types supported by Simulink.

Name	Description
`double`	Double-precision floating point
`single`	Single-precision floating point
`int8`	Signed 8-bit integer
`uint8`	Unsigned 8-bit integer
`int16`	Signed 16-bit integer
`uint16`	Unsigned 16-bit integer
`int32`	Signed 32-bit integer
`uint32`	Unsigned 32-bit integer

Besides the built-in types, Simulink defines a `boolean` (1 or 0) type, instances of which are represented internally by `uint8` values. Many Simulink blocks also support fixed-point data types. See "Simulink Blocks" in the online Simulink documentation for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

Fixed-Point Data

Simulink allows you to create models that use fixed-point numbers to represent signals and parameter values. Use of fixed-point data can reduce the memory requirements and increase the speed of code generated from a model.

To simulate a fixed-point model, you must have the Simulink Fixed Point product installed on your system. If Simulink Fixed Point is not installed on your system, you can simulate a fixed-point model as a floating-point model by enabling automatic conversion of fixed-point data to floating-point data during simulation. See "Fixed-Point Settings Interface" on page 7-4 for more information. If you do not have Simulink Fixed Point installed and do not enable automatic conversion of fixed-point to floating-point data, Simulink displays an error when you try to simulate a fixed-point model.

You can edit a model containing fixed-point blocks without Simulink Fixed Point. However, you must have Simulink Fixed Point to

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types
- Run a model containing fixed-point data types
- Generate code from a model containing fixed-point data types
- Log the minimum and maximum values produced by a simulation
- Automatically scale the output of a model using the autoscaling tool

Fixed-Point Settings Interface

Most of the functionality in the Fixed-Point Settings interface is for use with the Simulink Fixed Point product. However, even if you do not have Simulink Fixed Point, you can use the Fixed-Point Settings interface to perform a data type override that allows you to work with a fixed-point model.

If you do not have Simulink Fixed Point, you can work with a model containing Simulink blocks with fixed-point settings by doing the following:

1. Access the **Fixed-Point Settings** interface from the model by selecting **Tools -> Fixed-Point Settings**.

2. Set the **Logging mode** parameter to `Force off` model wide.

3. Set the **Data type override** parameter to `True doubles` or `True singles` model wide.

This procedure allows you to share fixed-point Simulink models among people in your company who may or may not have Simulink Fixed Point.

Block Support for Data and Numeric Signal Types

All Simulink blocks accept signals of type `double` by default. Some blocks prefer `boolean` input and others support multiple data types on their inputs. See “Simulink Blocks” in the online Simulink documentation for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

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 8-bit integer of value 2
`int32(3+2i)`	Specifies a complex value whose real and imaginary parts are 32-bit integers

You can specify any MATLAB built-in data type supported by Simulink as the data type of a parameter (see “Data Types Supported by Simulink” on page 7-2). You cannot specify fixed-point data types as parameter data types.

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 the Simulink **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 7-7 for more information.

Data Typing Rules

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

- Signal data types generally do not affect parameter data types, and vice versa.

  A significant exception to this rule is the Constant block, whose output data type is determined by the data type of its parameter.
- If the output of a block is a function of an input and a parameter, and the input and parameter differ in type, Simulink converts the parameter to the input type before computing the output.

  See "Typecasting Parameters" on page 7-7 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 non-`double` 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 `boolean` input. This ensures compatibility with models created by earlier versions of Simulink that support only `double` data type. You can enable strict Boolean type checking by clearing the **Implement logic signals as boolean data** option on the **Optimization** panel of the **Configuration Parameters** dialog box (see "The options are typically to do nothing or to display a warning or an error message (see "Diagnosing Simulation Errors" on page 10-72). A warning message does not terminate a simulation, but an error message does." on page 10-48).

Typecasting Signals

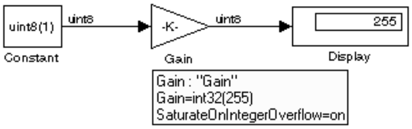
Simulink displays an error whenever it detects that a signal is connected to a block that does not accept the signal's data type. If you want to create such a connection, you must explicitly typecast (convert) the signal to a type that the block does accept. You can use the Data Type Conversion block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

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 to this rule occur:

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

  Consider, for example, the following model.

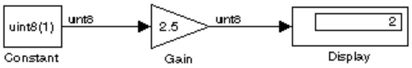


This model uses a Gain block to amplify a constant input signal. Computing the output of the Gain block requires computing the product of the input signal and the gain. Such a computation requires that the two values be of the same data type. However, in this case, the data type of the signal, `uint8`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

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

- If the signal data type can represent the parameter value but only at reduced precision, Simulink optionally issues a warning message and continues the simulation (see "Parameter precision loss" on page 10-53).

  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 allows you to create entities called data objects that specify values, data types, tunability, value ranges, and other key attributes of block outputs and parameters. You can assign such objects to workspace variables and use the variables in Simulink dialog boxes to specify parameter and signal attributes. This allows you to make model-wide changes to parameter and signal specifications simply by changing the values of a few variables. In other words, Simulink objects allow you to parameterize the specification of a model's data attributes.

Note This section uses the term *data* to refer generically to signals and parameters.

Simulink allows you to create various types of data objects, each intended to be used to specify a particular type of data attribute or set of data attributes, such as a data item's type or value. The rest of this section describes features and procedures for working with data objects that apply to all data objects regardless of type. For information on working with specific kinds of data object, see the "Data Object Classes" section of the Simulink Reference.

About Data Object Classes

Simulink uses objects called data classes to define the properties of specific types of data objects. The classes also define functions, called methods, for creating and manipulating instances of particular types of objects. Simulink provides a set of built-in classes for specifying specific types of attributes (see "Data Object Classes" on page 8-1 for information on these built-in classes). Some MathWorks products based on Simulink, such as Real-Time Workshop, also provide classes for specifying data attributes specific to their applications. See the documentation for those products for information on the classes they provide. You can also create subclasses of some of these built-in classes to specify attributes specific to your applications (see "Subclassing Simulink Data Classes" on page 7-18).

Simulink uses memory structures called *packages* to store the code and data that implement data classes. The classes provided by Simulink reside in the Simulink package. Classes provided by products based on Simulink reside in packages provided by those products. You can create your own packages for storing the classes that you define.

Class Naming Convention

Simulink uses dot notation to name classes:

```
PACKAGE.CLASS
```

where CLASS is the name of the class and PACKAGE is the name of the package to which the class belongs, for example, `Simulink.Parameter`. This notation allows you to create and reference identically named classes that belong to different packages. In this notation, the name of the package is said to qualify the name of the class.

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

Handle Versus Value Classes

Simulink data object classes fall into two categories: value classes and handle classes. An instance of a *value* class is permanently associated with the MATLAB variable to which it is initially assigned. Reassigning or passing the variable to a function causes MATLAB to create and assign or pass a copy of the original object. An instance of a *handle* class is associated with a handle object. The handle can be assigned to multiple variables or passed to functions without causing a copy of the original object to be created. A program can modify an instance of a handle class by modifying any variable that references it. Most Simulink data object classes are value classes. Exceptions include `Simulink.Signal` and `Simulink.Parameter` class.

About Data Object Methods

Data classes define functions, called methods, for creating and manipulating the objects that they define. Constructors (see below) are the only methods that you need to use Simulink data objects. The other methods defined by the Simulink built-in data classes are intended for internal use and are not documented.

Constructors

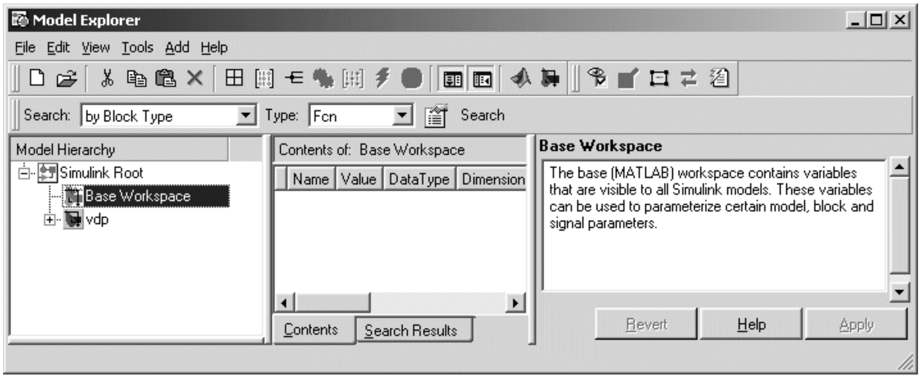
Every data class defines a method for creating instances of that class. The name of the method is the same as the name of the class. For example, the name of the `Simulink.Parameter` class's constructor is `Simulink.Parameter`. The constructors defined by Simulink data classes take no arguments. A constructor returns a handle to the instance that it creates if the class of the instance is a handle class; otherwise, it returns the instance itself (see "Handle Versus Value Classes" on page 7-11).

Creating Data Objects

You can use either the Model Explorer (see next topic) or MATLAB commands (see "Using MATLAB Commands to Create Data Objects" on page 7-13) to create Simulink data objects.

Using the Model Explorer to Create Data Objects

To use the Model Explorer (see "The Model Explorer" on page 9-2) to create data objects, first select the workspace in which you want to create the object in the Model Explorer's **Model Hierarchy** pane.

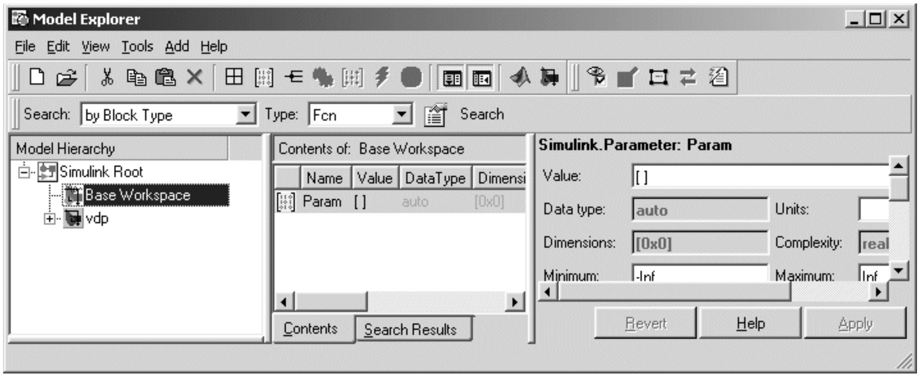


Search: by Block Type

Type: Fcn

The base (MATLAB) workspace contains variables that are visible to all Simulink models. These variables can be used to parameterize certain model, block and signal parameters.

Then, select the type of the object that you want to create (e.g., **Simulink Parameter** or **Simulink Signal**) from the Model Explorer's **Add** menu or from its toolbar. Simulink creates the object, assigns it to a variable in the selected workspace, and displays its properties in the Model Explorer's **Contents** and **Dialog** panes.



Search: by Block Type

Type: Fcn

Contents of: Base Workspace

Name	Value	DataType	Dimensi
Param	[ ]	auto	[0x0]

Simulink.Parameter: Param

Value: [ ]

Data type: auto

Units:

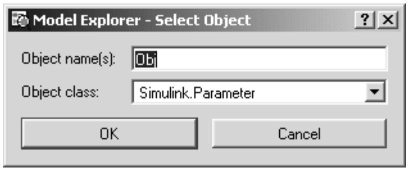
Dimensions: [0x0]

Complexity: real

Minimum: -Inf

Maximum: Inf

If the type of object you want to create does not appear on the **Add** menu, select **Find Custom** from the menu. Simulink searches the MATLAB path for all data object classes derived from Simulink class on the MATLAB path, including types that you have created, and displays the result in a dialog box.



Object name(s): Obj

Object class: Simulink.Parameter

Select the type of object (or objects) that you want to create from the **Object Class** list and enter the names of the workspace variables to which you want the objects to be assigned in the **Object name(s)** field. Simulink creates the specified objects and displays them in the Model Explorer's **Contents** pane.

Using MATLAB Commands to Create Data Objects

You can use data object constructors to create instances of data classes at the MATLAB command line or in MATLAB scripts. For example, the following command creates an instance of a Simulink parameter object:

```
gain = Simulink.Parameter;
```

Note Because `Simulink.Parameter` is a handle class (see “Handle Versus Value Classes” on page 7-11, the value of `gain` is a handle to the newly created object rather than the object itself. You could thus create additional references to the object by assigning `gain` to other variables.

About Object Properties

Object properties are variables associated with an object that specify properties of the entity that the object represents, for example, the size of a data type. The object’s class defines the names, value types, default values, and valid value ranges of the object’s properties.

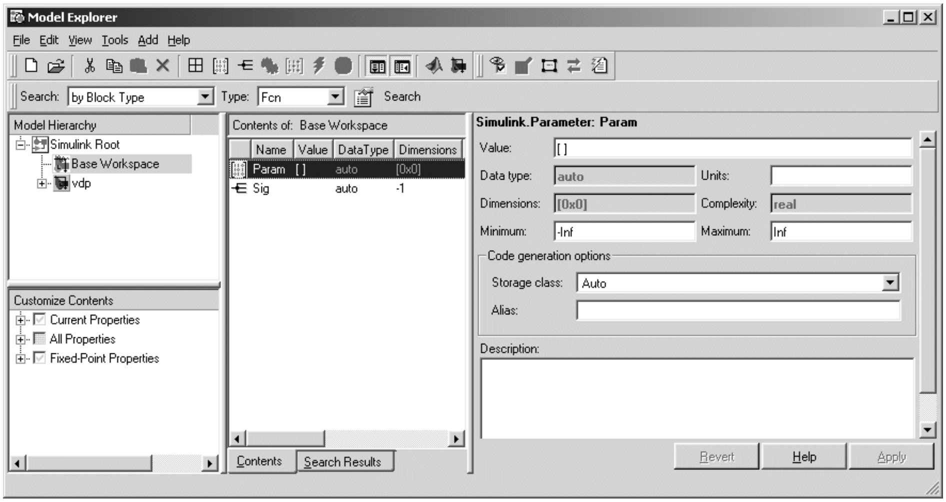
Changing Object Properties

You can use either the Model Explorer (see next topic) or MATLAB comands to change a data object's properties (see "Using MATLAB Commands to Change an Object's Properties" on page 7-16).

Using the Model Explorer to Change an Object's Properties

To use the Model Explorer to change an object's properties, select the workspace that contains the object in the Model Explorer's **Model Hierarchy** pane. Then select the object in the Model Explorer's **Contents** pane.

The Model Explorer displays the object's property dialog box in its **Dialog** pane (if the pane is visible).



Search: by Block Type Type: Fcn

Contents of: Base Workspace

Name	Value	DataType	Dimensions
Param	[ ]	auto	[0x0]
Sig		auto	-1

Simulink.Parameter: Param

Value: [ ]

Data type: auto Units:

Dimensions: [0x0] Complexity: real

Minimum: -Inf Maximum: Inf

Code generation options

Storage class: Auto

Alias:

Description:

You can configure the Model Explorer to display some or all of the object's properties in the **Contents** pane (see "Customizing the Contents Pane" on page 9-5). To edit a property, click its value in the **Contents** or **Dialog** pane. The value is replaced by a control that allows you to change the value.

Using MATLAB Commands to Change an Object's Properties

You can also use MATLAB commands to get and set data object properties. Use the following dot notation in MATLAB commands and programs to get and set a data object's properties:

```
VALUE = OBJ.PROPERTY;
OBJ.PROPERTY = VALUE;
```

where OBJ is a variable that references either the object if it is an instance of a value class or a handle to the object if the object is an instance of a handle class (see "Handle Versus Value Classes" on page 7-11), PROPERTY is the property's name, and VALUE is the property's value. For example, the following MATLAB code creates a data type alias object (i.e., an instance of Simulink.AliasType) and sets its base type to uint8:

```
gain= Simulink.AliasType;
gain.DataType = 'uint8';
```

Use dot notation recursively to get and set the properties of objects that are values of other object's properties, e.g.,

```
gain.RTWInfo.StorageClass = 'ExportedGlobal';
```

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.

Creating Persistent Data Objects

To create parameter and signal objects that persist across Simulink sessions, first write a script that creates the objects or create the objects with the Simulink **Data Explorer** (see “Subclassing Simulink Data Classes” on page 7-18) or at the command line and save them in a MAT-file (see “Saving and Loading Data Objects” on page 7-16). 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.

Subclassing Simulink Data Classes

The Simulink **Data Class Designer** allows you to create subclasses of some Simulink classes. To define a class with the **Data Class Designer**, you enter the package, name, parent class, properties, and other characteristics of the class in a dialog box. The **Data Class Designer** then generates P-code that defines the class. You can also use the **Data Class Designer** to change the definitions of classes that it created, for example, to add or remove properties.

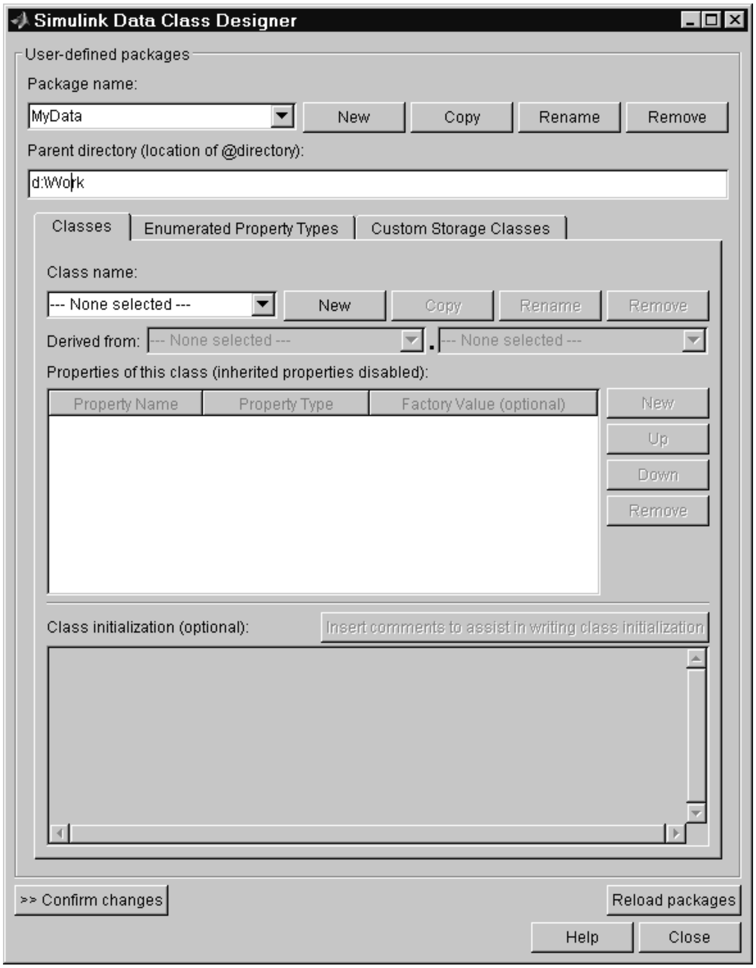
Note You can use the **Data Class Designer** to create custom storage classes. See the Real-Time Workshop documentation for information on custom storage classes.

Creating a Data Object Class

To create a class with the **Data Class Designer**:

1 Select **Data class designer** from the Simulink **Tools** menu.

The **Data Class Designer** dialog box appears.



Package name: MyData

Parent directory (location of @directory): d:\Work

Class name: --- None selected ---

Derived from: --- None selected --- . --- None selected ---

Simulink Data Class Designer
User-defined packages
New
Copy
Rename
Remove
Classes
Enumerated Property Types
Custom Storage Classes
New
Copy
Rename
Remove
Properties of this class (inherited properties disabled):
Property Name
Property Type
Factory Value (optional)
New
Up
Down
Remove
Class initialization (optional):
Insert comments to assist in writing class initialization
>> Confirm changes
Reload packages
Help
Close

2 Select the name of the package in which you want to create the class from the **Package name** list.

   Do not create a class in any of the Simulink built-in packages, i.e., packages in `matlabroot/toolbox/simulink`. See “Creating a Class Package” on page 7-28 for information on creating your own class packages.

3 Click the **New** button on the **Classes** pane of the **Data Class Designer** dialog box.

4 Enter the name of the new class in the **Class name** field on the **Classes** pane.

Note The name of the new class must be unique in the package to which the new class belongs. Class names are case sensitive. For example, Simulink considers `Signal` and `signal` to be names of different classes.

---

5 Press **Enter** or click the **OK** button on the **Classes** pane to create the specified class in memory.

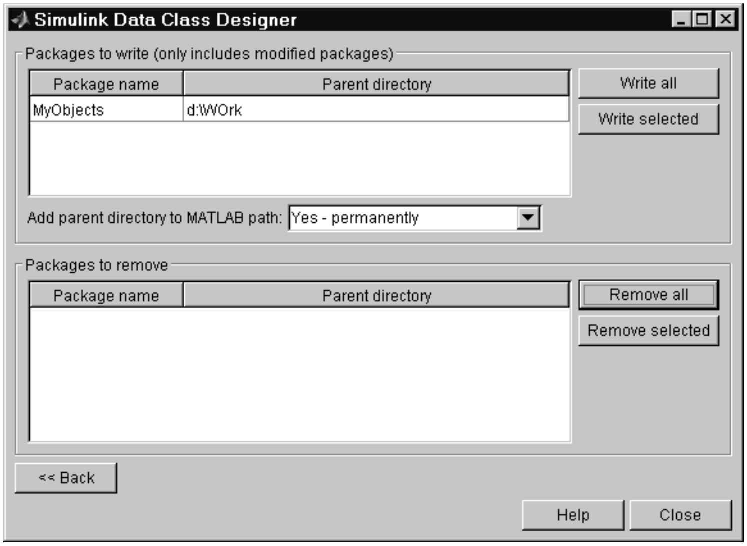
6 Select a parent class for the new class (see “Specifying a Parent for a Class” on page 7-22).

7 Define the properties of the new class (see “Defining Class Properties” on page 7-23).

8 If necessary, create initialization code for the new class (see “Creating Initialization Code” on page 7-27).

9 Click **Confirm changes**.

Simulink displays the **Confirm changes** pane.



Packages to write (only includes modified packages)

Package name	Parent directory
MyObjects	d:\WOrk

Add parent directory to MATLAB path: Yes - permanently

Packages to remove

Package name	Parent directory

10 Click **Write all** or select the package containing the new class definition and click **Write selected** to save the new class definition.

You can also use the **Classes** pane to perform the following operations.

Copy a class. To copy a class, select the class in the **Classes** pane and click **Copy**. Simulink creates a copy of the class under a slightly different name. Edit the name, if desired, click **Confirm changes**, and click **Write all** or, after selecting the appropriate package, **Write selected** to save the new class.

Rename a class. To rename a class, select the class in the **Classes** pane and click **Rename**. The **Class name** field becomes editable. Edit the field to reflect the new name. Save the package containing the renamed class, using the **Confirm changes** pane.

Remove a class from a package. To remove a class definition from the currently selected package, select the class in the **Classes** pane and click **Remove**.

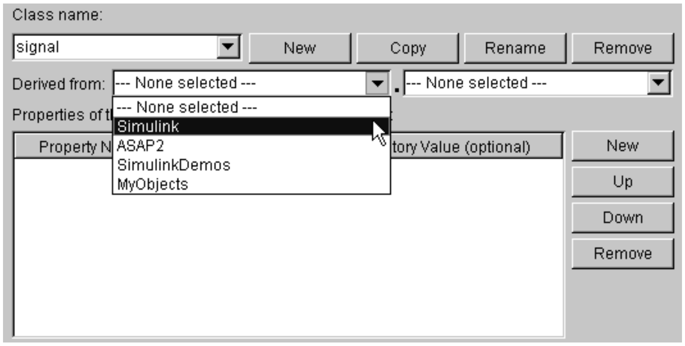
Simulink removes the class from the in-memory definition of the class. Save the package that formerly contained the class.

Specifying a Parent for a Class

To specify a parent for a class:

1 Select the name of the class from the **Class name** field on the **Classes** pane.

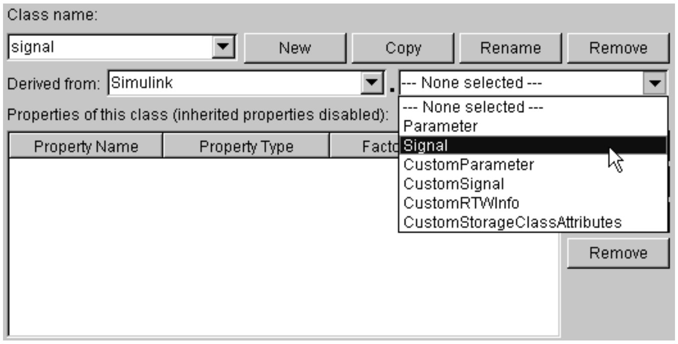
2 Select the package name of the parent class from the left-hand **Derived from** list box.



Class name: signal

Derived from: --- None selected --- . --- None selected ---

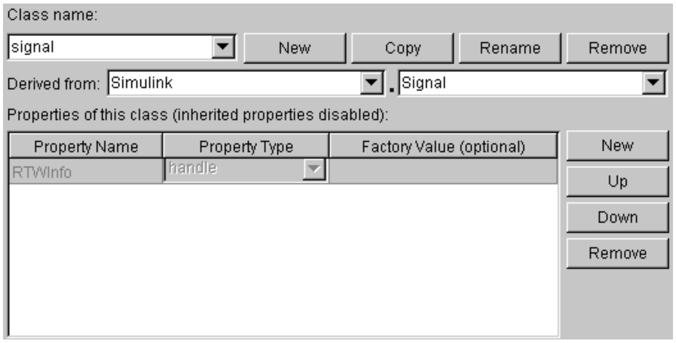
3 Select the parent class from the right-hand **Derived from** list.



Class name: signal

Derived from: Simulink . --- None selected ---

Simulink displays properties of the selected class derived from the parent class in the **Properties of this class** field.



Class name: signal

Derived from: Simulink . Signal

Properties of this class (inherited properties disabled):

Property Name	Property Type	Factory Value (optional)
RTWInfo	handle	

Simulink grays the inherited properties to indicate that they cannot be redefined by the child class.

4 Save the package containing the class.

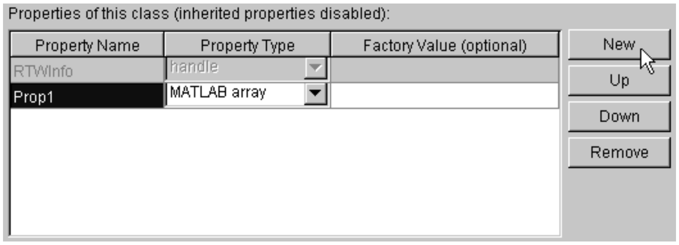
Defining Class Properties

To add a property to a class:

1 Select the name of the class from the **Class name** field on the **Classes** pane.

2 Select the **New** button next to the **Properties of this class** field on the **Classes** pane.

Simulink creates a property with a default name and value and displays the property in the **Properties of this class** field.



Properties of this class (inherited properties disabled):

Property Name	Property Type	Factory Value (optional)
RTWInfo	handle	
Prop1	MATLAB array	

3 Enter a name for the new property in the **Property Name** column.

Note The property name must be unique to the class. Unlike class names, property names are not case sensitive. For example, Simulink treats `Value` and `value` as referring to the same property.

4 Select the data type of the property from the **Property Type** list.

The list includes built-in property types and any enumerated property types that you have defined (see "Defining Enumerated Property Types" on page 7-25).

5 If you want the property to have a default value, enter the default value in the **Factory Value** column.

The default value is the value the property has when an instance of the associated class is created. The initialization code for the class can override this value (see "Creating Initialization Code" on page 7-27 for more information).

The following rules apply to entering factory values for properties:

- Do not use quotation marks when entering the value of a string property. Simulink treats the value that you enter as a literal string.
- The value of a MATLAB array property can be any expression that evaluates to an array, cell array, structure, or object. Enter the expression exactly as you would enter the value on the command line, for example, `[0 1; 1 0]`. Simulink evaluates the expression that you enter to check its validity. Simulink displays a warning message if evaluating the expression results in an error. Regardless of whether an evaluation error occurs, Simulink stores the expression as the factory value of the property. This is because an expression that is invalid at define time might be valid at run-time.
- You can enter any expression that evaluates to a numeric value as the value of a `double` or `int32` property. Simulink evaluates the expression and stores the result as the property's factory value.

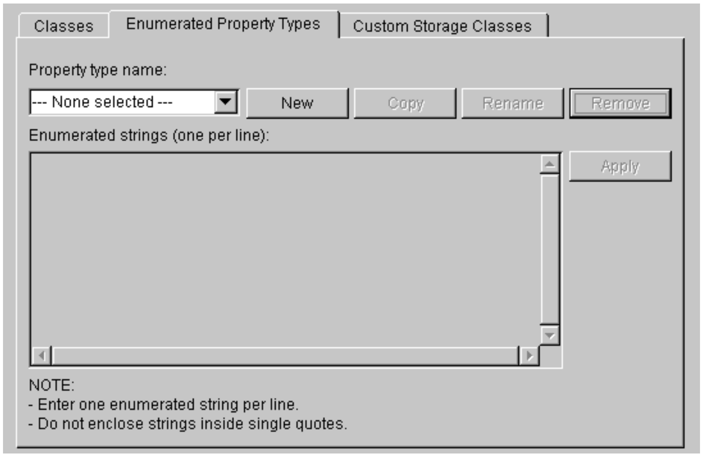
6 Save the package containing the class with new or changed properties.

Defining Enumerated Property Types

An *enumerated property type* is a property type whose value must be one of a specified set of values, for example, `red`, `blue`, or `green`. An enumerated property type is valid only in the package that defines it.

To create an enumerated property type:

1 Select the **Enumerated Property Types** pane of the **Data Class Designer**.

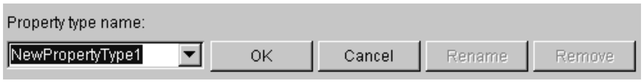


Property type name: --- None selected ---

NOTE:
- Enter one enumerated string per line.
- Do not enclose strings inside single quotes.

2 Click the **New** button next to the **Property type name** field.

Simulink creates an enumerated type with a default name.



Property type name: NewPropertyType1

3 Change the default name in the **Property type name** field to the desired name for the property.

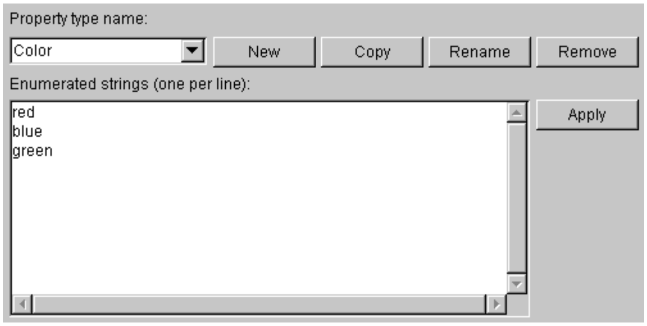
The currently selected package defines an enumerated property type and the type can be referenced only in the package that defines it. However, the name of the enumerated property type must be globally unique. There cannot be any other built-in or user-defined enumerated property with the same name. If you enter the name of an existing built-in or user-defined enumerated property for the new property, Simulink displays an error message.

4. Click the **OK** button.

   Simulink creates the new property in memory and enables the **Enumerated strings** field on the **Enumerated Property Types** pane.

5. Enter the permissible values for the new property type **Enumerated strings** field, one per line.

   For example, the following **Enumerated strings** field shows the permissible values for an enumerated property type named `Color`.



Property type name: Color

Enumerated strings (one per line):

```
red
blue
green
```

6. Click **Apply** to save the changes in memory.

7. Click **Confirm changes.** Then click **Write all** to save this change.

You can also use the **Enumerated Property Type** pane to copy, rename, and remove enumerated property types.

- Click the **Copy** button to copy the currently selected property type. Simulink creates a new property that has a new name, but has the same value set as the original property.
- Click the **Rename** button to rename the currently selected property type. The **Property name** field becomes editable. Edit the field to reflect the new name.
- Click the **Remove** button to remove the currently selected property.

Don't forget to save the package containing the modified enumerated property type.

Creating Initialization Code

You can specify code to be executed when Simulink creates an instance of a data object class. To specify initialization code for a class, select the class from the **Class name** field of the **Data Class Designer** and enter the initialization code in the **Class initialization** field.

The **Data Class Designer** inserts the code that you enter in the **Class initialization** field in the class instantiation function of the corresponding class. Simulink invokes this function when it creates an instance of this class. The class instantiation function has the form

```
function h = ClassName(varargin)
```

where `h` is the handle to the object that is created and `varargin` is a cell array that contains the function's input arguments.

By entering the appropriate code in the **Data Class Designer**, you can cause the instantiation function to perform such initialization operations as

- Error checking
- Loading information from data files
- Overriding factory values
- Initializing properties to user-specified values

For example, suppose you want to let a user initialize the `ParamName` property of instances of a class named `MyPackage.Parameter`. The user does this by passing the initial value of the `ParamName` property to the class constructor:

```
Kp = MyPackage.Parameter('Kp');
```

The following code in the instantiation function would perform the required initialization:

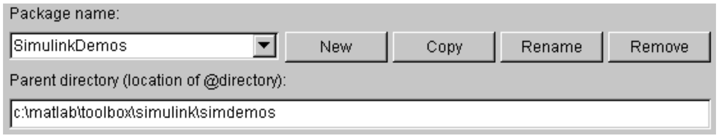
```
switch nargin
   case 0
       % No input arguments - no action
   case 1
       % One input argument
       h.ParamName = varargin{1};
```

```
    otherwise
        warning('Invalid number of input arguments');
end
```

Creating a Class Package

To create a new package to contain your classes:

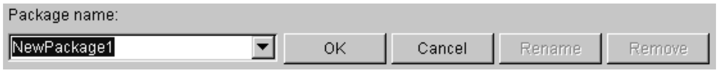
1 Click the **New** button next to the **Package name** field of the **Data Class Designer.**



Package name: SimulinkDemos

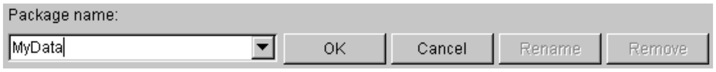
Parent directory (location of @directory): c:\matlab\toolbox\simulink\simdemos

Simulink displays a default package name in the **Package name** field.



Package name: NewPackage1

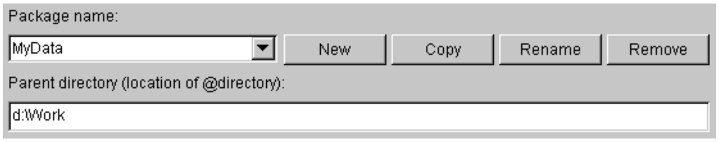
2 Edit the **Package name** field to contain the package name that you want.



Package name: MyData

3 Click **OK** to create the new package in memory.

4 In the package **Parent directory** field, enter the path of the directory where you want Simulink to create the new package.



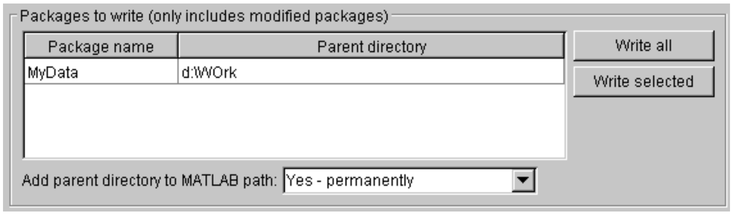
Package name: MyData

Parent directory (location of @directory): d:\Work

Simulink creates the specified directory, if it does not already exist, when you save the package to your file system in the succeeding steps.

5 Click the **Confirm changes** button on the **Data Class Designer.**

Simulink displays the **Packages to write** panel.



Packages to write (only includes modified packages)

Package name	Parent directory
MyData	d:\WOrk

Add parent directory to MATLAB path: Yes - permanently

6 To enable use of this package in the current and future sessions, ensure that the **Add parent directory to MATLAB path** box is selected (the default).

This adds the path of the new package's parent directory to the MATLAB path.

7 Click **Write all** or select the new package and click **Write selected** to save the new package.

You can also use the **Data Class Designer** to copy, rename, and remove packages.

Copying a package. To copy a package, select the package and click the **Copy** button next to the **Package name** field. Simulink creates a copy of the package under a slightly different name. Edit the new name, if desired, and click **OK** to create the package in memory. Then save the package to make it permanent.

Renaming a package. To rename a package, select the package and click the **Rename** button next to the **Package name** field. The field becomes editable. Edit the field to reflect the new name. Save the renamed package.

Removing a package. To remove a package, select the package and click the **Remove** button next to the **Package name** field to remove the package from memory. Click the **Confirm changes** button to display the **Packages to remove** panel. Select the package and click **Remove selected** to remove the package from your file system or click **Remove all** to remove all packages that you have removed from memory from your file system as well.

Associating User Data with Blocks

You can use the Simulink set_param command to associate your own data with a block. For example, the following command associates the value of the variable mydata with the currently selected block.

```
set_param(gcb, 'UserData', mydata)
```

The value of mydata can be any MATLAB data type, including arrays, structures, objects, and Simulink data objects.

Use get_param to retrieve the user data associated with a block.

```
get_param(gcb, 'UserData')
```

The following command causes Simulink to save the user data associated with a block in the model file of the model containing the block.

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

Note If persistent UserData for a block contains any Simulink data objects, the directories containing the definitions for the classes of those objects must be on the MATLAB path when you open the model containing the block.

Modeling with Simulink

The following sections provides tips and guidelines for creating Simulink models.

Modeling Equations

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

Converting Celsius to Fahrenheit

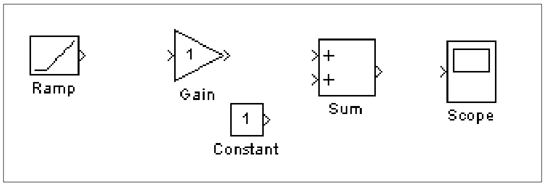
To model the equation that converts Celsius temperature to Fahrenheit

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

First, consider the blocks needed to build the model:

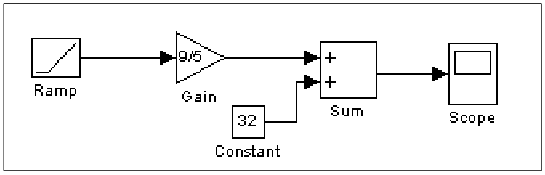
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library
- A Gain block to multiply the input signal by 9/5, from the Math library
- A Sum block to add the two quantities, also from the Math library
- A Scope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

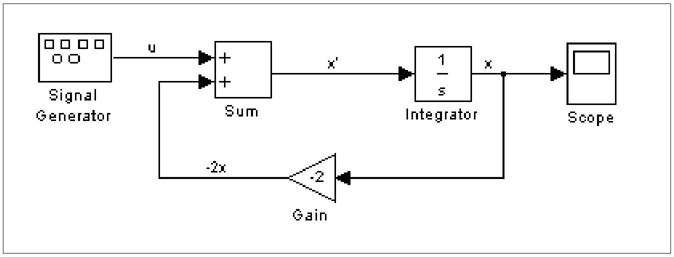
Modeling a Continuous System

To model the differential equation

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

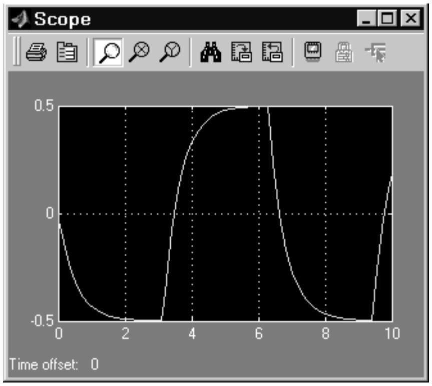
where *u(t)* is a square wave with an amplitude of 1 and a frequency of 1 rad/sec. The Integrator block integrates its input x' to produce x. Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator block and select the Square Wave form but change the default units to radians/sec. Again, view the output using a Scope block. Gather the blocks and define the gain.

In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see “Drawing a Branch Line” on page 4-12. Now you can connect all the blocks.



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

The Scope displays x at each time step. For a simulation lasting 10 seconds, the output looks like this:



The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts u as input and outputs x. So, the block implements x/u. If you substitute sx for x' in the above equation, you get

$$sx = -2x + u$$

Solving for x gives

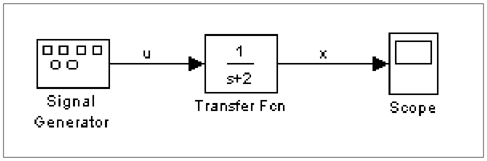
$$x = u/(s+2)$$

or,

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

The Transfer Fcn block uses parameters to specify the numerator and denominator coefficients. In this case, the numerator is 1 and the denominator is s+2. Specify both terms as vectors of coefficients of successively decreasing powers of s.

In this case the numerator is [1] (or just 1) and the denominator is [1 2].



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

Avoiding Invalid Loops

Simulink allows you to connect the output of a block directly or indirectly (i.e., via other blocks) to its input, thereby, creating a loop. Loops can be very useful. For example, you can use loops to solve differential equations diagramatically (see "Modeling a Continuous System" on page 8-3) or model feedback control systems. However, it is also possible to create loops that cannot be simulated. Common types of invalid loops include:

- Loops that create invalid function-call connections or an attempt to modify the input/output arguments of a function call
- Self-triggering subsystems and loops containing non-latched triggered subsystems
- Loops containing action subsystems

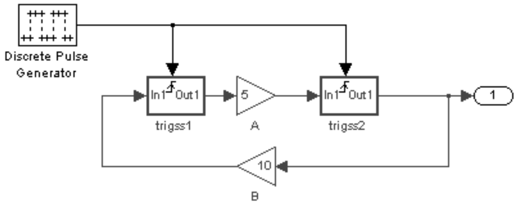
The Subsystem Examples block library in the Ports & Subsystems library contains models that illustrates examples of valid and invalid loops involving triggered and function-call subsystems. Examples of invalid loops include the following models:

- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1`
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2`
- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call systems/sl_subsys_fcncallerr3`

You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update diagram** from the model's **Edit** menu. If the model contains invalid loops, Simulink highlights the loops



and displays an error message in the Simulation Diagnostics Viewer.

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 4-20. The Model Browser provides useful information about complex models (see “The Model Browser” on page 9-22).

- 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 6-38 and “Annotating Diagrams” on page 4-16.

- 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.

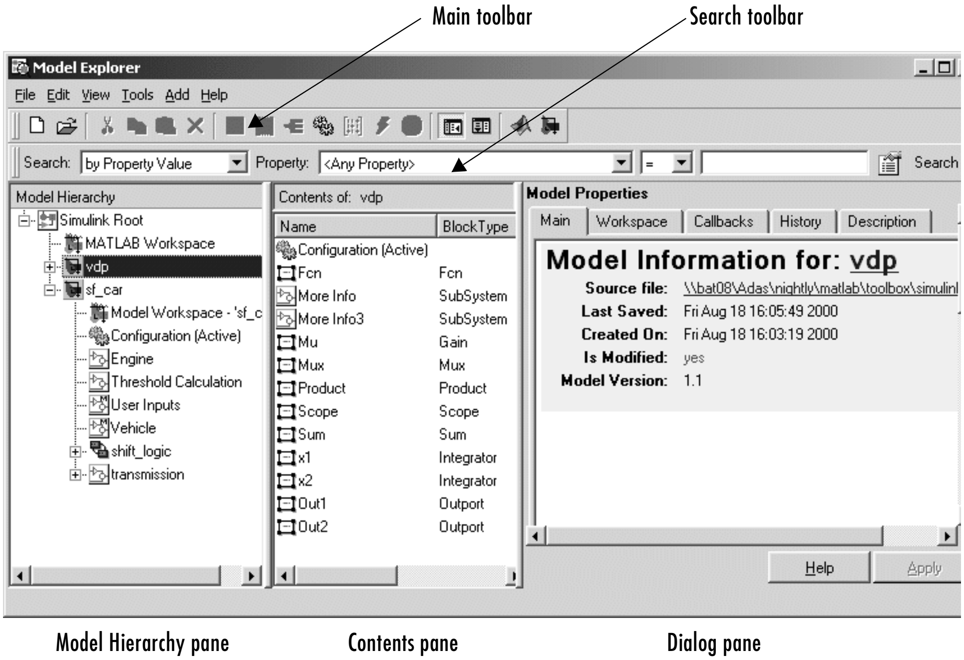
9

Exploring, Searching, and Browsing Models

The following sections describe tools that enable you to quickly navigate to any point in a model and find and modify objects in a model.

The Model Explorer

The Model Explorer allows you to quickly locate, view, and change elements of a Simulink model or Stateflow chart. To display the Model Explorer, select **Model Explorer** from the Simulink **View** menu or select an object in the block diagram and select **Explore** from its context menu. The Model Explorer appears.



- Simulink Root
  - MATLAB Workspace
  - vdp
  - sf_car
    - Model Workspace - 'sf_c
    - Configuration (Active)
    - Engine
    - Threshold Calculation
    - User Inputs
    - Vehicle
    - shift_logic
    - transmission

Name	BlockType
Configuration (Active)	
Fcn	Fcn
More Info	SubSystem
More Info3	SubSystem
Mu	Gain
Mux	Mux
Product	Product
Scope	Scope
Sum	Sum
x1	Integrator
x2	Integrator
Out1	Outport
Out2	Outport

Model Information for: vdp

Source file: \\bat08\Adas\nightly\matlab\toolbox\simulinl
Last Saved: Fri Aug 18 16:05:49 2000
Created On: Fri Aug 18 16:03:19 2000
Is Modified: yes
Model Version: 1.1

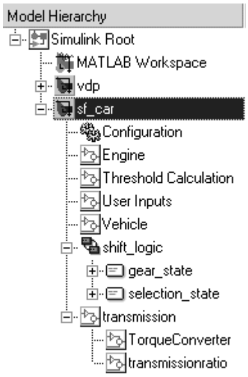
The Model Explorer includes the following components:

- **Model Hierarchy** pane (see "Model Hierarchy Pane" on page 9-3)
- **Contents** pane (see "Contents Pane" on page 9-4)
- **Dialog** pane (see "Dialog Pane" on page 9-9)
- **Main** toolbar (see "Main Toolbar" on page 9-9)
- **Search** bar (see "Search Bar" on page 9-11)

You can use the Model Explorer's **View** menu to hide the **Dialog** pane and the toolbars, thereby making more room for the other panes.

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy.



Simulink Root

The first node in the view represents the Simulink root. Expanding the root node displays nodes representing the MATLAB workspace (Simulink's base workspace) and each model and library loaded in the current session.

Base Workspace

This node represents the MATLAB workspace. The MATLAB workspace is the base workspace for Simulink models. Variables defined in this workspace are visible to all open Simulink models, i.e., to all models whose nodes appear beneath the **Base Workspace** node in the **Model Hierarchy** pane.

Model Nodes

Expanding a model node displays nodes representing the model's configuration sets (see "Configuration Sets" on page 10-26), top-level subsystems, model references, and Stateflow charts. Expanding a node representing a subsystem displays its subsystems, if any. Expanding a node representing a Stateflow chart displays the chart's top-level states. Expanding a node representing a state shows its substates.

Displaying Node Contents

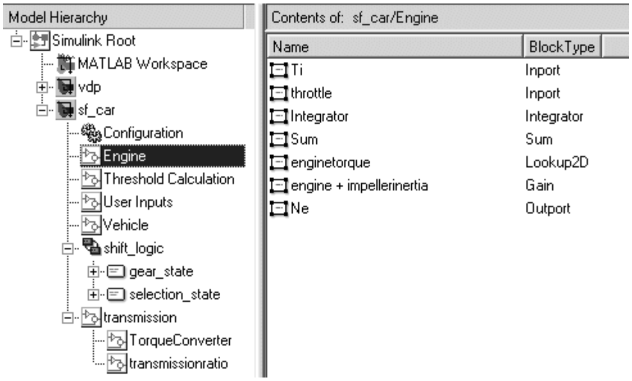
To display the contents of an object displayed in the **Model Hierarchy** pane (e.g., a model or configuration set) in the adjacent **Contents** pane, select the object. To open a graphical object (e.g., a model, subsystem, or chart) in an editor window, right-click the object. A context menu appears. Select **Open** from the context menu. To open an object's properties dialog, select **Properties** from the object's context menu or from the **Edit** menu. See "Configuration Sets" on page 10-26 for information on using the **Model Hierarchy** pane to delete, move, and copy configuration sets from one model to another.

Expanding Model References

To expand a node representing a model reference (see "Referencing Models" on page 4-53), you must first open the referenced model. To do this, right-click on the node to display its context menu, then select **Open Model** from the menu. Simulink opens the model to which the reference refers, displays a node for it in the **Model Hierarchy** pane, and make all references to the model expandable. You cannot edit the contents of a reference node, however. To edit the referenced model, you must expand its node.

Contents Pane

The **Contents** pane displays a tabular view of the contents of the object selected in the **Model Hierarchy** pane or the results of a search operation (see "Search Bar" on page 9-11).



Contents of: sf_car/Engine

Name	BlockType
Ti	Inport
throttle	Inport
Integrator	Integrator
Sum	Sum
enginetorque	Lookup2D
engine + impellerinertia	Gain
Ne	Outport

The table rows correspond to objects (e.g., blocks or states); the table columns, to object properties (e.g., name and type). The table cells display the values of the properties of the objects contained by the object selected in the **Model Hierarchy** pane.

The objects and properties displayed in the **Contents** pane depend on the type of object (e.g., subsystem, chart, or configuration set) selected in the **Model Hierarchy** pane. For example, if the object selected in the **Model Hierarchy** pane is a model or subsystem, the **Contents** pane by default displays the name and type of the top-level blocks contained by that model or subsystem. If the selected object is a Stateflow chart or state, the **Contents** pane by default shows the name, scope, and other properties of the events and data that make up the chart or state.

Customizing the Contents Pane

The Model Explorer's **View** menu allows you to control the type of objects and properties displayed in the **Contents** pane.

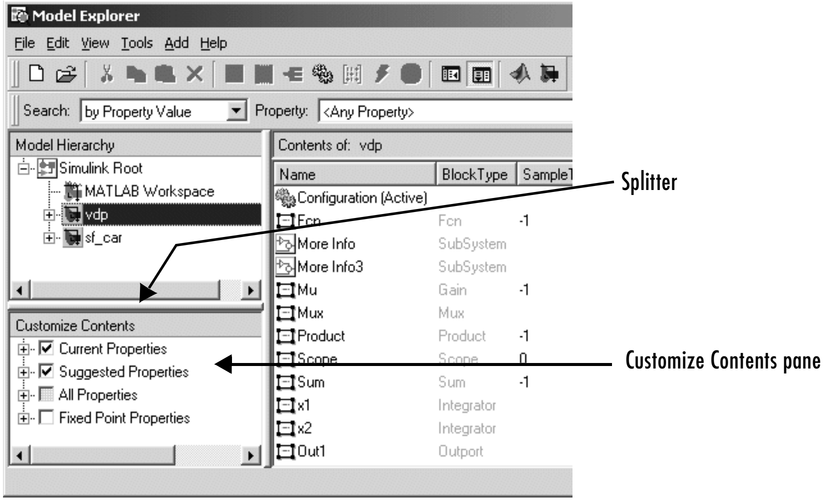
- To display only object names in the **Contents** pane, uncheck the **Show Properties in List View** item on the **View** menu.
- To customize the set or properties displayed in the **Contents** pane, select **Customize Contents** from the **View** menu or click the **Customize Contents** button on the Model Explorer's main toolbar (see "Main Toolbar" on page 9-9). The **Customize Contents** pane appears. Use the pane to select the properties you want the **Contents** pane to display.
- To specify the types of subsystem or chart contents displayed in the **Contents** pane, select **List View Options** from the **View** menu. A menu of object types appears. Check the types that you want to be displayed (e.g., **Blocks** and **Named Signals/Connections** or **All Simulink Objects** for models and subsystems).

Reordering the Contents Pane

The **Contents** pane by default displays its contents in ascending order by name. To order the contents in ascending order by any other displayed property, click the head of the column that displays the property. To change the order from ascending to descending, or vice versa, click the head of the property column that determines the current order.

Customize Contents Pane

The **Customize Contents** pane allows you to select the properties that the **Contents** pane displays for the object selected in the **Model Hierarchy** pane. When visible, the pane appears in the lower left corner of the Model Explorer window.



Search: by Property Value

Property: <Any Property>

Contents of: vdp

Name	BlockType	SampleT
Configuration (Active)		
Fcn	Fcn	-1
More Info	SubSystem	
More Info3	SubSystem	
Mu	Gain	-1
Mux	Mux	
Product	Product	-1
Scope	Scope	0
Sum	Sum	-1
x1	Integrator	
x2	Integrator	
Out1	Outport	

☑ Current Properties
☑ Suggested Properties
▨ All Properties
□ Fixed Point Properties

A splitter divides the **Customize Contents** pane from the **Model Hierarchy** pane above it. Drag the splitter up or down to adjust the relative size of the two panes.

The **Customize Contents** pane contains a tree-structured property list. The list's top-level nodes group object properties into the following categories:

- `Current Properties`

  Properties that the **Contents** pane currently displays.

- `Suggested Properties`

  Properties that Simulink suggests that the **Contents** pane should display, based on the type of object selected in the **Model Hierarchy** pane and the contents of the selected object.

- `All Properties`

  Properties of the contents of all models displayed in the Model Explorer thus far in this session.

- Fixed Point Properties

  Fixed-point properties of blocks.

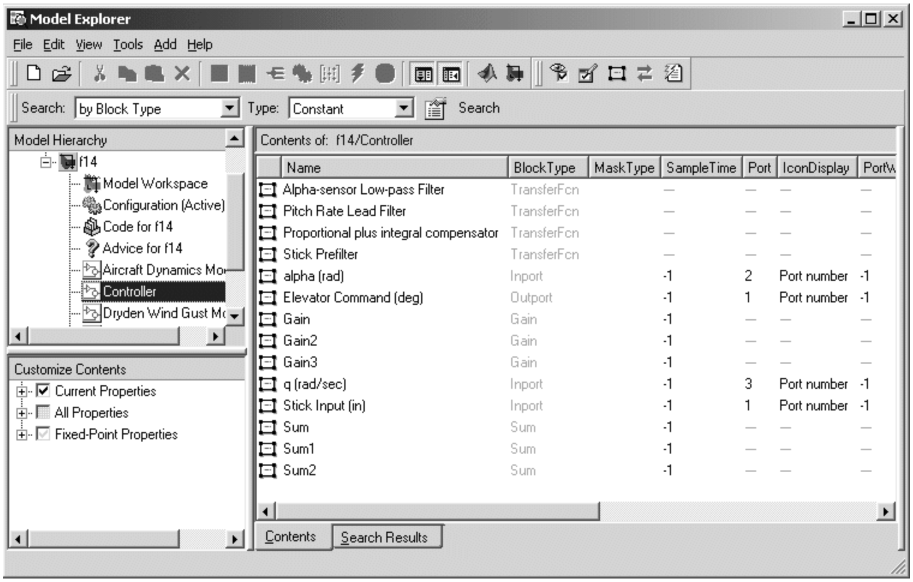
By default, the properties currently displayed in the **Contents** pane are the suggested properties for the currently selected model. The **Customize Contents** pane allows you to perform the following customizations:

- To display additional properties of the selected model, expand the All Properties node, if necessary, and check the desired properties.
- To delete some but not all properties from the **Contents** pane, expand the Current Properties node, if necessary, and uncheck the properties that you do not want to appear in the **Contents** pane.
- To delete all properties from the **Contents** pane (except the selected object's name), uncheck Current Properties.
- To restore the properties suggested for the current model, uncheck Current Properties and check Suggested Properties.
- To add or remove fixed-point block properties from the **Contents** pane, check or uncheck Fixed Point Properties.

Marking Nonexistant Properties

Some of the properties that the Contents pane is configured to display may not apply to all the objects currently listed in the Contents pane. You can configure the Model Explorer to indicate the inapplicable properties.

To do this, select **Mark Nonexistent Properties** from the Model Explorer's **View** menu. The Model Explorer now displays dashes for the values of properties that do not apply to the objects displayed in the **Contents** pane.



Search: by Block Type
Type: Constant

Contents of: f14/Controller

Name	BlockType	MaskType	SampleTime	Port	IconDisplay	PortW
Alpha-sensor Low-pass Filter	TransferFcn		—	—	—	—
Pitch Rate Lead Filter	TransferFcn		—	—	—	—
Proportional plus integral compensator	TransferFcn		—	—	—	—
Stick Prefilter	TransferFcn		—	—	—	—
alpha (rad)	Inport		-1	2	Port number	-1
Elevator Command (deg)	Outport		-1	1	Port number	-1
Gain	Gain		-1	—	—	—
Gain2	Gain		-1	—	—	—
Gain3	Gain		-1	—	—	—
q (rad/sec)	Inport		-1	3	Port number	-1
Stick Input (in)	Inport		-1	1	Port number	-1
Sum	Sum		-1	—	—	—
Sum1	Sum		-1	—	—	—
Sum2	Sum		-1	—	—	—

Changing Property Values

You can change modifiable properties displayed in the **Contents** pane (e.g., a block's name) by editing the displayed value. To edit a displayed value, first select the row that contains it. Then click the value. An edit control replaces the displayed value (e.g., an edit field for text values or a pull-down list for a range of values). Use the edit control to change the value of the selected property.

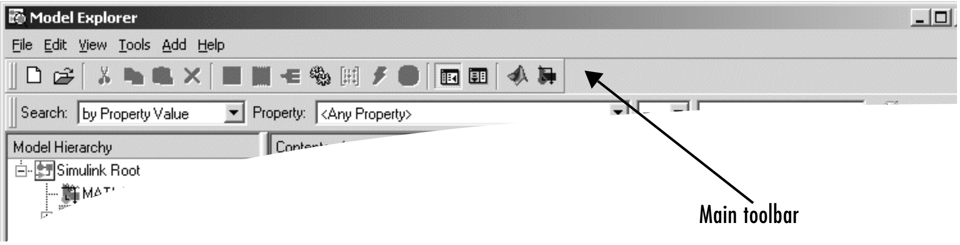
To assign the same property value to multiple objects displayed in the **Contents** pane, select the objects and then change one of the selected objects to have the new property value. The Model Explorer assigns the new property value to the other selected objects as well.

Dialog Pane

The **Dialog** pane displays the dialog view of the object selected in the **Contents** pane, e.g., a block or a configuration subset. You can use the **Dialog** pane to view and change the selected object's properties. To show or hide this pane, select the **Show Dialog View** menu from the Model Explorer's **View** menu or the **Show Dialog View** button on the Model Explorer's main toolbar (see "Main Toolbar" on page 9-9).

Main Toolbar

The Model Explorer's main toolbar appears near the top of the Model Explorer window under the Model Explorer's menu.



The toolbar contains buttons that select commonly used Model Explorer commands:

Button	Usage
🗋	Create a new model.
📂	Open an existing model.
✂	Cut the objects (e.g., variables) selected in the **Contents** pane from the object (e.g., a workspace) selected in the **Model Hierarchy** pane. Save a copy of the object on the system clipboard.

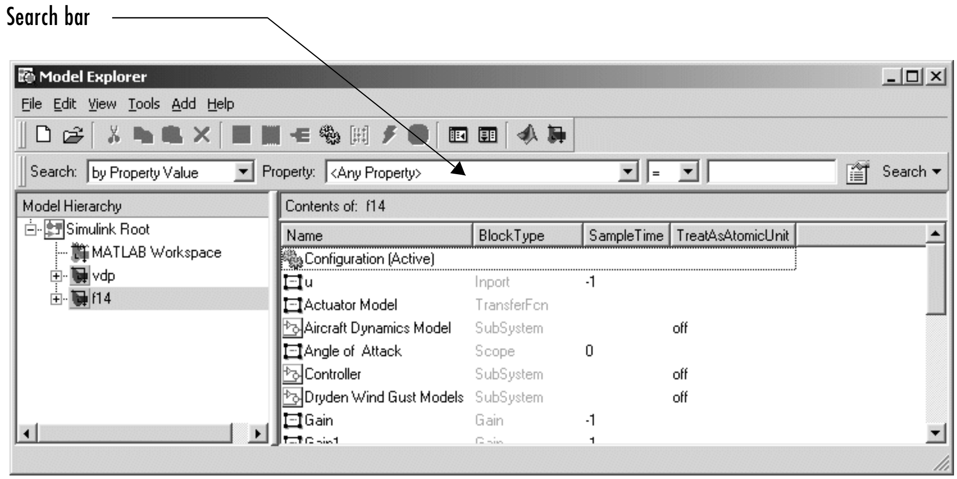
Button	Usage
⧉	Copy the objects selected in the **Contents** pane to the system clipboard.
📋	Paste objects from the clipboard into the object selected in the Model Explorer's **Model Hierarchy** pane.
✕	Delete the objects selected in the **Contents** pane from the object selected in the **Model Hierarchy** pane.
⊞	Add a MATLAB variable to the workspace selected in the **Model Hierarchy** pane.
[⁞]	Add a `Simulink.Parameter` object to the workspace selected in the **Model Hierarchy** pane.
⊣	Add a `Simulink.Signal` object to the workspace selected in the **Model Hierarchy** pane.
⚙	Add a configuration set to the model selected in the **Model Hierarchy** pane.
[⁞]	Add a Stateflow datum to the machine or chart selected in the **Model Hierarchy** pane.
⚡	Add a Stateflow event to the machine or chart selected in the **Model Hierarchy** pane or to the state selected in the Model Explorer.
◎	Add a code generation target to the model selected in the **Model Hierarchy** pane.
▤	Turn the Model Explorer's **Dialog** pane on or off.
▤	Customize the Model Explorer's **Contents** pane.

Button	Usage
	Bring the MATLAB desktop to the front.
	Display the Simulink Library Browser.

To show or hide the main toolbar, select **Main Toolbar** from the Model Explorer's **View** menu.

Search Bar

The Model Explorer's search bar allows you to select, configure, and initiate searches of the object selected in the **Model Hierarchy** pane. It appears at the top of the Model Explorer window.



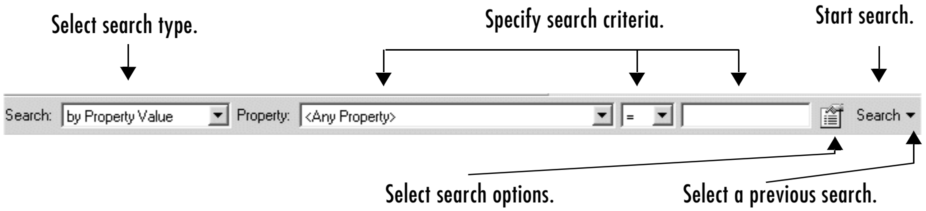
Search: by Property Value Property: <Any Property> =

Contents of: f14

Name	BlockType	SampleTime	TreatAsAtomicUnit
Configuration (Active)			
u	Inport	-1	
Actuator Model	TransferFcn		
Aircraft Dynamics Model	SubSystem		off
Angle of Attack	Scope	0	
Controller	SubSystem		off
Dryden Wind Gust Models	SubSystem		off
Gain	Gain	-1	
Gain1	Gain	-1	

To show or hide the search bar, check or uncheck **Search Bar** in the Model Explorer's **View** menu.

The search bar includes the following controls:



Search Type

Specifies the type of search to be performed. Options include:

- `by Property Value`

  Search for objects whose property matches a specified value. Selecting this search type causes the search bar to display controls that allow you to specify the name of the property, the value to be matched, and the type of match (equals, less than, greater than, etc.).

- `by Property Name`

  Search for objects that have a specified property. Selecting this search type causes the search bar to display a control that allows you to specify the target property's name by selecting from a list of properties that objects in the search domain can have.

- `by Block Type`

  Search for blocks of a specified block type. Selecting this search type causes the search bar to display a block type list control that allows you to select the target block type from the types contained by the currently selected model.

- `for Library Links`

  Searches for library links in the current model.

- `by Class`

  Searches for Simulink objects of a specified class.

- `for Model References`

  Searches a model for references to other models.

- `for Fixed Point`

  Searches a model for all blocks that support fixed-point computations.

- `by Dialog Prompt`

  Searches a model for all objects whose dialogs contain a specified prompt.
- `by String`

  Searches a model for all objects in which a specified string occurs.

Search Options

Specifies options that apply to the current search. The options include:

- `Search All Descendants`

  Search the descendants of the currently selected object as well as the selected object itself.
- `Look Inside Masked Subsystems`

  Search includes masked subsystems.
- `Look Inside Linked Subsystems`

  Search includes linked subsystems.
- `Match Whole String`

  Do not allow partial string matches, e.g., do not allow `sub` to match `substring`.
- `Match Case`

  Consider case when matching strings, e.g., `Gain` does not match `gain`.
- `Regular Expression`

  The Model Explorer considers a string to be matched as a regular expression.
- `Refine Search Uses Boolean 'AND'`

  When refining a search, search for objects that meet both the original and the new search criteria.
- `Refine Search Uses Boolean 'OR'`

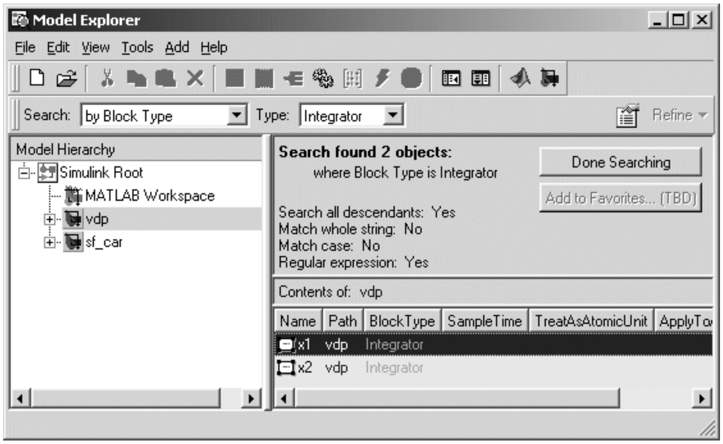
  When refining a search, search for objects that meet either the original or the new search criteria. (Not yet implemented.)
- `Clear Search History`

  Not yet implemented.

Search Button

Initiates the search specified by the current settings of the search bar on the object selected in the Model Explorer's **Model Hierarchy** pane. The Model

Explorer displays the results of the search in its **Contents** pane and enters search mode.



Search: by Block Type Type: Integrator

Search found 2 objects:
where Block Type is Integrator

Search all descendants: Yes
Match whole string: No
Match case: No
Regular expression: Yes

Contents of: vdp

Name	Path	BlockType	SampleTime	TreatAsAtomicUnit	ApplyTo
x1	vdp	Integrator			
x2	vdp	Integrator			

In search mode, you can perform the following tasks:

- Refine the previous search.

  In search mode, a **Refine** button replaces the **Search** button on the search bar. To refine the search results, use the search bar to define new search criteria and then click the **Refine** button. The Model Explorer searches for objects that match the previous search criteria and/or the new criteria, depending on the setting of the refine search options.

- Apply the previous search to another object.

  To apply the previous search to another object, select the object in the Model Explorer's **Model Hierarchy** pane. The Model Explorer repeats the search on the new object and displays the results.

- Edit search results.

  In search mode, you can edit the results displayed in the **Contents** pane just as you can edit them in explore mode. For example, to change all objects found by a search to have the same property value, select the objects in the **Contents** pane and change one of them to have the new property value.

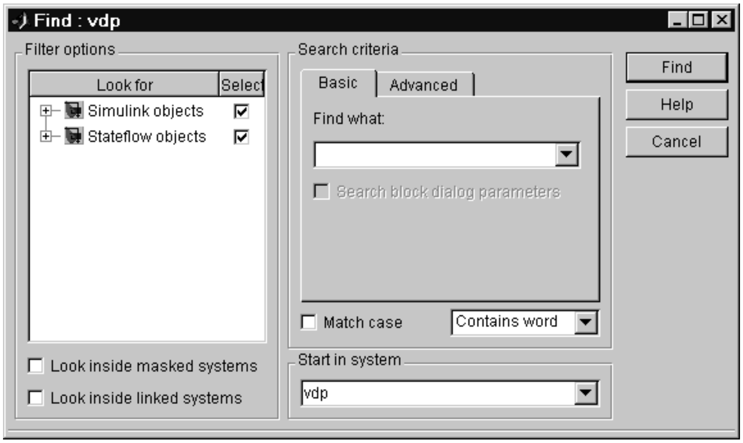
To exit search mode, click the **Done Searching** button at the top of the search results.

Search History

The down arrow control adjacent to the **Search** button displays a list of searches previously executed in the current Simulink session. To reexecute a search, select it from the history and click the **Search** button. (Note this feature is not yet implemented.)

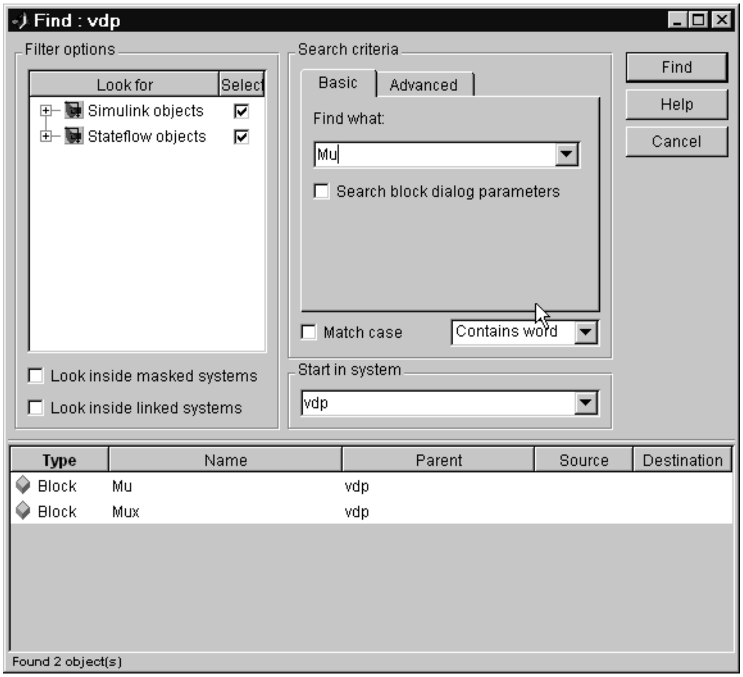
The Finder

The Finder locates blocks, signals, states, or other objects in a model. To display the Finder, select **Find** from the **Edit** menu. The **Find** dialog box appears.



Use the **Filter options** (see “Filter Options” on page 9-18) and **Search criteria** (see “Search Criteria” on page 9-18) panels to specify the characteristics of the object you want to find. Next, if you have more than one system or subsystem open, select the system or subsystem where you want the search to begin from the **Start in system** list. Finally, click the **Find** button. Simulink searches the selected system for objects that meet the criteria you have specified.

Any objects that satisfy the criteria appear in the results panel at the bottom of the dialog box.



Look for	Select
Simulink objects	☑
Stateflow objects	☑

Find what: Mu

□ Search block dialog parameters

□ Match case   Contains word

□ Look inside masked systems

□ Look inside linked systems

Start in system: vdp

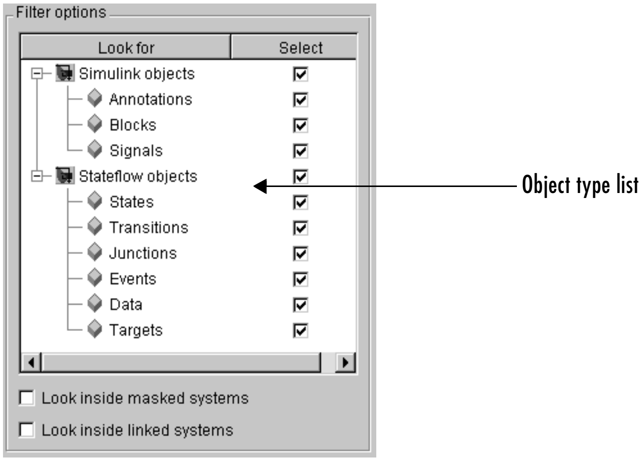
Type	Name	Parent	Source	Destination
Block	Mu	vdp		
Block	Mux	vdp		

Found 2 object(s)

You can display an object by double-clicking its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object. To sort the results list, click any of the buttons at the top of each column. For example, to sort the results by object type, click the **Type** button. Clicking a button once sorts the list in ascending order, clicking it twice sorts it in descending order. To display an object's parameters or properties, select the object in the list. Then press the right mouse button and select **Parameter** or **Properties** from the resulting context menu.

Filter Options

The **Filter options** panel allows you to specify the kinds of objects to look for and where to search for them.



Filter options

Look for	Select
Simulink objects	☑
Annotations	☑
Blocks	☑
Signals	☑
Stateflow objects	☑
States	☑
Transitions	☑
Junctions	☑
Events	☑
Data	☑
Targets	☑

☐ Look inside masked systems

☐ Look inside linked systems

Object type list

Object type list

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

Look inside masked subsystem

Selecting this option causes Simulink to look for objects inside masked subsystems.

Look inside linked systems

Selecting this option causes Simulink to look for objects inside subsystems linked to libraries.

Search Criteria

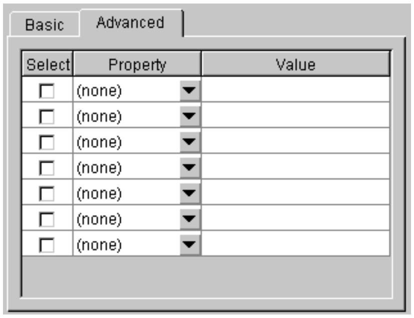
The **Search criteria** panel allows you to specify the criteria that objects must meet to satisfy your search request.

Basic

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

Advanced

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



Select	Property	Value
□	(none)	
□	(none)	
□	(none)	
□	(none)	
□	(none)	
□	(none)	
□	(none)	

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

Match case

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

Other match options

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

- `Match whole word`

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

- `Contains word`

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

- `Regular expression`

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

Character	Meaning
`^`	Matches start of string.
`$`	Matches end of string.
`.`	Matches any character.
`\`	Escape character. Causes the next character to have its ordinary meaning. For example, the regular expression `\..` matches `.a` and `.2` and any other two-character string that begins with a period.
`*`	Matches zero or more instances of the preceding character. For example, `ba*` matches `b`, `ba`, `baa`, etc.
`+`	Matches one or more instances of the preceding character. For example, `ba+` matches `ba`, `baa`, etc.
`[]`	Indicates a set of characters that can match the current character. A hyphen can be used to indicate a range of characters. For example, `[a-zA-Z0-9_]+` matches `foo_bar1` but not `foo$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]`).

Character	Meaning
`\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]`).
`\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

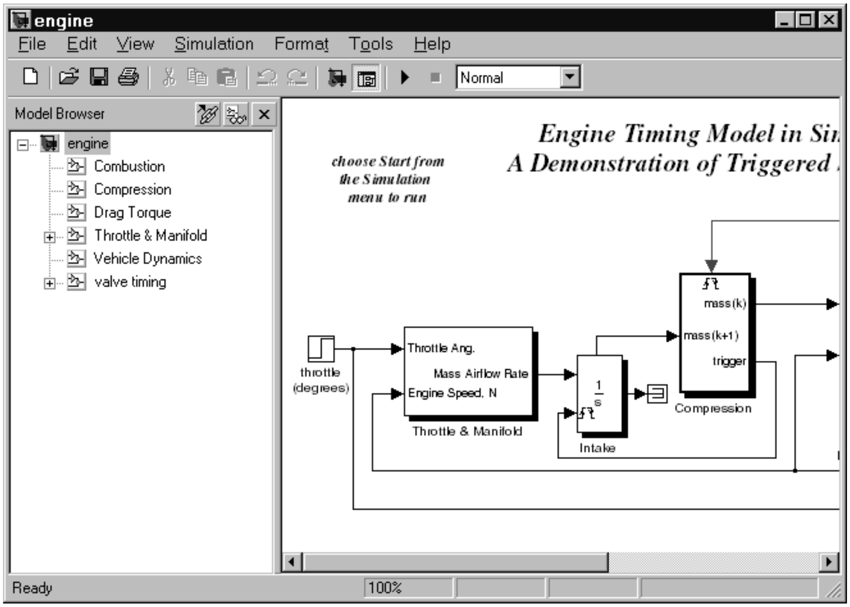
The Model Browser enables you to

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

The browser operates differently on Microsoft Windows and UNIX platforms.

Using the Model Browser on Windows

To display the Model Browser, 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

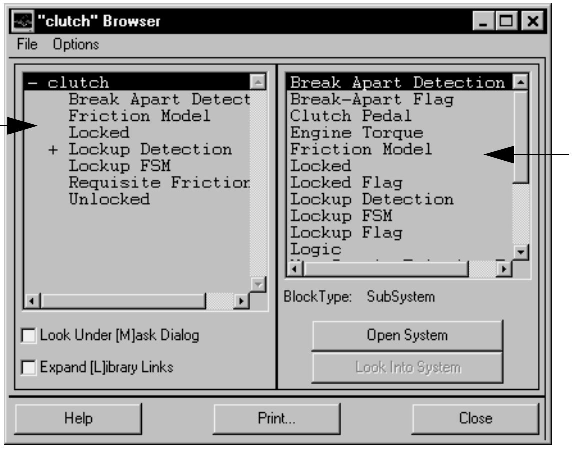
The Model Browser can include or omit library links from the tree view of a model. Use the Simulink **Preferences** dialog box to specify whether to display library links by default. To toggle display of library links, select **Show library links** from the **Model browser options** submenu of the Simulink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



```
- clutch
    Break Apart Detect
    Friction Model
    Locked
  + Lockup Detection
    Lockup FSM
    Requisite Frictior
    Unlocked
```

```
Break Apart Detection
Break-Apart Flag
Clutch Pedal
Engine Torque
Friction Model
Locked
Locked Flag
Lockup Detection
Lockup FSM
Lockup Flag
Logic
```

BlockType: SubSystem

□ Look Under [M]ask Dialog

□ Expand [L]ibrary Links

Current system and subsystems it contains

Blocks in the selected 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 the contents of

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

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

Interpreting List Contents

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

- A plus sign (+) before a system name in the systems list indicates that the system is expandable, which means that it has systems beneath it. Double-click 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 12, "Creating Masked Subsystems."
- [L] indicates that the block is a reference block. For more information, see "Connecting Blocks" on page 4-9.
- [O] indicates that an open function (`OpenFcn`) callback is defined for the block. For more information about block callbacks, see "Using Callback Routines" on page 4-90.
- [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 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 its name or select its name and click the **Open System** button.

- If the system has an [M] or [O] before its name, select the system name and click 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 **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 the block in a block diagram). Similarly, if the block's `OpenFcn` callback parameter is defined, clicking **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 **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.

Running Simulations

The following sections explain how to use Simulink to simulate a dynamic system.

Simulation Basics

Simulating a Simulink model requires only that you start the simulation (see “Starting a Simulation” on page 10-3). However, before starting the simulation, you may want to specify various simulation options, such as the simulation’s start and stop time and the type of solver used to solve the model at each simulation time step. Specifying simulation options is called configuring a simulation. Simulink enables you to create multiple simulation configurations, called configuration sets, for a model, modify existing configuration sets, and switch configuration sets with a click of a mouse button (see “Configuration Sets” on page 10-26 for information on creating and selecting configuration sets).

Once you have defined or selected a simulation configuration set that meets your needs, you can start the simulation. Simulink then runs the simulation from the specified start time to the specified stop time. While the simulation is running, you can interact with the simulation in various ways, stop or pause the simulation (see “Pausing or Stopping a Simulation” on page 10-4), and launch simulations of other models. If an error occurs during a simulation, Simulink halts the simulation and pops up a diagnostic viewer that helps you to determine the cause of the error.

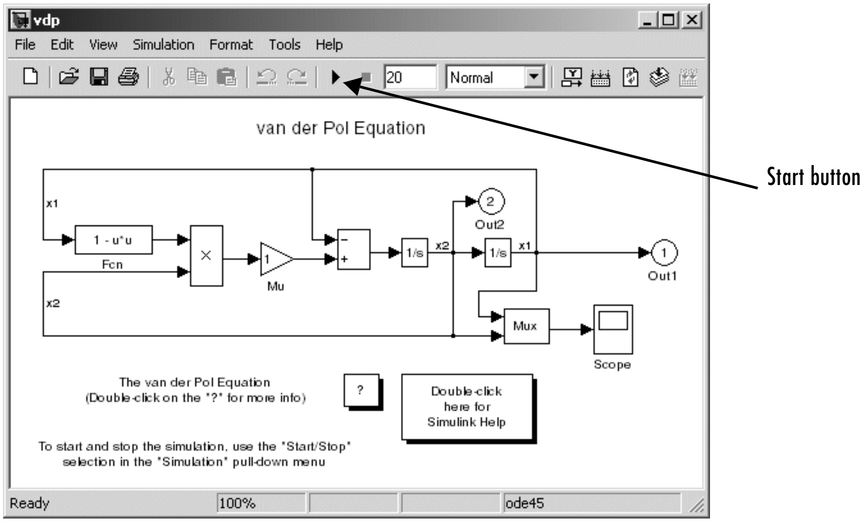
Note The following sections explain how to run a simulation interactively. See “Running a Simulation Programmatically” on page 10-78 for information on running a simulation from a program or the MATLAB command line.

Controlling Execution of a Simulation

The Simulink graphical interface includes menu commands and toolbar buttons that enable you to start, stop, and pause a simulation.

Starting a Simulation

To start execution of a model, select **Start** from the model editor's **Simulation** menu or click the **Start** button on the model's toolbar.



The van der Pol Equation
(Double-click on the "?" for more info)

Double-click
here for
Simulink Help

To start and stop the simulation, use the "Start/Stop"
selection in the "Simulation" pull-down menu

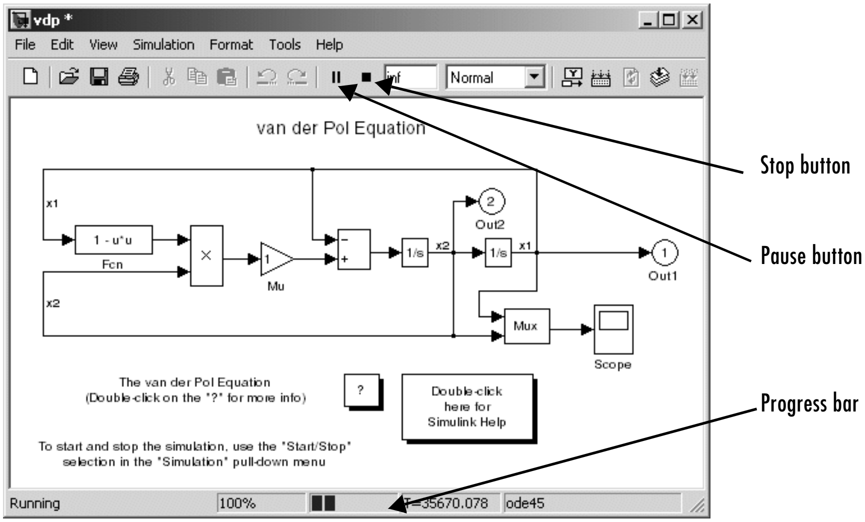
Start button

You can also use the keyboard shortcut, **Ctrl+T**, to start 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.

Simulink starts executing the model at the start time specified on the **Configuration Parameters** dialog box. Execution continues until the simulation reaches the final time step specified on the **Configuration Parameters** dialog box, an error occurs, or you pause or terminate the simulation (see "The Configuration Parameters Dialog Box" on page 10-30).

While the simulation is running, a progress bar at the bottom of the model window shows how far the simulation has progressed. A **Stop** command replaces the **Start** command on the **Simulatio**n menu. A **Pause** command appears on the menu and replaces the **Start** button on the model toolbar.



The van der Pol Equation
(Double-click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

Stop button

Pause button

Progress bar

Your computer beeps to signal the completion of the simulation.

Pausing or Stopping a Simulation

Select the **Pause** command or button to pause the simulation. Simulink completes execution of the current time step and suspends execution of the simulation. When you select **Pause**, the menu item and button change to **Continue**. (The button has the same appearance as the **Start** button). You can resume a suspended simulation at the next time step by choosing **Continue**.

To terminate execution of the model, select the **Stop** command or button. The keyboard shortcut for stopping a simulation is **Ctrl+T**, the same as for starting a simulation. Simulink completes execution of the current time step before terminating the model. Subsequently selecting the **Start** command or button restarts the simulation at the first time step specified on the **Configuration Parameters** dialog box.

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

Interacting with a Running Simulation

You can perform certain operations interactively while a simulation is running. You can

- Modify some configuration parameters, including the stop time and the maximum step size
- Click a line to see the signal carried on that line on a floating (unconnected) Scope or Display block
- Modify the parameters of a block, as long as you do not cause a change in
    - Number of states, inputs, or outputs
    - Sample time
    - Number of zero crossings
    - Vector length of any block parameters
    - Length of the internal block work vectors

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.

Specifying a Simulation Start and Stop Time

Simulink simulations start by default at 0.0 seconds and end at 10.0 seconds. The **Solver** configuration pane allows you to specify other start and stop times for the currently selected simulation configuration. See “The Solver Pane” on page 10-31 for more information.

Note Simulation time and actual clock time are not the same. For example, running a simulation for 10 seconds usually does 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 speed.

Choosing a Solver

A solver is a Simulink software component that determines the next time step that a simulation needs to take to meet target accuracy requirements that you specify. Simulink provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. The following sections explain how to choose the solver best suited to your application. For information on tailoring the selected solver to your model, see “Improving Simulation Accuracy” on page 10-77.

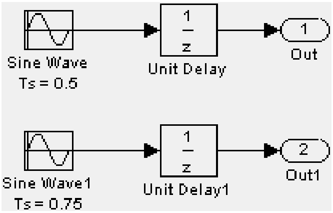
Choosing a Solver Type

Simulink divides solvers into two types: fixed-step and variable-step. Both types of solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the step size. With a fixed-step solver, the step size remains constant throughout the simulation. By contrast, with a variable-step solver, the step size can vary from step to step, depending on the model’s dynamics. In particular, a variable-step solver reduces the step size when a model’s states are changing rapidly to maintain accuracy and increases the step size when the system’s states are changing slowly in order to avoid taking unnecessary steps. The **Type** control on the Simulink **Solver** configuration pane allows you to select either of these two types of solvers (see “The Solver Pane” on page 10-31).

The choice between the two types depends on how you plan to deploy your model and the model’s dynamics. If you plan to generate code from your model and run the code on a real-time computer system, you should choose a fixed-step solver to simulate the model. This is because real-time computer systems operate at fixed-size signal sample rates. A variable-step solver may cause the simulation to miss error conditions that can occur on a real-time computer system.

If you do not plan to deploy your model as generated code, the choice between a variable-step and a fixed-step solver depends on the dynamics of your model. If your model’s states change rapidly or contain discontinuities, a variable-step solver can shorten the time required to simulate your model significantly. This is because, for such a model, a variable-step solver can require fewer time steps than a fixed-step solver to achieve a comparable level of accuracy.

The following model illustrates how a variable-step solver can shorten simulation time for a multirate discrete model.



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

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

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

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

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

The variable-step discrete solver uses zero-crossing detection (see "Zero-Crossing Detection" on page 2-19) to handle continuous signals. Simulink uses this solver by default if you specify a continuous solver and your model has no continuous states.

Choosing a Fixed-Step Solver

When the **Type** control of the **Solver** configuration pane is set to `fixed-step`, the configuration pane's **Solver** control allows you to choose one of the set of fixed-step solvers that Simulink provides. The set of fixed-step solvers comprises two types of solvers: discrete and continuous.

About the Fixed-Step Discrete Solver

The fixed-step discrete solver computes the time of the next time step by adding a fixed step size to the time of the current time. The accuracy and length of time of the resulting simulation depends on the size of the steps taken by the simulation: the smaller the step size, the more accurate the results but the longer the simulation takes. You can allow Simulink to choose the size of the step size (the default) or you can choose the step size yourself. If you allow Simulink to choose the step size, Simulink sets the step size to the fundamental sample time of the model if the model has discrete states or to the result of dividing the difference between the simulation start and stop time by 50 if the model has no discrete states. This choice assures that the simulation will hit every simulation time required to update the model's discrete states at the model's specified sample times

The fixed-step discrete solver has a fundamental limitation. It cannot be used to simulate models that have continuous states. That's because the fixed-step discrete solver relies on a model's blocks to compute the values of the states that they define. Blocks that define discrete states compute the values of those states at each time step taken by the solver. Blocks that define continuous states, on the other hand, rely on the solver to compute the states. Continuous solvers perform this task. You should thus select a continuous solver if your model contains continuous states.

Note If you attempt to use the fixed-step discrete solver to update or simulate a model that has continuous states, Simulink displays an error message. Thus, updating or simulating a model is a quick way to determine whether it has continuous states.

About Fixed-Step Continuous Solvers

Simulink provides a set of fixed-step continuous solvers that, like the fixed-step discrete solver, compute the simulation's next time by adding a fixed-size time step to the current time. In addition, the continuous solvers employ numerical integration to compute the values of a model's continuous states at the current step from the values at the previous step and the values of the state derivatives. This allows the fixed-step continuous solvers to handle models that contain both continuous and discrete states.

Note In theory, a fixed-step continuous solver can handle models that contain no continuous states. However, that would impose an unnecessary computational burden on the simulation. Consequently, Simulink always uses the fixed-step discrete solver for a model that contains no states or only discrete states, even if you specify a fixed-step continuous solver for the model.

Simulink provides two distinct types of fixed-step continuous solvers: explicit and implicit solvers. Explicit solvers (see “Explicit Fixed-Step Continuous Solvers” on page 10-10) compute the value of a state at the next time step as an explicit function of the current value of the state and the state derivative, e.g.,

```
X(n+1) = X(n) + h * DX(n)
```

where X is the state, DX is the state derivative, and h is the step size. An implicit solver (see “Implicit Fixed-Step Continuous Solvers” on page 10-11) computes the state at the next time step as an implicit function of the state and the state derivative at the next time step, e.g.,

```
X(n+1) - X(n) - h*DX(n+1) = 0
```

This type of solver requires more computation per step than an explicit solver but is also more accurate for a given step size. This solver thus can be faster than explicit fixed-step solvers for certain types of stiff systems.

Explicit Fixed-Step Continuous Solvers. Simulink provides a set of explicit fixed-step continuous solvers. The solvers differ in the specific integration technique used to compute the model’s state derivatives. The following table lists the available solvers and the integration techniques they use.

Solver	Integration Technique
ode1	Euler’s Method
ode2	Heun’s Method
ode3	Bogacki-Shampine Formula
ode4	Fourth-Order Runge-Kutta (RK4) Formula
ode5	Dormand-Prince Formula

The integration techniques used by the fixed-step continuous solvers trade accuracy for computational effort. The table lists the solvers in order of the computational complexity of the integration methods they use from least complex (`ode1`) to most complex (`ode5`).

As with the fixed-step discrete solver, the accuracy and length of time of a simulation driven by a fixed-step continuous solver depends on the size of the steps taken by the solver: the smaller the step size, the more accurate the results but the longer the simulation takes. For any given step size, the more computationally complex the solver, the more accurate the simulation.

If you specify a fixed-step solver type for a model, Simulink sets the solver's model to ode3, i.e., it chooses a solver capable of handling both continuous and discrete states with moderate computational effort. As with the discrete solver, Simulink by default sets the step size to the fundamental sample time of the model if the model has discrete states or to the result of dividing the difference between the simulation start and stop time by 50 if the model has no discrete states. This assures that the solver will take a step at every simulation time required to update the model's discrete states at the model's specified sample rates. However, it does not guarantee that the default solver will accurately compute a model's continuous states or that the model cannot be simulated in less time with a less complex solver. Depending on the dynamics of your model, you may need to choose another solver and/or sample time to achieve acceptable accuracy or to shorten the simulation time.

Implicit Fixed-Step Continuous Solvers. Simulink provides one solver in this category: `ode14x`. This solver uses a combination of Newton's method and extrapolation from the current value to compute the value of a model state at the next time step. Simulink allows you to specify the number of Newton's method iterations and the extrapolation order that the solver uses to compute the next value of a model state (see "Fixed-Step Solver Options" on page 10-35). The more iterations and the higher the extrapoloation order that you select, the greater the accuracy but also the greater the computational burden per step size.

Choosing a Fixed-Step Continuous Solver

Any of the fixed-step continuous solvers in Simulink can simulate a model to any desired level of accuracy, given enough time and a small enough step size. Unfortunately, in general, it is not possible, or at least not practical, to decide *a priori* which solver and step size combination will yield acceptable results for a model's continuous states in the shortest time. Determining the best solver for a particular model thus generally requires experimentation.

Here is the most efficient way to choose the best fixed-step solver for your model experimentally. First, use one of the variable-step solvers to simulate your model to the level of accuracy that you desire. This will give you an idea of what the simulation results should be. Next, use `ode1` to simulate your model at the default step size for your model. Compare the results of simulating your model with `ode1` with the results of simulating with the variable-step solver. If the results are the same within the specified level of accuracy, you have found the best fixed-step solver for your model, namely `ode1`. That's because `ode1` is the simplest of the Simulink fixed-step solvers and hence yields the shorted simulation time for the current step size.

If `ode1` does not give accurate results, repeat the preceding steps with the other fixed-step solvers until you find the one that gives accurate results with the least computational effort. The most efficient way to do this is to use a binary search technique. First, try `ode3`. If it gives accurate results, try `ode2`. If `ode2` gives accurate results, it is the best solver for your model; otherwise, `ode3` is the best. If `ode3` does not give accurate results, try `ode5`. If `ode5` gives accurate results, try `ode4`. If ode4 gives accurate results, select it as the solver for your model; otherwise, select `ode5`.

If `ode5` does not give accurate results, reduce the simulation step size and repeat the preceding process. Continue in this way until you find a solver that solves your model accurately with the least computational effort.

Choosing a Variable-Step Solver

When the **Type** control of the **Solver** configuration pane is set to `variable-step`, the configuration pane's **Solver** control allows you to choose one of the set of variable-step solvers that Simulink provides. As with fixed-step solvers in Simulink, the set of variable-step solvers comprises a discrete solver and a subset of continuous solvers. Both types compute the time of the next time step by adding a step size to the time of the current time that varies depending on the rate of change of the model's states. The continuous solvers, in addition, use numerical integration to compute the values of the model's continuous states at the next time step. Both types of solvers rely on blocks that define the model's discrete states to compute the values of the discrete states that each defines.

The choice between the two types of solvers depends on whether the blocks in your model defines states and, if so, the kind of states that they define. If your model defines no states or defines only discrete states, you should select the discrete solver. In fact, if a model has no states or only discrete states, Simulink will use the discrete solver to simulate the model even if the model specifies a continuous solver.

About Variable-Step Continuous Solvers

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

Simulink provides the following variable-step continuous solvers:

- `ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver; that is, in computing `y(`t_n`)`, it needs only the solution at the immediately preceding time point, `y(`t_{n-1}`)`. In general, `ode45` is the best solver to apply as a first try for most problems. For this reason, `ode45` is the default solver used by Simulink for models with continuous states.
- `ode23` is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It can be more efficient than `ode45` at crude tolerances and in the presence of mild stiffness. `ode23` is a one-step solver.
- `ode113` is a variable-order Adams-Bashforth-Moulton PECE solver. It can be more efficient than `ode45` at stringent tolerances. `ode113` is a *multistep* solver; that is, it normally needs the solutions at several preceding time 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 can 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 can be more efficient than `ode15s` at crude tolerances.

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.

Specifying Variable-Step Solver Error Tolerances

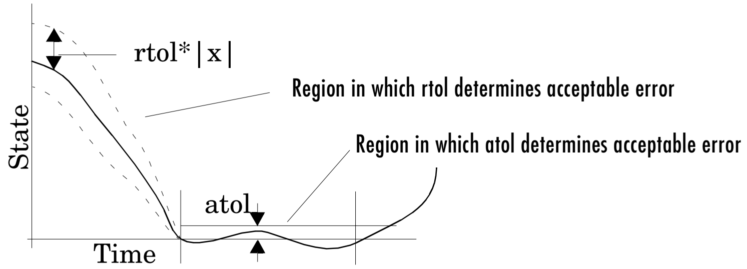
The solvers use standard local error control techniques to monitor the error at each time step. During each time step, the solvers compute the state values at the end of the step and also determine the *local error*, the estimated error of these state values. They then compare the local error to the *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 is accurate to within 0.1%.
- *Absolute tolerance* is a threshold error value. This tolerance represents the acceptable error as the value of the measured state approaches zero.

The error for the `i`th state, e_i, is required to satisfy

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

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


rtol*|x|
Region in which rtol determines acceptable error
Region in which atol determines acceptable error
State
atol
Time

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.

The Integrator, Transfer Fcn, State-Space, and Zero-Pole blocks allow you to specify absolute tolerance values for solving the model states that they compute or that determine their output. The absolute tolerance values that you specify for these blocks override the global settings in the **Configuration Parameters** dialog box. You might want to override the global setting in this way, if the global setting does not provide sufficient error control for all of your model's states, for example, because they vary widely in magnitude.

Importing and Exporting Simulation Data

Simulink allows you to import input signal and initial state data from the MATLAB workspace and export output signal and state data to the MATLAB workspace during simulation. This capability allows you to use standard or custom MATLAB functions to generate a simulated system's input signals and to graph, analyze, or otherwise postprocess the system's outputs. See the following sections for more information:

- "Importing Input Data from the MATLAB Workspace" on page 10-16
- "Exporting Output Data to the MATLAB Workspace" on page 10-20
- "Importing and Exporting States" on page 10-22

Importing Input Data from the MATLAB 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, select the **Input** box in the **Load from workspace** area of the **Data Import/Export** pane (see "Data Import/Export Pane" on page 10-39). Then, enter an external input specification (see below) in the adjacent edit box and click **Apply**.

The input data can take any of the following forms.

Importing Data Arrays

To use this format, select **Input** in the **Load from workspace** pane and select the `Array` option from the **Format** list on the **Data Import/Export** 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.

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 `Array`. So if you define `t` and `u` in the base workspace, you need only select 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 array input format allows you to load only real (noncomplex) scalar or vector data of type `double`. Use the structure format to input complex data, matrix (2-D) data, and/or data types other than `double`.

Using a MATLAB Time Expression to Import Data

You can use a MATLAB time expression to import data from the MATLAB workspace. To use a time expression, enter the expression as a string (i.e., enclosed in single quotes) in the **Input** field of the **Data Import/Export** pane. The time expression can be any MATLAB expression that evaluates to a row vector equal in length to the number of signals entering the model's 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)`.

Importing Data Structures

Simulink can read data from the workspace in the form of a structure whose name is specified in the **Input** text field. You can import structures that include only signal data or both signal and time data.

Importing signal-and-time data structures. To import structures that include both signal and time data, select the `Structure with time` option on from the **Format** list on the **Data Import/Export** pane. The input structure must have two top-level fields: `time` and `signals`. The `time` field contains a column vector of the simulation times. The `signals` field contains an array of substructures, each of which corresponds to a model input port.

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

Note You must set the **Port dimensions** parameter of the Inport to be the same value as the `dimensions` field of the corresponding input structure. If the values differ, Simulink stops and displays an error message when you try to simulate the model.

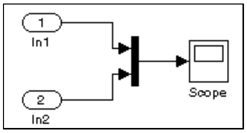
If the inputs for a port are scalar or vector values, the `values` field must be an `M-by-N` array where `M` is the number of time points specified by the `time` field and `N` is the length of each vector value. For example, the following code creates an input structure for loading 11 time samples of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

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

To load this data into the model's inport, you would select the **Input** option on the **Data Import/Export** pane and enter `a` in the input expression field.

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

As another example, consider the following model, which has two inputs.



Suppose that you want to input a sine wave into the first port and a cosine wave into the second port. To do this, define a vector, `a`, as follows, in the 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;
```

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

Importing Signal-Only Structures. The `Structure` format is the same as the `Structure with time` format except that the `time` field is empty. For example, in the preceding example, you could set the `time` field as follows:

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

In this case, 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.

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.

Exporting Output Data to the MATLAB 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 fields 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.

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 `values` field is 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`, `dimensions`, `label`, and `blockName`. The `values` field contains time samples of a state of the block specified by the `blockName` field. The `label` field for built-in blocks indicates the type of state: either `CSTATE` (continuous state) or `DSTATE` (discrete state). For S-Function blocks, the label contains whatever name is assigned to the state by the S-Function block.

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

Simulink can read back simulation data saved to the workspace in the `Structure with time` output format. See "Importing signal-and-time data structures" on page 10-18 for more information.

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.

Importing and Exporting 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 **Initial state** field of the **Load from workspace** area of the **Data Import/Export** pane.

You can also save the final states for the current simulation run and apply them to a subsequent simulation run. This feature can 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 **Data Import/Export** pane.

Saving Final States

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

Loading Initial States

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 a previous session, using the `Structure` or `Structure with time` format.

Model Reference Limitations On Loading Initial States. Simulink imposes the following limitations on loading the states of models that reference other models or that are referenced by other models.

- You cannot initialize the states of a referenced model from the workspace. Simulink ignores the **Initial state** setting for such models.
- You can use the array format to initialize the states of a top model only if the models that the top model references do not themselves have states.
- You can use the structure format to initialize the states of a top model but not those of the models that it references.

Limiting Output

Saving data to the workspace can slow down the simulation and consume memory. To avoid this, you can limit the number of samples saved to the most recent samples or you can skip samples by applying a decimation factor. To set a limit on the number of data samples saved, select the check box labeled **Limit data points to last** and specify the number of samples to save. To apply a decimation factor, enter a value in the field to the right of the **Decimation** label. For example, a value of 2 saves every other point generated.

Specifying Output Options

The **Output options** list on the **Data Import/Export** configuration pane ("Data Import/Export Pane" on page 10-39) enables you to control how much output the simulation generates. You can choose from three options:

- Refine output

- Produce additional output
- Produce specified output only

Refining Output

The `Refine output` choice provides additional output points when the simulation output is too coarse. This parameter provides an integer number of output points between time steps; for example, a refine factor of 2 provides output midway between the time steps, as well as at the steps. The default refine factor is 1.

To get smoother output, it is much faster to change the refine factor instead of reducing the step size. When the refine factor is changed, the solvers generate additional points by evaluating a continuous extension formula at those points. 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 you are using `ode45`. The `ode45` solver is capable of taking large steps; when graphing simulation output, you might find that output from this solver is not sufficiently smooth. If this is the case, run the simulation again with a larger refine factor. A value of 4 should provide much smoother results.

Note This option does not help the solver to locate zero crossings (see “Zero-Crossing Detection” on page 2-19).

Producing Additional Output

The `Produce additional output` choice enables you to specify directly those additional times at which the solver generates output. When you select this option, Simulink displays an **Output times** field on the **Data Import/Export** pane. Enter a MATLAB expression in this field that evaluates to an additional time or a vector of additional times. 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.

Producing 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 you are 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. For example,

```
[1:100]*0.01
```

is more accurate than

```
[1:0.01:100]
```

Configuration Sets

A configuration set is a named set of values for a model's parameters, such as solver type and simulation start or stop time. Every new model is created with a default configuration set, called Configuration, that initially specifies default values for the model's parameters. You can subsequently create and modify additional configuration sets and associate them with the model. The sets associated with a model can each specify different values for any given model parameter.

The Active Set

Only one of the configuration sets associated with a model is active at any given time. The active set determines the current values of the model's model parameters. Changing the value of a parameter in the Model Explorer changes its value in the active set. Simulink allows you to change the active set at any time (except when executing the model). In this way, you can quickly reconfigure a model for different purposes, e.g., testing and production, or apply standard configuration settings to new models.

To determine the configuration sets associated with a model, open the Model Explorer (see "The Model Explorer" on page 9-2). The configuration sets associated with the model appear as gear-shaped nodes in the Model Explorer's **Model Hierarchy** pane.

Activating a Configuration Set

To activate a configuration set, right-click the configuration set's node to display the node's context menu, then select **Activate** from the context menu.

Copying and Moving Configuration Sets

You can copy or move a configuration set by dragging its node and dropping it on any model node in the **Model Hierarch**y pane. To move a configuration set from one model to another, hold the **Ctrl** key and the left mouse button down and drag the configuration set's node to the node of the destination model; to copy a configuration, hold the **Ctrl** key and the right mouse button down and drag the configuration set's node to the node of the same or a different model.

Creating Configuration Sets

To create a new configuration set, copy an existing configuration set.

Setting Values in Configuration Sets

To set the value of a parameter in a configuration set, select the configuration set in the Model Explorer and then edit the value of the parameter on the corresponding dialog in the Model Explorer's dialog view.

Configuration Set API

Simulink provides an application program interface (API) that permits you to create and manipulate configuration sets from the command line or in a MAT-file or M-file. For example, to create a configuration set from scratch at the command line, enter

```
cfg_set = Simulink.ConfigSet('name')
```

where **name** is the name of the new configuration set. Use `get_param` and `set_param` to get and set the value of a parameter in a configuration set. For example, to specify the Simulink fixed-step discrete solver in the configuration set, execute

```
set_param(cfg_set, 'Solver', 'FixedStepDiscrete')
```

To save the configuration set in a MAT-file, execute

```
save mat_file cfg_set
```

where **mat_file** is the name of the MAT-file. To load the configuration set, execute

```
load mat_file
```

To prevent or allow a user to change the value of a parameter in a configuration set, execute

```
setPropEnabled(cfg_set, 'param’, [0 | 1])
```

where **param** is the name of the parameter. To attach a configuration set to a model, execute

```
attachConfigSet(model, cfg_set)
```

where model is the model name (in quotes) or object. To get a model's active configuration set, execute

```
cfg_set = getActiveConfigSet(model)
```

To get a configuration set's full name (e.g., `engine/Configuration`), execute

```
getFullName(cfg_set)
```

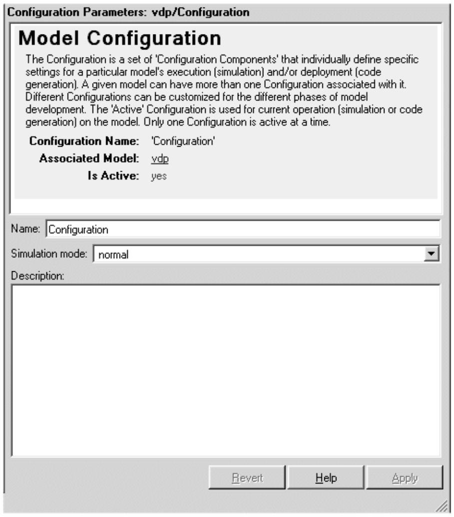
To set a model's active set, execute

```
setActiveConfigSet(model, cfg_set)
```

where **`cfg_set`** is the configuration set's name in quotes.

The Model Configuration Dialog Box

The Model Configuration dialog box appears when you select a model configuration in the Model Explorer.



Configuration Parameters: vdp/Configuration

Model Configuration

The Configuration is a set of 'Configuration Components' that individually define specific settings for a particular model's execution (simulation) and/or deployment (code generation). A given model can have more than one Configuration associated with it. Different Configurations can be customized for the different phases of model development. The 'Active' Configuration is used for current operation (simulation or code generation) on the model. Only one Configuration is active at a time.

Configuration Name: 'Configuration'
Associated Model: vdp
Is Active: yes

Name: Configuration

Simulation mode: normal

Description:

The dialog box has the following fields.

Name

Name of the configuration. You can change the name of the configuration by editing this field.

Simulation mode

The simulation mode used to simulate the model in this configuration. The options are `normal` ("Simulation Basics" on page 10-2), `accelerator` (see "The Simulink Accelerator" on page 14-2), or external mode (see the Real-Time Workshop documentation).

Description

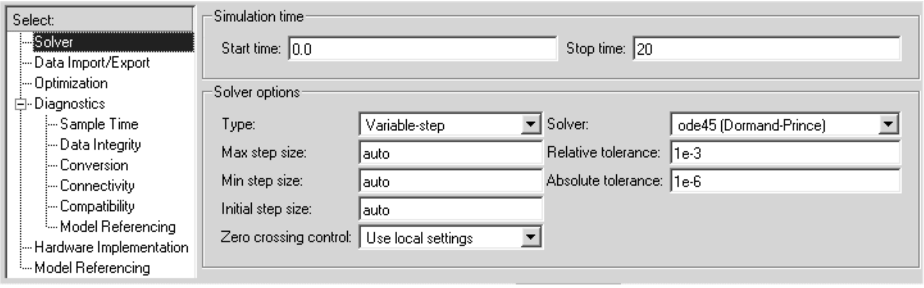
A description of this configuration. You can use this field to enter information pertinent to using this configuration.

The Configuration Parameters Dialog Box

The **Configuration Parameters** dialog box allows you to modify settings for a model's active configuration set (see "Configuration Sets" on page 10-26).

Note You can also use the Model Explorer to modify settings for the active configuration set as well as for any other configuration set. See "The Model Explorer" on page 9-2 for more information.

To display the dialog box, select **Configuration Parameters** from the model editor's **Simulation** or context menu. The dialog box appears.



Start time: 0.0
Stop time: 20
Type: Variable-step
Solver: ode45 (Dormand-Prince)
Max step size: auto
Relative tolerance: 1e-3
Min step size: auto
Absolute tolerance: 1e-6
Initial step size: auto
Zero crossing control: Use local settings

The dialog box groups the controls used to set the configuration parameters into various categories. To display the controls for a specific category, click the category in the **Select** tree on the left side of the dialog box. See the following sections for information on how to use the various categories of controls to set configuration parameters for the active configuration set.

- "The Solver Pane" on page 10-31
- "Data Import/Export Pane" on page 10-39
- "The Optimization Pane" on page 10-43
- "The Diagnostics Pane" on page 10-48
- "Hardware Implementation Pane" on page 10-63
- "Model Referencing Pane" on page 10-67

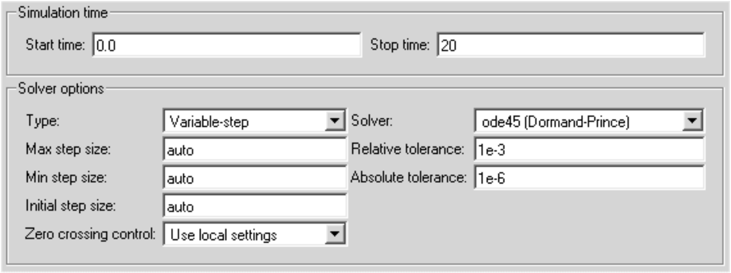
In most cases, Simulink does not immediately apply a change that you have made with a control. To apply a change, you must click either the **OK** or the

Apply button at the bottom of the dialog box. The **OK** button applies all the changes you made and dismisses the dialog box. The **Apply** button applies the changes but leaves the dialog box open so that you can continue to make changes.

Note Each of the controls on the **Configuration Parameters** dialog box correspond to a configuration parameter that you can set via the `sim` and `simset` commands. The “Model Parameters” subsection of the “Model and Block Parameters” section of the Simulink Reference documenation lists these parameters. This section also specifies for each configuration parameter the **Configuration Parameters** dialog box prompt of the control that sets it. This allows you to determine the model parameter corresponding to a control on the **Configuration Parameters** dialog box.

The Solver Pane

The **Solver** configuration parameters pane allows you to specify a simulation start and stop time and select and configure a solver for a particular simulation configuration.



Simulation time

Start time: 0.0

Stop time: 20

Solver options

Type: Variable-step

Solver: ode45 (Dormand-Prince)

Max step size: auto

Relative tolerance: 1e-3

Min step size: auto

Absolute tolerance: 1e-6

Initial step size: auto

Zero crossing control: Use local settings

The **Solver** pane contains the following control groups.

Simulation time

This control group enables you to specify the simulation start and stop time. It contains the following controls.

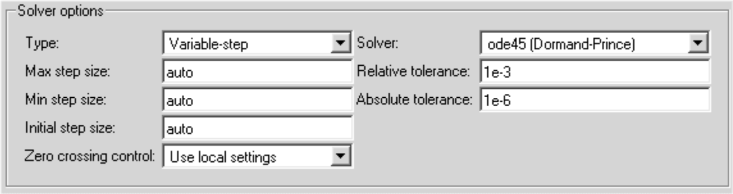
Start time. Specifies the simulation start time. The default start time is 0.0 seconds.

Stop time. Specifies the simulation stop time. The default stop time is 10.0 seconds. Specify `inf` to cause the simulation to run until you pause or stop it.

Simulation time and actual clock time are not the same. For example, running a simulation for 10 seconds usually does 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 speed.

Solver Options

The **Solver options** controls group allows you to specify the type of solver to be used and simulation options specific to that solver.



Type: Variable-step
Max step size: auto
Min step size: auto
Initial step size: auto
Zero crossing control: Use local settings
Solver: ode45 (Dormand-Prince)
Relative tolerance: 1e-3
Absolute tolerance: 1e-6

The contents of the group depends on the solver type.

General Solver Options

The follow options always appear.

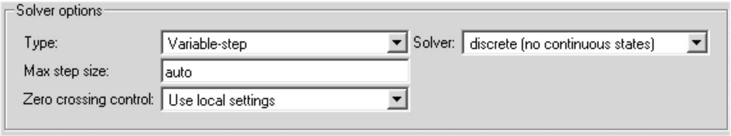
Type. Specifies the type of solver to be used to solve the currently selected model, either `Fixed-step` or `Variable-step`. See "Choosing a Solver Type" on page 10-7 and "Improving Simulation Performance and Accuracy" on page 10-76 for information on how to choose the solver type that best suits your application.

Solver. Specifies the solver used to simulate this configuration of the current model. The associated pull-down list displays available solvers of the type specified by the **Type** control. To specify another solver of the specified type, select the solver from the pull-down list. See "Choosing a Fixed-Step Solver" on page 10-8 and "Choosing a Variable-Step Solver" on page 10-12 for information on how to choose the solvers listed in the **Solver** list.

The other controls that appear in this group depend on the type of solver you have selected.

Variable-Step Discrete Solver Options

The following options appear when you select the Simulink variable-step discrete solver.



Solver options

Type: Variable-step
Solver: discrete (no continuous states)
Max step size: auto
Zero crossing control: Use local settings

Max step size. Appears only if the solver **Type** is `Variable-step`. Specifies the largest time step the selected variable-step solver can take. The default auto causes Simulink to choose the model's shortest sample time as the maximum step size.

Zero crossing control. 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 2-19 for more information.

You can override this optimization on a block-by-block basis for the following types of blocks:

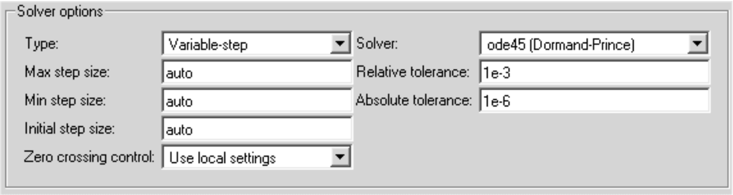
Abs
Backlash
Dead Zone
Enable
Hit Crossing
If
Integrator
MinMax
Relay
Relational Operator
Saturation
Sign
Step
Switch
Switch Case
Trigger

To override zero-crossing detection for an instance of one of these blocks, open the block's parameter dialog box and uncheck the **Enable zero crossing detection** option. You can enable or disable zero-crossing selectively for these blocks only if you have selected the `Use local settings` setting of the **Zero**

crossing control control on the **Solver** pane of the **Configuration Parameters** dialog box.

Variable-Step Continuous Solver Options

The following options appear when you select any of the Simulink variable-step continuous solvers.



Solver options

Type: Variable-step

Solver: ode45 (Dormand-Prince)

Max step size: auto

Relative tolerance: 1e-3

Min step size: auto

Absolute tolerance: 1e-6

Initial step size: auto

Zero crossing control: Use local settings

Max step size. Specifies the largest time step the solver can take. The default is determined from the start and stop times. If the stop time equals the start time or is `inf`, Simulink chooses `0.2` sec. as the maximum step size. Otherwise, it sets the maximum step size to

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

Generally, the default maximum step size is sufficient. If you are concerned about the solver's 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 might 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 “Output options” on page 10-42.

Initial step size. By default, the solver selects 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 might 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.

Min step size. This option appears only for variable-step continuous solvers. Specifies the smallest time step the selected variable-step 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** pane. 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.

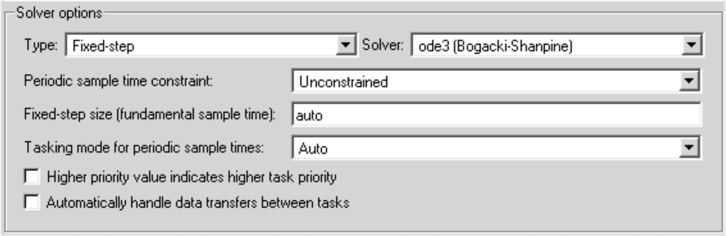
Relative tolerance. Relative tolerance for this solver (see “Specifying Variable-Step Solver Error Tolerances” on page 10-14).

Absolute tolerance. Absolute tolerance for this solver (see “Specifying Variable-Step Solver Error Tolerances” on page 10-14).

Maximum order. This option appears only if you select the `ode15s` solver, which is based on NDF formulas of orders 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). As an alternative, you can try using the `ode23s` solver, which is a lower order (and A-stable) solver.

Fixed-Step Solver Options

The following options appear when you choose one of the Simulink fixed-step solvers.



Solver options

Type: Fixed-step Solver: ode3 (Bogacki-Shanpine)

Periodic sample time constraint: Unconstrained

Fixed-step size (fundamental sample time): auto

Tasking mode for periodic sample times: Auto

□ Higher priority value indicates higher task priority

□ Automatically handle data transfers between tasks

Periodic sample time constraint. Allows you to specify constraints on the sample times defined by this model. During simulation, Simulink checks to ensure that the model satisifies the constraints. If the model does not satisfy the specified constraint, Simulink displays an error message. The contents of the **Solver options** group changes depending on the options selected. The options are

- `Unconstrained`

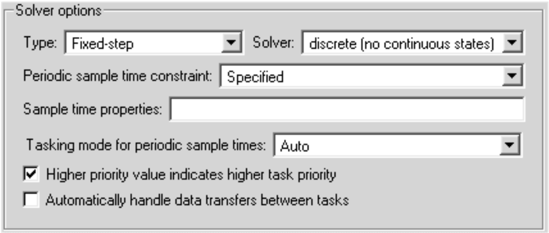
  No constraints. Selecting this option causes Simulink to display a field for entering the solver step size.

  See "Fixed step size (fundamental sample time)" on page 10-36 for a description of this field.
- `Ensure sample time independent`

  Check to ensure that this model can inherit its sample times from a model that references it without altering its behavior. Models that specify a step size (i.e., a base sample time) cannot satisfy this constraint. For this reason, selecting this option causes Simulink to hide the group's step size field (see "Fixed step size (fundamental sample time)" on page 10-36.
- `Specified`

  Check to ensure that this model operates at a specified set of prioritized periodic sample times.

  Selecting this option causes Simulink to display additional controls for specifying prioritized sample times and sample time priority options.



Solver options

Type: Fixed-step ▼ Solver: discrete (no continuous states) ▼

Periodic sample time constraint: Specified ▼

Sample time properties:

Tasking mode for periodic sample times: Auto ▼

☑ Higher priority value indicates higher task priority

□ Automatically handle data transfers between tasks

  See below for a description of these additional controls.

Fixed step size (fundamental sample time). Specifies the step size used by the selected fixed-step solver. Entering `auto` (the default) in this field causes Simulink to choose the step size. If the model specifies one or more periodic sample times, Simulink chooses a step size equal to the least common denominator of the specified sample times. This step size, known as the fundamental sample time of the model, ensures that the solver will take a step at every sample time defined by the model. If the model does not define any periodic sample times, Simulink chooses a step size that divides the total simulation time into 50 equal steps.

Sample time properties. Specifies and assigns priorities to the sample times that this model implements. Enter an `Nx3` matrix in this field whose rows specify the sample times specified by this model in order from fastest rate to slowest rate.

Note If the model's fundamental rate differs from the fastest rate specified by the model (see "Determining Step Size for Discrete Systems" on page 2-36), you should specify the fundamental rate as the first entry in the matrix followed by the specified rates in order from fastest to slowest.

The row for each sample time should have the form

```
[period, offset, priority]
```

where `period` is the sample time's period of a sample time, `offset` is the sample time's offset, and `priority` is the execution priority of the real-time task associated with the sample rate, with faster rates receiving higher priorities. For example, the following entry

```
[[0.1, 0, 10]; [0.2, 0, 11]; [0.3, 0, 12]]
```

declares that this model should specify two sample rates, whose fundamental sample time is `0.1` second, and assigns priorities of `10`, `11`, and `12` to the sample times. This example assumes that for this model, higher priority values indicate lower priorities, i.e., the **Higher priority value indicates higher task priority** option is not selected (see "Higher priority value indicates higher task priority" on page 10-39).

Note If your model operates at only one rate, you can enter the rate as a three-element vector in this field, e.g., `[0.1, 0, 10]`.

When updating a model, Simulink checks the sample times defined by the model against this field. If the model defines more or fewer sample times than this field specifies, Simulink displays an error message.

Note If you select `Unconstrained` as the **Periodic sample time constraint**, Simulink assigns a priority of `40` to the model's base sample rate. If the **Higher priority value indicates higher task priority** option is selected (see "Higher priority value indicates higher task priority" on page 10-39), Simulink assigns priorities `39`, `38`, etc., to subrates of the base rate; otherwise, it assigns priorities `41`, `42`, `43`, etc., to the subrates. Continuous rate is assigned a higher priority than is the discrete base rate no matter whether you select `Specified` or `Unconstrained` as the **Periodic sample time constraint**.

Tasking mode for periodic sample times. Specifies one of the following options:

- `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 the Rate Transition block to eliminate illegal rate transitions from your model. For more information, see "Models with Multiple Sample Rates" in the Real-Time Workshop documentation for more information.

- `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.

Higher priority value indicates higher task priority. If checked, this option indicates that the real-time system targeted by this model assigns a higher priority to tasks with higher priority values. This in turn causes Simulink Rate Transition blocks to treat asynchronous transitions between rates with lower priority values to rates with higher priority values as low-to-high rate transitions. If unchecked (the default), this option indicates that the real-time system targeted by this model assigns a higher priority to tasks with lower priority values. This in turn causes Simulink Rate Transition blocks to treat asynchronous transitions between rates with lower priority values to rates with higher priority values as high-to-low rate transitions. See the Real-Time Workshop documentation for more information on this option.

Automatically handle data transfers between tasks. If checked, this option causes Simulink to insert hidden Rate Transition blocks where rate transitions occur between blocks.

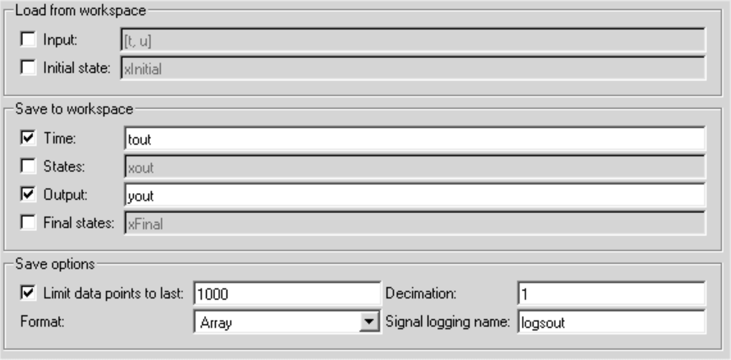
The next two options appear only if you select the ode14x solver (see “Implicit Fixed-Step Continuous Solvers” on page 10-11).

Extrapolation Order. Extrapolation order used by the `ode14x` solver to compute a model’s states at the next time step from the states at the current time step. The higher the order, the more accurate but the more computationally intensive is the solution per step size.

Number Newton’s iterations. Number of Newton’s method iterations used by the ode14x solver to compute a model’s states at the next time step from the states at the current time step. The more iterations, the more accurate but the more computationally intensive is the solution per step size.

Data Import/Export Pane

The **Data Import/Export** pane allows you to import and export data to the MATLAB workspace. To display the pane, select **Data Import/Export** from the **Select** tree of the **Configuration Parameters** dialog box or select a configuration set (see “Configuration Sets” on page 10-26) in the Model Explorer and display the configuration’s **Data Import/Export** subset.



Load from workspace

□ Input: [t, u]

□ Initial state: xInitial

Save to workspace

☑ Time: tout

□ States: xout

☑ Output: yout

□ Final states: xFinal

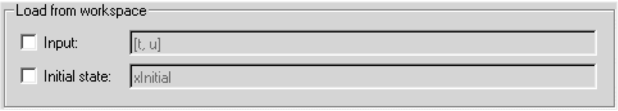
Save options

☑ Limit data points to last: 1000 Decimation: 1

Format: Array Signal logging name: logsout

Load from workspace

This group contains controls that enable you to specify options for importing data from the MATLAB workspace.



Load from workspace

□ Input: [t, u]

□ Initial state: xInitial

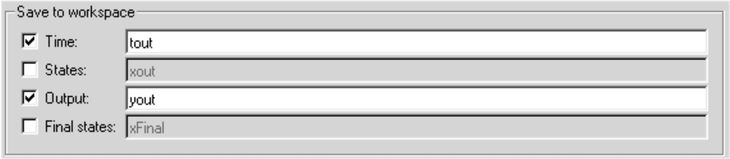
It includes the following controls.

Input. A MATLAB expression that specifies the data to be imported from the MATLAB workspace. See “Importing Input Data from the MATLAB Workspace” on page 10-16 for information on how to use this field.

Initial state. A MATLAB expression that specifies the initial values of a model’s states. See “Importing and Exporting States” on page 10-22 for more information.

Save to workspace

This group contains controls that enable you to specify options for exporting data to the MATLAB workspace.



Save to workspace

☑ Time: tout

□ States: xout

☑ Output: yout

□ Final states: xFinal

It includes the following controls.

Time. Name of the MATLAB variable to be used to store simulation time data to be exported during simulation.

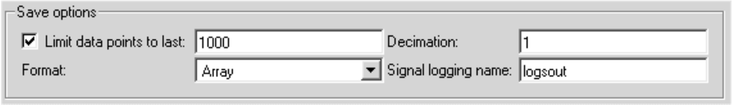
States. Specifies the name of a MATLAB variable to be used to store state data exported during a simulation. See “Importing and Exporting States” on page 10-22 for more information.

Ouput. Name of the MATLAB variable to be used to store signal data exported during this simulation. See “Exporting Output Data to the MATLAB Workspace” on page 10-20 for more information.

Final states. Specifies the name of a MATLAB variable to be used to store the values of this model’s states at the end of a simulation. See “Importing and Exporting States” on page 10-22 for more information.

Save options

This group contains controls that allow you to specify options for saving (and reloading) data from the MATLAB workspace.



Save options

☑ Limit data points to last: 1000

Decimation: 1

Format: Array

Signal logging name: logsout

It includes the following controls.

Limit data points to last. Limits the number of data points exported to the MATLAB workspace to `N`, the number specified in the adjacent edit field. At the end of the simulation, the MATLAB workspace contains the last `N` points generated by the simulation.

Decimation. If specified, Simulink outputs only every `N` points, where `N` is the specified decimation factor.

Format. Specifies the format of state and output data saved to or loaded from the MATLAB workspace. The options are

- `Array`

  The format of the data is a matrix each row of which corresponds to a simulation time step.
- `Structure with time`

  The format of the data is a structure that has two fields: a time field and a signals field. The time field contains a vector of simulation times. The signals field contains a substructure for each model input port (for imported data) or output port (for exported data). Each port substructure contains signal data for the corresponding port.
- `Structure`

  The format of the data is a structure that contains substructures for each port. Each port substructure contains signal data for the corresponding port.

See “Importing and Exporting Simulation Data” on page 10-16 for more information on these formats.

Signal logging name. Variable name used to store signals logged during a simulation (see “Logging Signals” on page 6-27).

Output options. Options for generating additional output signal data.

Note These options appear only if the model specifies a variable-step solver (see “The Solver Pane” on page 10-31).

The options are

- `Refine output`

  Output data between as well as at simulation times steps. Selecting this option causes the **Refine factor** edit field to appear below this control (see

"Refine factor" on page 10-43). Use this field to specify the number of points to generate between simulation time steps. For more information, see "Refining Output" on page 10-24.

- `Produce additional output`

  Produce additional output at specified times. Selecting this option causes the **Output times** field to appear. Use this field to specify the simulation times at which Simulink should generate additional output.

- `Produce specified output`

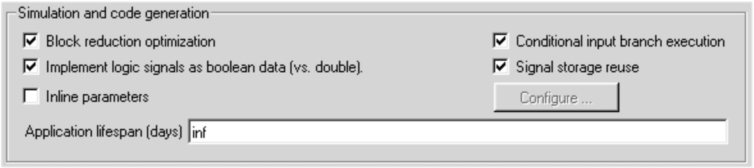
  Produce output only at specified times. Selecting this option causes the **Output times** field to appear. Use this field to specify the simulation times at which Simulink should generate additional output.

Refine factor. This field appears when you select `Refine output` as the value of **Output options**. It specifies how many points to generate 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. For more information, see "Refining Output" on page 10-24.

Output times. This field appears when you select `Produce additional output` or `Produce specified output` as the value of **Output options**. Use this field to specify times at which Simulink should generate output in addition to or instead of at the simulation steps taken by the solver used to simulate the model.

The Optimization Pane

The **Optimization** pane allows you to select various options that improve simulation performance and the performance of code generated from this model.



Simulation and code generation

☑ Block reduction optimization

☑ Implement logic signals as boolean data (vs. double).

□ Inline parameters

☑ Conditional input branch execution

☑ Signal storage reuse

Application lifespan (days) inf

The pane contains the following controls.

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

Conditional input branch execution. This optimization applies to models containing Switch and Multiport Switch blocks. When enabled, this optimization executes only the blocks required to compute the control input and the data input selected by the control input at each time step for each Switch or Multiport Switch block in the model. Similarly, code generated from the model by Real-Time Workshop executes only the code needed to compute the control input and the selected data input. This optimization speeds simulation and execution of code generated from the model.

At the beginning of the simulation or code generation, Simulink examines each signal path feeding a switch block data input to determine the portion of the path that can be optimized. The optimizable portion of the path is that part of the signal path that stretches from the corresponding data input back to the first block that is a nonvirtual subsystem, has continuous or discrete states, or detects zero crossings.

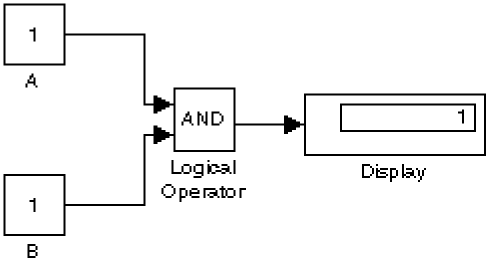
Simulink encloses the optimizable portion of the signal path in an invisible atomic subsystem. During simulation, if a switch data input is not selected, Simulink executes only the nonoptimizable portion of the signal path feeding the input. If the data input is selected, Simulink executes both the nonoptimizable and the optimizable portion of the input signal path.

Inline parameters. By default you can modify (“tune”) many block parameters during simulation (see “Tunable Parameters” on page 2-8). Selecting this option makes all parameters nontunable by default. Making parameters nontunable allows Simulink to move blocks whose outputs depend only on block parameter values outside the simulation loop, thereby speeding up simulation of the model and execution of code generated from the model. When this option is selected, Simulink disables the parameter controls of the block dialog boxes for the blocks in your model to prevent you from accidentally modifying the block parameters.

Simulink allows you to override the **Inline parameters** option for parameters whose values are defined by variables in the MATLAB workspace. To specify that such a parameter remain tunable, specify the parameter as global in the **Model Parameter Configuration** dialog box (see “Model Parameter Configuration Dialog Box” on page 10-47). To display the dialog box, click the adjacent **Configure** button. To tune a global parameter, change the value of the corresponding workspace variable and choose **Update Diagram** (**Ctrl+D)** from the Simulink **Edit** menu.

Note You cannot tune inlined parameters in code generated from a model. However, when simulating a model, you can tune an inlined parameter if its value derives from a workspace variable. For example, suppose that a model has a Gain block whose **Gain** parameter is inlined and equals `a`, where `a` is a variable defined in the model's workspace. When simulating the model, Simulink disables the **Gain** parameter field, thereby preventing you from using the block's dialog box to change the gain. However, you can still tune the gain by changing the value of `a` at the MATLAB command line and updating the diagram.

Implement logic signals as boolean data (vs. double). Causes blocks that accept Boolean signals to require Boolean signals. If this option is off, blocks that accept inputs of type `boolean` also accept inputs of type `double`. For example, consider the following model.



This model connects signals of type `double` to a Logical Operator block, which accepts inputs of type `boolean`. If the `Boolean logic signals` option is on, this model generates an error when executed. If the `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 `double`.

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

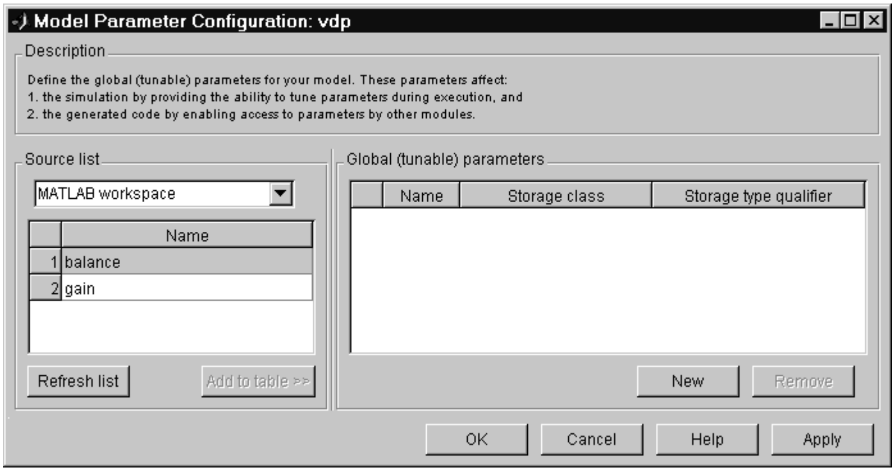
- Debug a C-MEX S-function
- Use a Floating Scope or a Display block with the **Floating display** option selected to inspect signals in a model that you are debugging

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

Application lifespan (days). Specifies the lifespan in days of the system represented by this model. This value and the simulation step size determine the data type used by fixed-point blocks to store absolute time values.

Model Parameter Configuration Dialog Box

The **Model Parameter Configuration** dialog box allows you to override the **Inline parameters** option (see “Inline parameters” on page 10-44) for selected parameters.



Define the global (tunable) parameters for your model. These parameters affect:
1. the simulation by providing the ability to tune parameters during execution, and
2. the generated code by enabling access to parameters by other modules.

	Name
1	balance
2	gain

	Name	Storage class	Storage type qualifier

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 this button if you have added a variable to the workspace since the last time the list was displayed.

Add to table. Adds the variables 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.

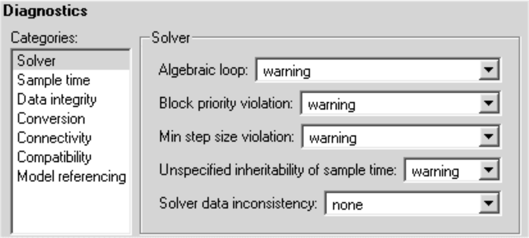
Note This option does not create the corresponding variable in the MATLAB workspace. You must create the variable yourself.

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.

The Diagnostics Pane

The **Diagnostics** configuration parameters pane enables you to specify what diagnostic action Simulink should take, if any, when it detects an abnormal condition during compilation or simulation of a model.



Algebraic loop: warning

Block priority violation: warning

Min step size violation: warning

Unspecified inheritability of sample time: warning

Solver data inconsistency: none

The options are typically to do nothing or to display a warning or an error message (see “Diagnosing Simulation Errors” on page 10-72). A warning message does not terminate a simulation, but an error message does.

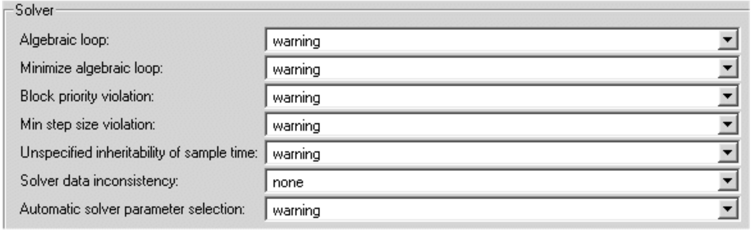
The pane displays groups of controls corresponding to various categories of abnormal conditions that can occur during a solution. To display controls for a specific category, left-click the category in the **Categories** list on the left side of the **Diagnostics** pane. To display controls for additional categories, left-click the categories while pressing the **Ctrl** key on your keyboard. See the following sections for information on using the controls on the **Diagnostics** pane:

- “Solver Diagnostics” on page 10-49
- “Sample Time Diagnostics” on page 10-51
- “Data Integrity Diagnostics” on page 10-52

- “Conversion Diagnostics” on page 10-54
- “Connectivity Diagnostics” on page 10-55
- “Compatibility Diagnostics” on page 10-56
- “Model Reference Diagnostics” on page 10-60

Solver Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects a solver-related error.



Solver

Algebraic loop: warning

Minimize algebraic loop: warning

Block priority violation: warning

Min step size violation: warning

Unspecified inheritability of sample time: warning

Solver data inconsistency: none

Automatic solver parameter selection: warning

Algebraic loop. Simulink detected an algebraic loop while compiling the model. See “Algebraic Loops” on page 2-24 for more information. If you set this option to `Error`, Simulink displays an error message and highlights the portion of the block diagram that comprises the loop (see “Highlighting Algebraic Loops” on page 2-26).

Minimize algebraic loop. Specifies diagnostic action to take if you have requested that Simulink attempt to remove algebraic loops involving a specified subsystem (see “Eliminating Algebraic Loops” on page 2-27) and an input port of that subsystem has direct feedthrough. If the port is involved in an algebraic loop, Simulink can remove the loop only if at least one other input port in the loop lacks direct feedthrough.

Block priority violation. Simulink detected a block priority specification error while compiling the model.

Min step size violation. The next simulation step is smaller than the 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 “Min step size” on page 10-35 and “Maximum order” on page 10-35 for more information.

Unspecified inheritability of sample time. Specifies diagnostic action to be taken if this model contains S-functions that do not specify whether they preclude this model from inheriting their sample times from a parent model. Simulink checks for this condition only if the solver used to simulate this model is a fixed-step discrete solver and the periodic sample time constraint for the solver is set to ensure sample time independence (see "Periodic sample time constraint" on page 10-36).

Solver data inconsistency. Consistency checking is a debugging tool that validates certain assumptions made by Simulink 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 `none`. 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 matrix, the block's output functions can be called many times at the same value of *t*.

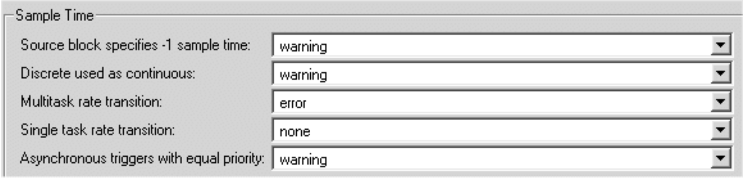
When consistency checking is enabled, Simulink recomputes the appropriate values and compares them to the cached values. If the values are not the same, a consistency error occurs. Simulink compares computed values for these quantities:

- Outputs
- Zero crossings
- Derivatives
- States

Automatic solver parameter selection. Specifies diagnostic action to take if Simulink changes a solver parameter setting. For example, suppose that you simulate a discrete model that specifies a continuous solver and warning as the setting for this diagnosic. In this case, Simulink changes the solver type to discrete and displays a warning message about this change at the MATLAB command line.

Sample Time Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects a compilation error related to model sample times.



Sample Time

Source block specifies -1 sample time: warning

Discrete used as continuous: warning

Multitask rate transition: error

Single task rate transition: none

Asynchronous triggers with equal priority: warning

Source block specifies -1 sample time. A source block (e.g., a Sine Wave block) specifies a sample time of -1.

Discrete used as continuous. The Unit Delay block, which is a discrete block, inherits a continuous sample time from the block connected to its input.

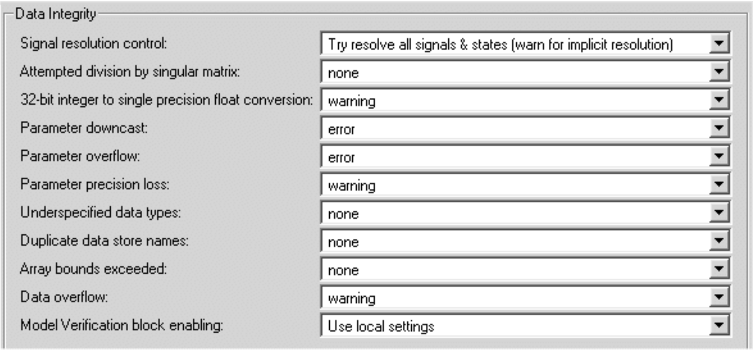
Multitask rate transition. An invalid rate transition occurred between two blocks operating in multitasking mode (see "Tasking mode for periodic sample times" on page 10-38).

Single task rate transition. A rate transition occurred between two blocks operating in single-tasking mode (see "Tasking mode for periodic sample times" on page 10-38).

Asynchronous triggers with equal priority. One asynchronous task of the target represented by this model has the same priority as another of the target's asynchronous tasks. This option must be set to `Error` if the target allows tasks having the same priority to preempt each other.

Data Integrity Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects a condition that could compromise the integrity of data defined by the model.



Data Integrity

Signal resolution control:	Try resolve all signals & states (warn for implicit resolution)
Attempted division by singular matrix:	none
32-bit integer to single precision float conversion:	warning
Parameter downcast:	error
Parameter overflow:	error
Parameter precision loss:	warning
Underspecified data types:	none
Duplicate data store names:	none
Array bounds exceeded:	none
Data overflow:	warning
Model Verification block enabling:	Use local settings

Signal resolution control. Specifies how Simulink resolves signals to `Simulink.Signal` objects in the MATLAB workspace. The options are

- `Try resolve all signals & states (warn for implicit resolution)`

  Try to resolve every signal or discrete state that has a name to a `Simulink.Signal` object having the same name. Display a warning message if a signal or state resolves implicitly to a signal object, i.e., a signal object with the same name as the signal or state exists in the MATLAB workspace but the model does not specify that the signal or state should resolve to a signal object.
- `Try resolve all signals & states`

  Try to resolve every signal or discrete state that has a name to a `Simulink.Signal` object having the same name regardless of whether the model specifies that the signal or state should resolve to a signal object.
- `Use local settings`

  Try to resolve every signal or discrete state that the model specifies should resolve to a `Simulink.Signal` object in the MATLAB workspace.

Note Use the Signal Properties dialog box (see "Signal Properties Dialog Box" on page 6-30) to specify explicit resolution for signals. Use the **State Properties** dialog boxes of blocks that have discrete states, e.g., the Discrete-Time Integrator block, to specify explicit resolution for discrete states.

Attempted division by singular matrix. The Product block detected a singular matrix while inverting one of its inputs in matrix multiplication mode.

32-bit integer to single precision 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 7-2 for more information.

Parameter downcast. Computation of the output of the block required converting the parameter's specified type to a type having a smaller range of values (e.g., from `uint32` to `uint8`). This diagnostic applies only to named tunable parameters.

Parameter overflow. The data type of the parameter could not accommodate the parameter's value.

Parameter precision loss. Computation of the output of the block required converting the specified data type of the parameter to a less precise data type (e.g., from `double` to `uint8`).

Underspecified data types. Simulink could not infer the data type of a signal during data type propagation.

Duplicate data store names. The model contains multiple Data Store Memory blocks that specify the same data store name.

Array bounds exceeded. 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.

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 7-2 for more information.

Model Verification block enabling. This parameter allows you to enable or disable model verification blocks in the current model either globally or locally. Select one of the following options:

- `Use local settings`

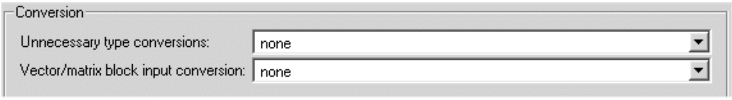
  Enables or disables blocks based on the value of the **Enable assertion** parameter of each block. If a block's **Enable assertion** parameter is on, the block is enabled; otherwise, the block is disabled.
- `Enable all`

  Enables all model verification blocks in the model regardless of the settings of their **Enable assertion** parameters.
- `Disable all`

  Disables all model verification blocks in the model regardless of the settings of their **Enable assertion** parameters.

Conversion Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects a data type conversion problem while compiling the model.



Conversion

Unnecessary type conversions: none

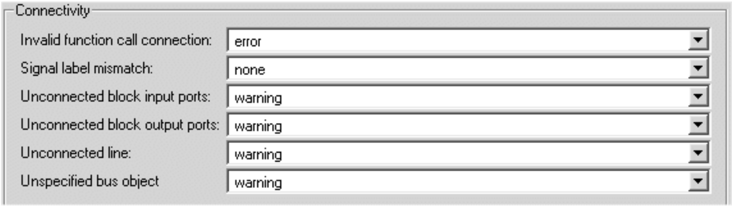
Vector/matrix block input conversion: none

Unnecessary type conversions. A Data Type Conversion block is used where no type conversion is necessary.

Vector/matrix block input conversion. A vector-to-matrix or matrix-to-vector conversion occurred at a block input (see "Vector or Matrix Input Conversion Rules" on page 6-14).

Connectivity Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects a problem with block connections while compiling the model.



Connectivity

Invalid function call connection: error

Signal label mismatch: none

Unconnected block input ports: warning

Unconnected block output ports: warning

Unconnected line: warning

Unspecified bus object warning

Invalid function call connection. Simulink has detected an incorrect use of a function-call subsystem in your model (see the “Function-call systems” examples in the Simulink “Subsystem Semantics” library for examples of invalid uses of function-call subsystems. Disabling this error message can lead to invalid simulation results.

Signal label mismatch. The simulation encountered virtual signals that have a common source signal but different labels (see “Virtual Signals” on page 6-4).

Unconnected block input ports. Model contains a block with an unconnected input.

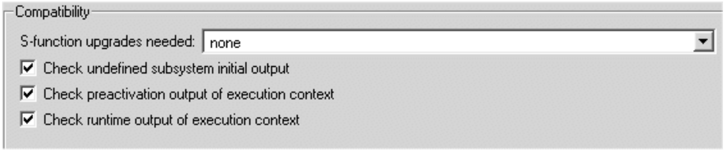
Unconnected block output ports. Model contains a block with an unconnected output.

Unconnected line. Model contains an unconnected line.

Unspecified bus object. Specifies diagnostic action to take while generating a simulation target for a referencced model if any of the model’s root Outport blocks is connected to a bus but does not specify a bus object (see `Simulink.Bus`).

Compatibility Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects an incompatibility between this version of Simulink and the model when updating or simulating the model.



Compatibility

S-function upgrades needed: none

☑ Check undefined subsystem initial output

☑ Check preactivation output of execution context

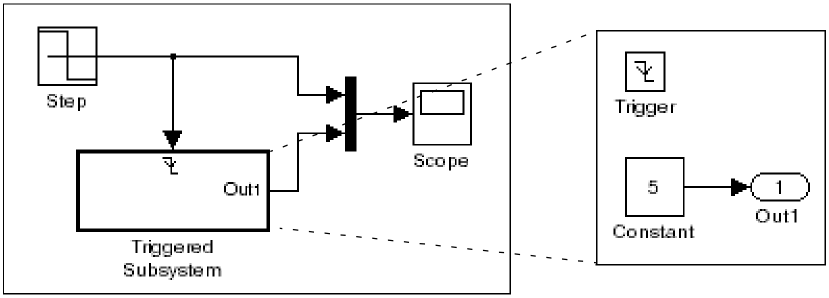
☑ Check runtime output of execution context

S-function upgrade needed. A block was encountered that has not been upgraded to use features of the current release.

Check undefined subsystem initial output. Display a warning if the model contains a conditionally executed subystem in which a block with a specified initial condition (e.g., a Constant, Initial Condition, or Delay block) drives an Outport block with an undefined initial condition, i.e., the Outport block's **Initial output** parameter is set to [].

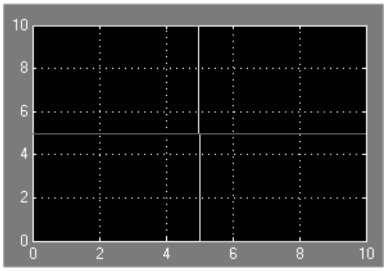
Models with such subystems can produce initial results (i.e., before initial activation of the conditionally executed subsystem) in the current release that differ from initial results produced in Release 13 or earlier releases.

Consider for example the following model.

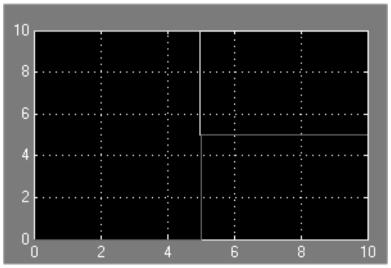


This model does not define the initial condition of the triggered subsystem's output port.

The following figure compares the superimposed output of this model's Step block and the triggered subsystem in Release 13 and the current release.



Release 13



Current Release

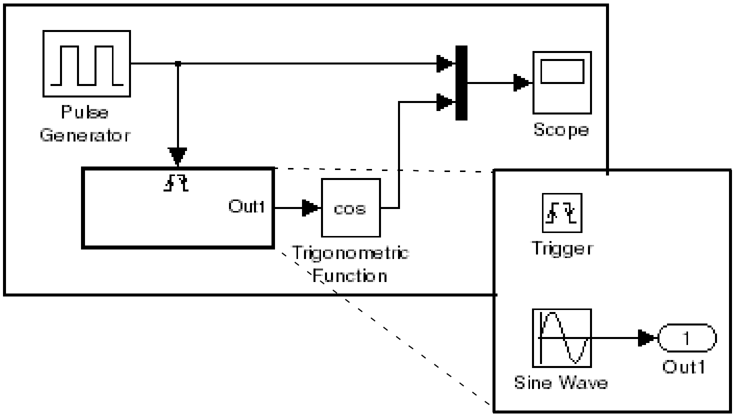
Notice that the initial output of the triggered subsystem differs between the two releases. This is because Release 13 and earlier releases use the initial output of the block connected to the output port (i.e., the Constant block) as the triggered subsystem's initial output. By contrast, this release outputs `0` as the initial output of the triggered subsystem because the model does not specify the port's initial output.

Check preactivation output of execution context. Display a warning if the model contains a block that meets the following conditions:

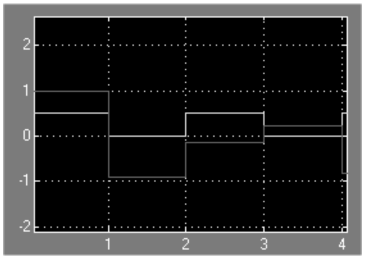
- The block produces nonzero output for zero input (e.g., a Cosine block).
- The block is connected to an output of a conditionally executed subsystem.
- The block inherits its execution context from that subsystem.
- The Outport to which it is connected has an undefined initial condition, i.e., the Outport block's **Initial output** parameter is set to [].

Models with blocks that meet these criteria can produce initial results (i.e., before the conditionally executed subsystem is first activated in the current release that differ from initial results produced in Release 13 or earlier releases.

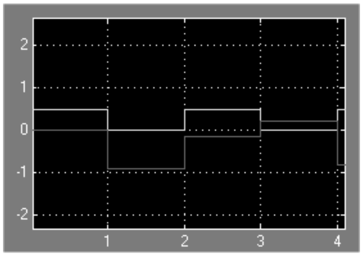
Consider for example the following model.



The following figure compares the superimposed output of the Pulse Generator and cos block in Release 13 and the current release.



Release 13



Current Release

Note that the initial output of the cos block differs between the two releases. This is because in Release 13, the cos block belongs to the execution context of the root system and hence executes at every time step whereas in the current release, the cos block belongs to the execution context of the triggered subsystem and hence executes only when the triggered subsystem executes.

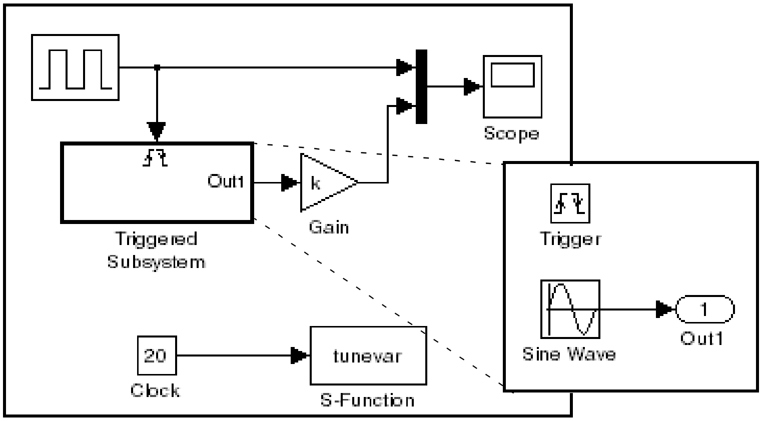
Check runtime output of execution context. Display a warning if the model contains a block that meets the following conditions:

- The block has a tunable parameter.

- The block is connected to an output of a conditionally executed subsystem.
- The block inherits its execution context from that subsystem.
- The Outport to which it is connected has an undefined initial condition, i.e., the Outport block's **Initial output** parameter is set to [].

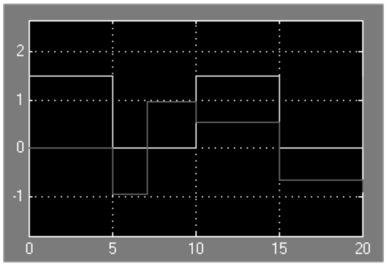
Models with blocks that meet these criteria can produce results when the parameter is tuned in the current release that differ from results produced in Release 13 or earlier releases.

Consider for example the following model.

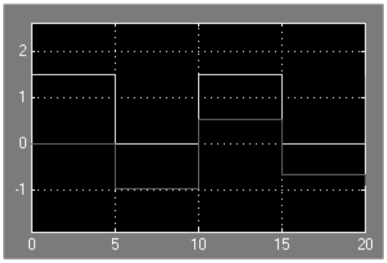


In this model, the `tunevar` S-function changes the value of the Gain block's `k` parameter and updates the diagram at simulation time 7 (i.e., it simulates interactively tuning the parameter).

The following figure compares the superimposed output of the model's Pulse Generator block and its Gain block in Release 13 and the current release.



Release 13

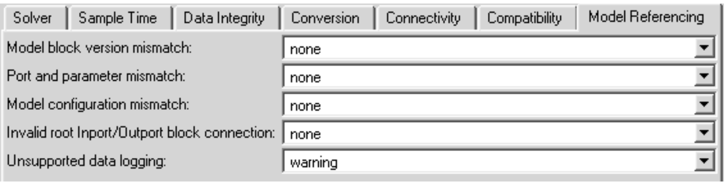


Current Release

Note that the output of the Gain block changes at time 7 in Release 13 but does not change in the current release. This is because in Release 13, the Gain block belongs to the execution context of the root system and hence executes at every time step whereas in the current release, the Gain block belongs to the execution context of the triggered subsystem and hence executes only when the triggered subsystem executes, i.e., at times 5, 10, 15, and 20.

Model Reference Diagnostics

This control group enables you to specify the diagnostic action that Simulink should take when it detects in incompatibility between this version of Simulink and the model while when updating or simulating the model.



Model block version mismatch: none

Port and parameter mismatch: none

Model configuration mismatch: none

Invalid root Inport/Outport block connection: none

Unsupported data logging: warning

Model block version mismatch. Specifies the diagnostic action to take during loading or updating of this model when Simulink detects a mismatch between the version of the model used to create or refresh a Model block in this model and the referenced model's current version. The options are

- none (the default)

- `warning`

  Refresh the Model block and report a warning message.
- `error`

  Display an error message but do not refresh the Model block.

If you have enabled displaying of referenced model version numbers on Model blocks for this model (see "Displaying Referenced Model Version Numbers" on page 4-65), Simulink displays a version mismatch on the Model block icon as, for example: `Rev:1.0 != 1.2`.

Port and parameter mismatch. Specifies the diagnostic action to take during model loading or updating when Simulink detects a mismatch between the I/O ports of a Model block in this model and the root-level I/O ports of the model it references or between the parameter arguments recognized by the Model block and the parameter arguments declared by the referenced model. The options are

- `none` (the default)
- `warning`

  Refresh the out-of-date Model block and report a warning message.
- `error`

  Display an error message but do not refresh the out-of-date Model block.

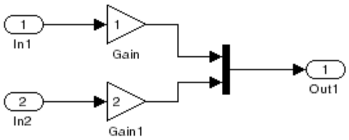
Model block icons can display a message indicating port or parameter mismatches. To enable this feature, select **Block display -> Model block I/O mismatch** from the parent model's **Format** menu.

Model configuration mismatch. Specifies the diagnostic action to take if the configuration parameters of a model referenced by this model do not match this model's configuration parameters or are inappropriate for a referenced model. The default action is `none`. Set this diagnostic to `warning` or `error` if you suspect that an inappropriate or mismatched configuration parameter may be causing your model to give the wrong result.

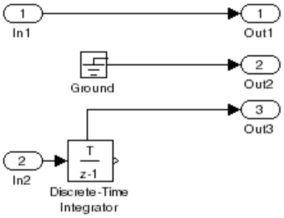
Invalid root Inport/Outport block connection. Specifies the diagnostic action to take during code generation if Simulink detects invalid internal connections to this model's root-level Output port blocks.

When this option is set to `error`, Simulink reports an error if any of the following types of connections appear in this model.

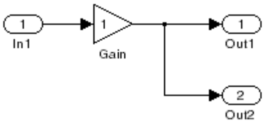
- A root Output port is connected directly or indirectly to more than one nonvirtual block port, for example:



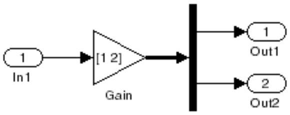
- A root Output port is connected to a root Inport block, a Ground block, or a nondata port (e.g, a state port).



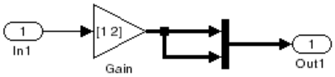
- Two root Outport blocks cannot be connected to the same block port.



- An Outport block cannot be connected to some elements of a block output and not others.



- An Outport block cannot be connected more than once to the same element.



If you select none (the default), Simulink silently inserts blocks to satisfy the constraints wherever possible. In a few cases (such as function-call feedback loops), the inserted blocks may introduce delays and thus may change simulation results.

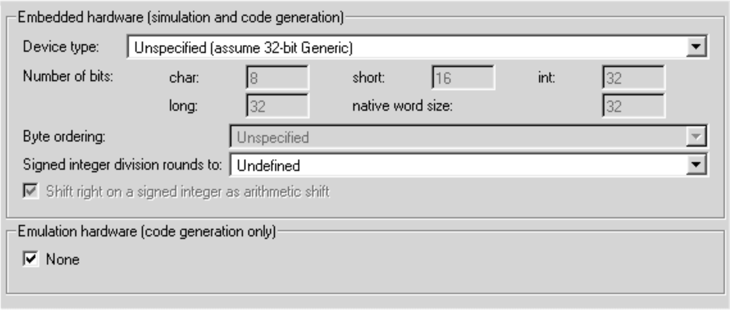
If you select warning, Simulink warns you that a connection constraint has been violated and attempts to satisfy the constraint by inserting hidden blocks.

Auto-inserting hidden blocks to eliminate root I/O problems stops at subsystem boundaries. Therefore, you may need to manually modify models with subsystems that violate any of the above constraints.

Unsupported data logging. Specifies the diagnostic action to take if this model contains To Workspace blocks or Scope blocks with data logging enabled. The default action warns you that Simulink does not support use of these blocks to log data from referenced models. See "Logging Referenced Model Signals" on page 6-28 for information on how to log signals from a reference to this model.

Hardware Implementation Pane

This pane applies to models of computer-based systems, such as embedded controllers. It allows you to specify the characteristics of the hardware to be used to implement the system represented by this model. This in turn enables simulation of the model to detect error conditions that could arise on the target hardware, such as hardware overflow.



Embedded hardware (simulation and code generation)

Device type: Unspecified (assume 32-bit Generic)

Number of bits: char: 8 short: 16 int: 32

long: 32 native word size: 32

Byte ordering: Unspecified

Signed integer division rounds to: Undefined

☑ Shift right on a signed integer as arithmetic shift

Emulation hardware (code generation only)

☑ None

This pane contains the following groups of controls.

Embedded hardware

This group of controls enables you to specify the characteristics of the hardware that will be used to implement the production version of the system represented by this model. (See “Emulation hardware” on page 10-66 for information on specifying the characteristics of hardware used to emulate the production hardware.) This group includes the following controls.

Device type. Specifies the type of hardware that will be used to implement the production version of the system represented by this model. The adjacent list lists types of hardware that Simulink knows about and hence does not require you to enter their characteristics. If your production hardware does not match any of the listed types, select `Unspecified (assume 32-bit Generic)` if it has the characteristics of a generic 32-bit microprocessor; otherwise, `Custom`.

Number of bits. This group of controls specifies the length in bits of C data types supported by the selected device type. Simulink disables these controls if it knows the data type lengths for the selected device type.

Native word size. Specifies the word length in bits of the selected production hardware device type. Simulink disables this field if it knows the word length of the selected device type.

Signed integer division rounds to. Specifies how an ANSI C conforming compiler used to compile code for the production hardware rounds the result of dividing one signed integer by another to produce a signed integer quotient. The options are

- `Zero`

  If the ideal quotient is between two integers, the compiler chooses the integer that is closest to zero as the result.
- `Floor`

  If the ideal quotient is between two integers, the compiler chooses the integer that is closest to negative infinity as the result.
- `Undefined`

  The compiler's rounding behavior is undefined if either or both operands are negative.

The following table illustrates the compiler behavior specified by these options.

N	D	Ideal N/D	Zero	Floor	Undefined
33	4	8.25	8	8	8
-33	4	-8.25	-8	-9	-8 or -9
33	-4	-8.25	-8	-9	-8 or -9
-33	-4	8.25	8	8	-8 or -9

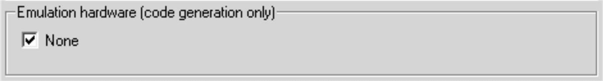
The setting of this option affects only generation of code from the model (see the Real-Time Workshop documentation for information on how this option affects code generation). Use the **Round integer calculations toward** parameter settings on your model's blocks to simulate the rounding behavior of the C compiler that you intend to use to compile code generated from the model. This setting appears on the **Signal data type** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product and Sum blocks.

Shift right on a signed integer as arithmetic shift. Select this option if the C compiler implements a signed integer right shift as an arithmetic right shift. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number in twos complement notation. It is equivalent to dividing the number by 2. This setting affects only code generation.

Byte ordering. Specifies the significance of the first byte of a data word of the target hardware. Select `Big Endian` if the first byte is the most significant, `Little Endian` if it is the least significant, or `Unspecified` if the significance is unknown. This setting affects only code generation. See the Real-Time Workshop documentation for more information.

Emulation hardware

This group of controls allows you to specify the characteristics of hardware used to test code generated from this model.

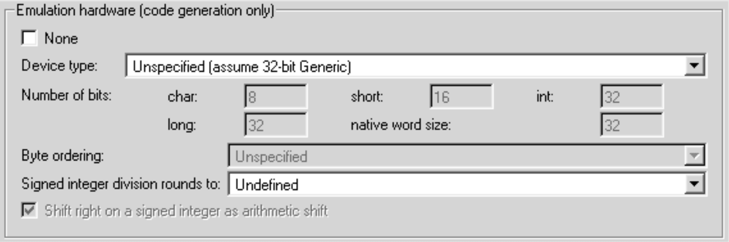


Emulation hardware (code generation only)

☑ None

Initially, this group of controls has only one control.

None. If checked, this check box specifies that the hardware used to test the code generated from this model is the same as the production hardware or has the same characteristics. If you plan to use emulation hardware that has different characteristics, unselect this check box. This causes Simulink to expand the group to display controls that allow you to specify the characteristics of the emulation hardware.



Emulation hardware (code generation only)

☐ None

Device type: Unspecified (assume 32-bit Generic)

Number of bits: char: 8 short: 16 int: 32

long: 32 native word size: 32

Byte ordering: Unspecified

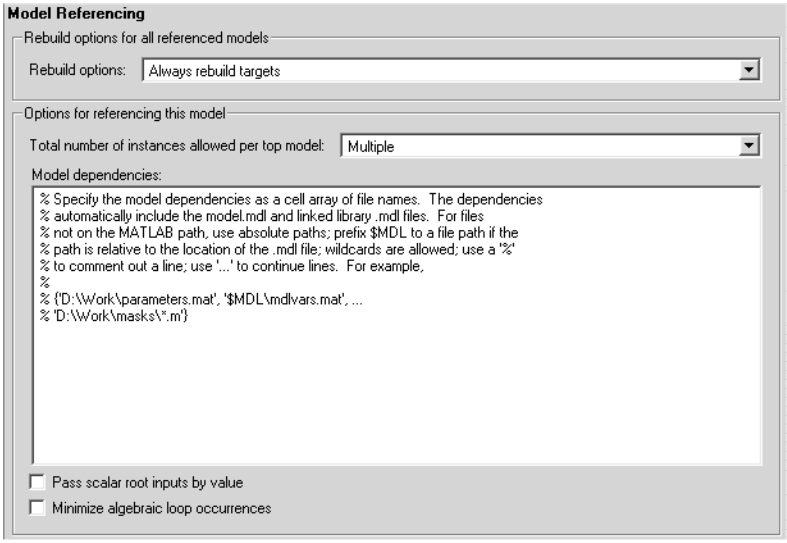
Signed integer division rounds to: Undefined

☑ Shift right on a signed integer as arithmetic shift

The additional controls are identical to the ones used to specify the characteristics of the target hardware for your system. See "Embedded hardware" on page 10-64 for information on using these controls.

Model Referencing Pane

The **Model Referencing** pane allows you to specify options for including other models in this model and this model in other models and for building simulation and code generation targets.



Model Referencing

Rebuild options for all referenced models

Rebuild options: Always rebuild targets

Options for referencing this model

Total number of instances allowed per top model: Multiple

Model dependencies:

```
% Specify the model dependencies as a cell array of file names.  The dependencies
% automatically include the model.mdl and linked library .mdl files.  For files
% not on the MATLAB path, use absolute paths; prefix $MDL to a file path if the
% path is relative to the location of the .mdl file; wildcards are allowed; use a '%'
% to comment out a line; use '...' to continue lines.  For example,
%
% {'D:\Work\parameters.mat', '$MDL\mdlvars.mat', ...
% 'D:\Work\masks\*.m'}
```

☐ Pass scalar root inputs by value

☐ Minimize algebraic loop occurrences

Note The option descriptions use the term *this model* to refer to the model that you are configuring and the term *referenced model* to designate models referenced by *this model*.

The pane includes controls for specifying options for

- Including other models in this model (see "Rebuild options for all referenced models" on page 10-67)
- Including the current model in other models (see "Options for referencing this model" on page 10-69)

Rebuild options for all referenced models

This group allows you to specify rebuild options for models directly or indirectly referenced by this model. It includes the following controls.

Rebuild targets. This control specifies whether to rebuild simulation and Real-Time Workshop targets for referenced models before updating, simulating, or generating code from this model. This includes models indirectly referenced by this model. The options, in order from safe and slow to fast and risky, are

- `Always rebuild targets`

  Always rebuild all targets referenced by this model before simulating, updating, or generating code from it.

- `If any changes detected` (the default)

  Rebuild the target for a referenced model if Simulink detects any changes of any kind in the target's dependencies. The dependencies include

  - The referenced model's model file
  - Block library files used by the referenced model
  - Targets of models referenced by the referenced model
  - S-functions and associated TLC files used by the referenced model
  - User-specified dependencies (see "Model dependencies" on page 10-70)
  - Workspace variables used by the referenced model

  This also checks for changes in the compiled form of the referenced model. Checking the compiled model can detect some changes that occur even in dependencies that you do not specify.

- `If any changes in known dependencies detected`

  Rebuild a target if Simulink detects any changes in known target dependencies (see above) since the target was last built. This option ignores cosmetic changes, such as annotation changes, in the referenced model and in any block library dependencies, thus preventing unnecessary rebuilds. However, before selecting it, you should be certain that you have specified every user-created dependency (e.g., M-files or MAT-files) for this model to ensure that all targets that need to be rebuilt are rebuilt. Otherwise, invalid simulation results may occur.

  Note that this option cannot detect changes in unspecified dependencies, such as M-files used to initialize block masks. If you suspect that a model has such unknown dependencies, you can still guarantee valid simulation by selecting the `Always rebuild targets` or the `If any changes detected` option.

- `Never rebuild targets`

  Never rebuild targets before simulating or generating code from this model. If you are certain that your targets are up-to-date, you can use this option to avoid time-consuming target dependency checking when simulating, updating, or generating code from a model. Use this option with caution because it may lead to invalid results if referenced model targets are not in fact up-to-date.

Note It is a good idea to use the `Always rebuild targets` option before deployment of a model to assure that all the model reference targets are up-to-date.

Never rebuild targets diagnostic. This control appears only if you select the `Never rebuild targets` option. It allows you to specify the diagnostic action that Simulink should take if it detects a target that needs to be rebuilt. The options are

- `Error if targets require rebuild` (the default)
- `Warn if targets require rebuild`
- `None`

  Selecting `None` bypasses dependency checking, and thus enables faster updating, simulation, and code generation, but can cause models that are not up-to-date to malfunction or generate incorrect results.

Options for referencing this model

This group of controls specifies options for including this model in other models. It includes the following controls.

Total number of instances allowed per top model. This option allows you to specify how many references to this model (i.e., the model you are configuring) can safely occur in another model. The options are

- `One`
- `Multiple` (the default)
- `None`

If you specify `None`, and a reference to this model occurs in another model (including its model references), Simulink displays an error when you try to simulate or update the root model. Simulink similarly displays an error, if you specify `One` and multiple references to this model occur in a root model (including its model references). If you specify multiple and Simulink determines that for some reason this model cannot be mutiply referenced, Simulink displays an error when the model that references it is compiled or simulated. This occurs even if the model is referenced only once.

Model dependencies. Specifies files on which this model relies. They are typically MAT-files and M-files used to initialize parameters and to provide data.

Specify the dependencies as a cell array of strings, where each cell array entry is the filename or path of a dependent file. These filenames may include spaces and must include file extensions (e.g., `.m`, `.mat`, etc.).

Prefix the token `$MDL` to a dependency to indicate that the path to the dependency is relative to the location of this model file.

If Simulink cannot find a specified dependent file when you update or simulate a model that references this model, Simulink displays an error.

Pass scalar root inputs by value. Checking this option causes a model that calls (i.e., references) this model to pass this model's scalar inputs by value. Otherwise, the calling model passes the inputs by reference, i.e., it passes the addresses of the inputs rather than the input values.

Passing roots by value allows this model to read its scalar inputs from register or local memory which is faster than reading the inputs from their original locations. However, this option can lead to incorrect results if the model's root scalar inputs can change within a time step. This can happen, for instance, if this model's inputs and outputs share memory locations (e.g., as a result of a feedback loop) and the model is invoked multiple times in a time step (i.e., by a Function-Call Subsystem). In such cases, this model sees scalar input changes that occur in the same time step only if the inputs are passed by reference. That is why this option is off by default. If you are certain that this model is not referenced in contexts where its inputs can change within a time step, select this option to generate more efficient code for this model.

Note Selecting this option can affect reuse of code generated for subsystems. See the Real-Time Workshop documentation for more information.

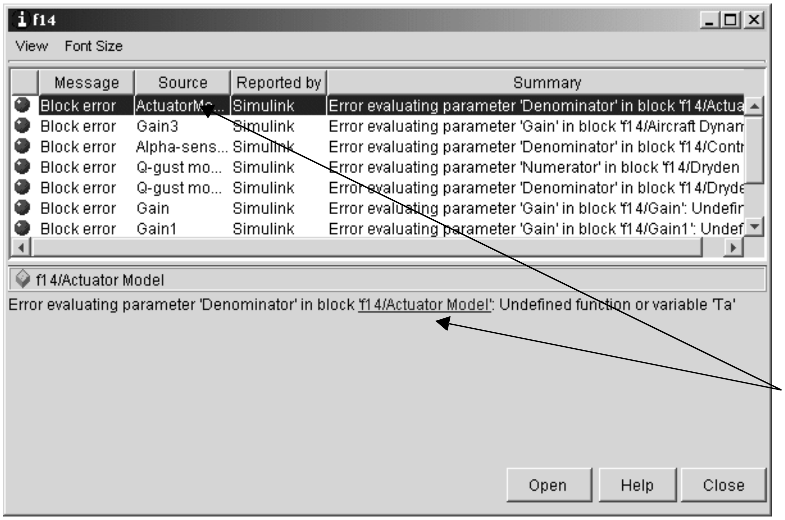
Minimize algebraic loop occurrences. Checking this option causes Simulink to try to eliminate algebraic loops involving this model from models that reference it. Enabling this option disables conditional input branch optimization for simulation and the Real-Time Workshop single update/output function optimization for code generation. See "Eliminating Algebraic Loops" on page 2-27 for more information.

Diagnosing Simulation Errors

If errors occur during a simulation, Simulink halts the simulation, opens the subsystems that caused the error (if necessary), and displays the errors in the Simulation Diagnostics Viewer. The following section explains how to use the viewer to determine the cause of the errors.

Simulation Diagnostics Viewer

The viewer comprises an Error Summary pane and an Error Message pane.



	Message	Source	Reported by	Summary
●	Block error	ActuatorMo...	Simulink	Error evaluating parameter 'Denominator' in block 'f14/Actua
●	Block error	Gain3	Simulink	Error evaluating parameter 'Gain' in block 'f14/Aircraft Dynar
●	Block error	Alpha-sens...	Simulink	Error evaluating parameter 'Denominator' in block 'f14/Contr
●	Block error	Q-gust mo...	Simulink	Error evaluating parameter 'Numerator' in block 'f14/Dryden
●	Block error	Q-gust mo...	Simulink	Error evaluating parameter 'Denominator' in block 'f14/Dryde
●	Block error	Gain	Simulink	Error evaluating parameter 'Gain' in block 'f14/Gain': Undefir
●	Block error	Gain1	Simulink	Error evaluating parameter 'Gain' in block 'f14/Gain1': Undef

f14/Actuator Model

Error evaluating parameter 'Denominator' in block 'f14/Actuator Model': Undefined function or variable 'Ta'

Click to display error source.

Error Summary Pane

The upper pane lists the errors that caused Simulink to terminate the simulation. The pane displays 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

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

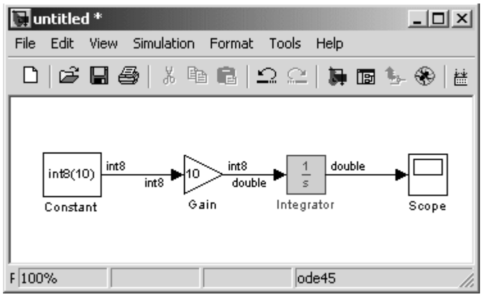
Summary. Error message, abbreviated to fit in the column

You can remove any of these columns of information to make more room for the others. To remove a column, select the viewer's **View** menu and uncheck the corresponding item.

Error Message Pane

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

In addition to displaying the viewer, Simulink opens (if necessary) the subsystem that contains the first error source and highlights the source.



You can display the sources of other errors by clicking anywhere in the error message in the upper pane, by clicking the name of the error source in the error message (highlighted in blue), or by clicking the **Open** button on the viewer.

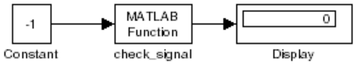
Changing Font Size

To change the size of the font used to display errors, select **Font Size** from the viewer's menu bar. A menu of font sizes appears. Select the desired font size from the menu.

Creating Custom Simulation Error Messages

The Simulation Diagnostics Viewer displays the output of any instance of the MATLAB error function executed during a simulation, including instances invoked by block or model callbacks or S-functions that you create or that are executed by the MATLAB Fcn block. Thus, you can use the MATLAB error function in callbacks and S-functions or in the MATLAB Fcn block to create simulation error messages specific to your application.

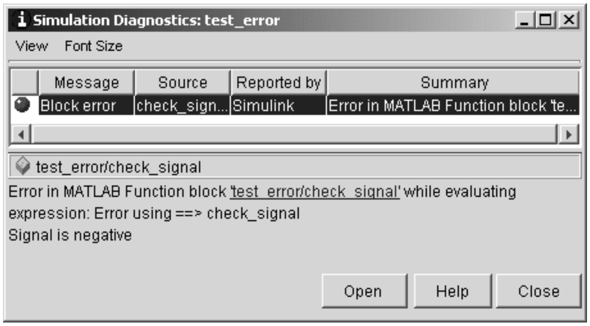
For example, in the following model,



the MATLAB Fcn block invokes the following function:

```
function y=check_signal(x)
  if x<0
    error('Signal is negative.');
  else
    y=x;
  end
```

Executing this model displays an error message in the Simulation Diagnostics Viewer.



	Message	Source	Reported by	Summary
●	Block error	check_sign...	Simulink	Error in MATLAB Function block 'te...

test_error/check_signal

Error in MATLAB Function block 'test_error/check_signal' while evaluating expression: Error using ==> check_signal
Signal is negative

Including Hyperlinks in Error Messages

You can include hyperlinks to blocks, text files, and directories.

To include a hyperlink to a block, path, or directory, include the item's path in the error message enclosed in quotation marks, e.g.,

- `error ('Error evaluating parameter in block "mymodel/Mu"')` displays a text hyperlink to the block Mu in the current model in the error message. Clicking the hyperlink displays the block in the model window.

- `error ('Error reading data from "c:/work/test.data"')`

  displays a text hyperlink to the file `test.data` in the error message. Clicking the link displays the file in your preferred MATLAB editor.
- `error ('Could not find data in directory "c:/work"')`

  displays a text hyperlink to the c:/`work` directory. Clicking the link opens a system command window (shell) and sets its working directory to `c:/work`.

Note The text hyperlink is enabled only if the corresponding block exists in the current model or if the corresponding file or directory exists on the user's system.

Improving Simulation Performance and Accuracy

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

The solvers handle most model simulations accurately and efficiently with their default parameter values. However, some models yield better results if you adjust solver parameters. Also, if you know information about your model's behavior, your simulation results can be improved if you provide this information to the solver.

Speeding Up the Simulation

Slow simulation speed can have many causes. Here are a few:

- Your model includes a MATLAB Fcn block. When a model includes a MATLAB Fcn block, the MATLAB interpreter is called at each time step, drastically slowing down the simulation. Use the built-in Fcn block or Math Function block whenever possible.
- Your model includes an M-file S-function. M-file S-functions also cause the MATLAB interpreter to be called at each time step. Consider either converting the S-function to a subsystem or to a C-MEX file S-function.
- Your model includes a Memory block. Using a Memory block causes the variable-order solvers (`ode15s` and `ode113`) to be reset back to order 1 at each time step.
- The maximum step size is too small. If you changed the maximum step size, try running the simulation again with the default value (`auto`).
- Did you ask for too much accuracy? The default relative tolerance (0.1% accuracy) is usually sufficient. For models with states that go to zero, if the absolute tolerance parameter is too small, the simulation can take too many steps around the near-zero state values. See the discussion of error in "Maximum order" on page 10-35.
- The time scale might be too long. Reduce the time interval.
- The problem might be stiff, but you are using a nonstiff solver. Try using `ode15s`.
- The model uses sample times that are not multiples of each other. Mixing sample times that are not multiples of each other causes the solver to take small enough steps to ensure sample time hits for all sample times.

- The model contains an algebraic loop. The solutions to algebraic loops are iteratively computed at every time step. Therefore, they severely degrade performance. For more information, see "Algebraic Loops" on page 2-24.
- 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, either reduce the relative tolerance to 1e-4 (the default is 1e-3) or reduce the absolute tolerance and run it again. Compare the results of both simulations. If the results are not significantly different, you can feel confident that the solution has converged.

If the simulation misses significant behavior at its start, reduce the initial step size to ensure that the simulation does not step over the significant behavior.

If the simulation results become unstable over time,

- Your system might be unstable.
- If you are using `ode15s`, you might need to restrict the maximum order to 2 (the maximum order for which the solver is A-stable) or try using the `ode23s` solver.

If the simulation results do not appear to be accurate,

- For a model that has states whose values approach zero, if the absolute tolerance parameter is too large, the simulation takes too few steps around areas of near-zero state values. Reduce this parameter value or adjust it for individual states in the Integrator dialog box.
- If reducing the absolute tolerances does not sufficiently improve the accuracy, reduce the size of the relative tolerance parameter to reduce the acceptable error and force smaller step sizes and more steps.

Running a Simulation Programmatically

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 **Configuration Parameters** dialog box settings.

For detailed syntax for the `sim` command, see the documentation for the `sim` command. The `options` parameter is a structure that supplies additional configuration parameters, including the solver name and error tolerances. You define parameters in the `options` structure using the `simset` command (see `simset`). The configuration parameters are discussed in "Configuration Sets" on page 10-26.

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. 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'`.

Similarly, you can use the `get_param` command to check the status of a simulation. The format of the `get_param` command for this use is

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

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