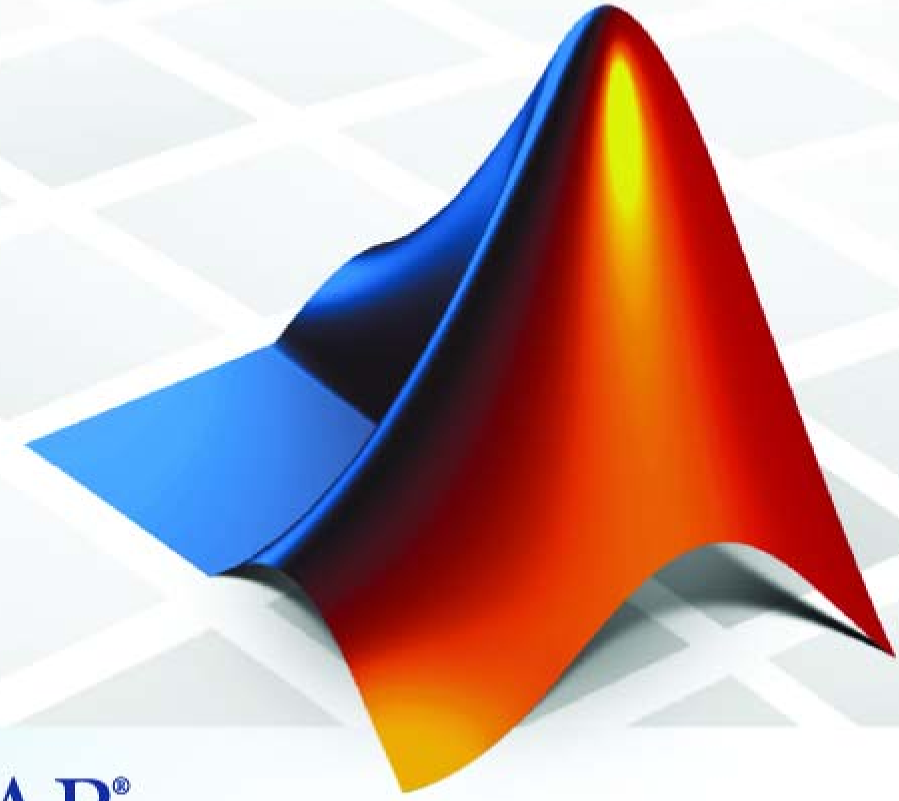
Simulink® 6

Using Simulink®



MATLAB® & SIMULINK®

The MathWorks

Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Using Simulink

Trademarks

Patents

Revision History

November 1990	First printing	New for Simulink 1
December 1996	Second printing	Revised for Simulink 2
January 1999	Third printing	Revised for Simulink 3 (Release 11)
November 2000	Fourth printing	Revised for Simulink 4 (Release 12)
July 2002	Fifth printing	Revised for Simulink 5 (Release 13)
April 2003	Online only	Revised for Simulink 5.1 (Release 13SP1)
April 2004	Online only	Revised for Simulink 5.1.1 (Release 13SP1+)
June 2004	Sixth printing	Revised for Simulink 5.0 (Release 14)
October 2004	Seventh printing	Revised for Simulink 6.1 (Release 14SP1)
March 2005	Online only	Revised for Simulink 6.2 (Release 14SP2)
September 2005	Eighth printing	Revised for Simulink 6.3 (Release 14SP3)
March 2006	Online only	Revised for Simulink 6.4 (Release 2006a)
March 2006	Ninth printing	Revised for Simulink 6.4 (Release 2006a)
September 2006	Online only	Revised for Simulink 6.5 (Release 2006b)
March 2007	Online only	Revised for Simulink 6.6 (Release 2007a)

Contents

1 How Simulink Works

2 Simulink Basics

Creating a Model

3

4 Working with Blocks

5 Working with Signals

6 Using Composite Signals

7 Working with Data

8 Working with Lookup Tables

9 Modeling with Simulink

10 Exploring, Searching, and Browsing Models

11 Running Simulations

12 Analyzing Simulation Results

13 Creating Block Masks

14 Simulink Debugger

15 Simulink Accelerator

16 Customizing the Simulink User Interface

17 Using the Embedded MATLAB Function Block

18 PrintFrame Editor

A Examples

Index

1

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 3, “Creating a Model”) selected from libraries of block types (see Blocks — Alphabetical List) via a library browser. Simulink provides libraries of blocks representing elementary systems that can be used as 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. User-defined blocks are called custom blocks.

Time

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

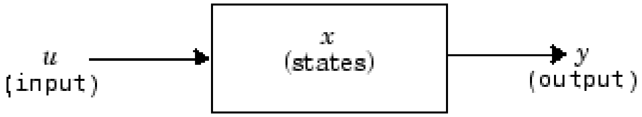
States

Typically the current values of some system, and hence model, outputs are functions of the previous values of temporal variables. Such variables are called states. Computing a model's outputs from a block diagram hence entails saving the value of states at the current time step for use in computing the outputs at a subsequent time step. 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
- Variable Transport Delay
- Zero-Pole

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

Working with States

Simulink provides the following facilities for determining, initializing, and logging a model's states during simulation:

- The `model` command displays information about the states defined by a model, including the total number of states defined by the model, the block that defines each state, and the initial value of each state.
- The Simulink debugger displays the value of a state at each time step during a simulation, and the Simulink debugger's `states` command displays information about the model's current states (see Chapter 14, "Simulink Debugger").
- The **Data Import/Export** pane of a model's Configuration Parameters dialog box (see "Importing and Exporting States" on page 11-31) allows you to specify initial values for a model's states and instruct Simulink to record the values of the states at each time step during simulation as an array or structure variable in the MATLAB® workspace.

- The Block Parameters dialog box (and the `ContinuousStateAttributes` parameter) allows you to give names to states for those blocks (such as the Integrator) that employ continuous states. This can simplify analyzing data logged for states, especially when a block has multiple states.

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

Continuous States

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

In general, excluding simple dynamic systems, analytical methods do not exist for integrating the states of real-world dynamic systems represented by ordinary differential equations. Integrating the states requires the use of numerical methods called ODE solvers. These various methods trade computational accuracy for computational workload. 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 its value at the current time step and its value at the previous time step. 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 discrete blocks to model the dependency of a system's discrete states on its inputs.

As with continuous states, discrete states set a constraint on the simulation time step size. Specifically, the step size must ensure that all the sample times of the model's states are hit. 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 system that has both discrete and continuous states. Strictly speaking, Simulink treats any model that has both continuous and discrete sample times as a hybrid model, presuming that the model has both continuous and discrete states. Solving such a model entails choosing a step size that satisfies both the precision constraint on the continuous state integration and the sample time hit constraint on the discrete states. 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.

Tunable Parameters

Many block parameters are tunable. A *tunable parameter* is a parameter whose value can be changed without recompiling the model (see “Model Compilation” on page 1-15 for more information on compiling a Simulink 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.

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

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

You can use the **Inline parameters** option on the **Optimization** pane of the **Configuration Parameters** dialog box to specify that all parameters in your model are nontunable except for those that you specify. This can speed up execution of large models and enable generation of faster code from your model. See “Configuration Parameters Dialog Box” on page 11-66 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. Most blocks allow 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 does not specify its sample time 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. See also “Sample Time Propagation” on page 1-42 for a description of how Simulink uses a process called sample time propagation to determine the sample times of blocks that inherit their 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 “Displaying Sample Time Colors” on page 3-10 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*). 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. Subsystems do not define a separate block diagram.

Simulink differentiates between two different types of subsystems: virtual and nonvirtual. 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 which is referred to as the root system, and several lower-level systems derived from nonvirtual subsystems and other elements in the block diagram. You will see these systems in the Simulink Debugger. The act of creating these internal systems is often referred to as flattening the model hierarchy.

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 3-40). Conditionally executed subsystems are atomic, i.e., the equations that they define are evaluated as a unit.

Atomic Subsystems

Unconditionally executed subsystems are virtual by default. You can, however, designate an unconditionally executed subsystem as atomic (see the Atomic Subsystem block for more information). This is useful if you need to ensure that the equations defined by a subsystem are evaluated 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 arrowhead. The source of the signal corresponds to the block that writes to the signal during evaluation of its block methods (equations). The destinations of the signal are blocks that read the signal during the evaluation of 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" on page 1-15
- "Link Phase" on page 1-16
- "Simulation Loop Phase" on page 1-16
- "Solvers" on page 1-18
- "Zero-Crossing Detection" on page 1-20
- "Algebraic Loops" on page 1-26

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 1-18).

- Determines the block sorted order (see “Controlling and Displaying the Sorted Order” on page 4-31 for more information).
- Determines the sample times of all blocks in the model whose sample times you did not explicitly specify (see “Sample Time Propagation” on page 1-42).

Link Phase

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

Method Execution Lists

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

Block Priorities

Simulink allows you to assign update priorities to blocks (see “Assigning Block Priorities” on page 4-33). 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 1-18) used to compute the system’s continuous states, the system’s fundamental sample time (see “Modeling and Simulating

Discrete Systems" on page 1-35), and whether the system's continuous states have discontinuities (see "Zero-Crossing Detection" on page 1-20).

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 1-18).

   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 1-20 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 11-11).

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 11-11).

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 1-41 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 velocity. 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 1-24 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.

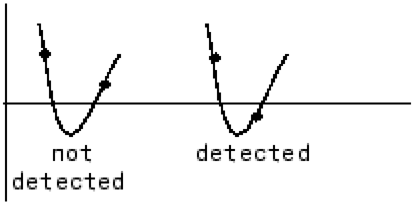
Zero-crossing signals have a direction attribute, which can have these values:

- *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 11-74.

Note Using the `Refine output` and `Produce additional output` options (see “Output options” on page 11-85) do not help locate the missed zero crossings; they help reduce the chance of missing a zero crossing. You should alter the maximum step size or output times.

Preventing Excessive Zero Crossings

It is possible to create models that exhibit high-frequency fluctuations about a discontinuity (i.e., chattering). For example, the `bounce` demo model represents a rubber ball that is thrown into the air and repeatedly bounces on the ground. Each time the ball hits the ground, Simulink encounters a discontinuity in the ball’s velocity and position. Because chattering causes repeated detection of zero crossings, the step sizes of the simulation become very small, essentially halting the simulation.

Simulink checks for excessive occurrences of zero crossings during simulation and reports either a warning or error based on the setting of the **Consecutive zero crossings violation** diagnostic (see “Consecutive zero crossings violation" on page 11-105). To deal with this situation, implement one of the following options:

- You can adjust the criteria that Simulink uses to determine what constitutes an excessive number of zero crossings. In particular, you can use the **Number of consecutive zero crossings allowed** option on the **Solver** pane of the **Configuration Parameters** dialog box to increase the threshold at which Simulink triggers the diagnostic (see "Number of consecutive zero crossings allowed" on page 11-72). Likewise, you can decrease the value of the **Consecutive zero crossings relative tolerance** option (see "Consecutive zero crossings relative tolerance" on page 11-71). Altering these settings may afford your model's behavior more time to recover.
- You can disable zero-crossing detection for the offending block. To do so, uncheck the **Enable zero crossing detection** option on the block's parameter dialog box, and set the **Zero crossing control** option on the **Solver** pane of the **Configuration Parameters** dialog box to `Use local settings`. For more information, see "Zero crossing control" on page 11-71. Locally disabling zero-crossing detection can alleviate the symptoms of this problem while the remainder of your model still benefits from the increased accuracy that zero-crossing detection provides.
- You can disable zero-crossing detection for the entire model. To do so, set the **Zero crossing control** option on the **Solver** pane of the **Configuration Parameters** dialog box to `Disable all`. For more information, see "Zero crossing control" on page 11-71. Although globally disabling zero-crossing detection can alleviate the symptoms of this problem, your model no longer benefits from the increased accuracy that zero-crossing detection provides.

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.

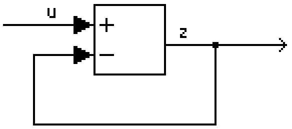
Block	Description of Zero Crossing
Dead Zone	Two: one to detect when the dead zone is entered (the input signal minus the lower limit), and one to detect when the dead zone is exited (the input signal minus the upper limit).
From Workspace	One: to detect when the input signal has a discontinuity in either the rising or falling direction
Hit Crossing	One: to detect when the input crosses the threshold.
If	One: to detect when the If condition is met.
Integrator	If the reset port is present, to detect when a reset occurs. If the output is limited, there are three zero crossings: one to detect when the upper saturation limit is reached, one to detect when the lower saturation limit is reached, and one to detect when saturation is left.
MinMax	One: for each element of the output vector, to detect when an input signal is the new minimum or maximum.
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.
Signal Builder	One: to detect when the input signal has a discontinuity in either the rising or falling direction
Step	One: to detect the step time.
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.
Switch Case	One: to detect when the case condition is met.

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:

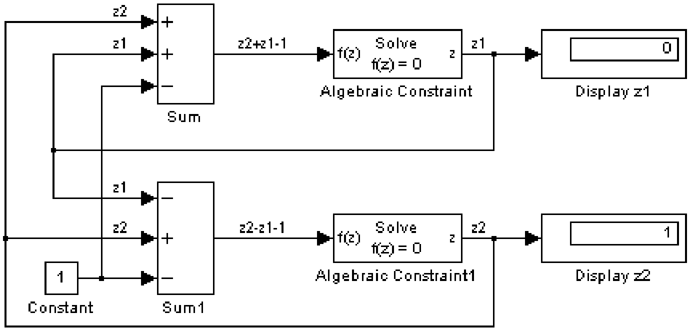
- The Math Function block
- The Gain block
- The Integrator block's initial condition ports
- The Product block
- The State-Space block when there is a nonzero D matrix
- The Sum block
- The Transfer Fcn block when the numerator and denominator are of the same order
- The Zero-Pole block when there are 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 direct feedback path, i.e., the feedback path solely contains blocks with direct feedthrough. You can provide an initial guess of the algebraic state value in the block's dialog box to improve algebraic loop solver efficiency.

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

```
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 an initial guess for the algebraic state variables in a loop.

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.

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`.

Eliminating Algebraic Loops

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

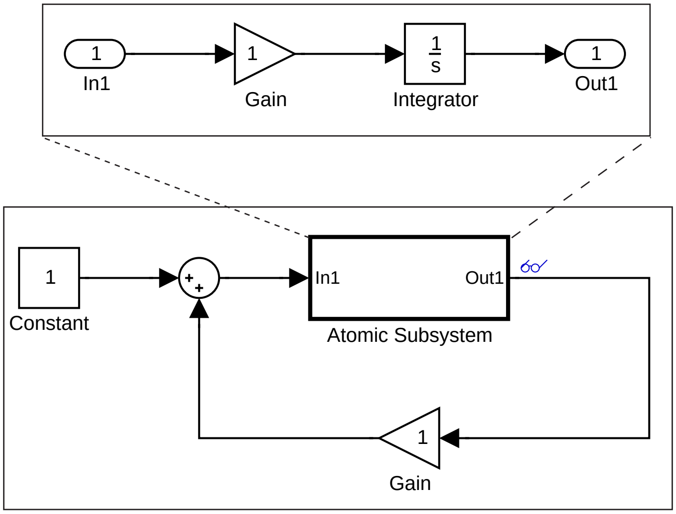
- Atomic Subsystem
- Enabled Subsystem
- Model

To enable automatic algebraic loop elimination for a loop involving a particular instance of an Atomic Subsystem or Enabled Subsystem block, select the **Minimize algebraic loop occurrences** parameter on the block's parameters dialog box. To enable algebraic loop elimination for a loop involving a Model block, check the **Minimize algebraic loop occurrences** parameter on the **Model Referencing Pane** of the **Configuration Parameters** dialog box (see "Model Referencing Pane" on page 11-139) 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.

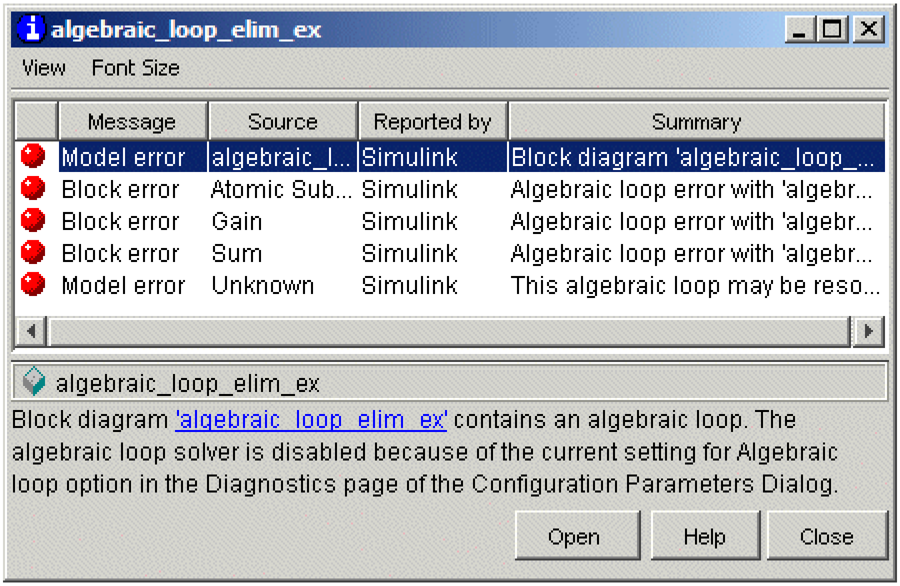
The Simulink **Minimize algebraic loop** solver diagnostic allows you to specify the action Simulink should take, for example, display a warning message, if it is unable to eliminate an algebraic loop involving a block for which algebraic loop elimination is enabled. See "Diagnostics Pane" on page 11-102 for more information.

Algebraic loop minimization is off by default because it is incompatible with conditional input branch optimization in Simulink (see "Optimization Pane" on page 11-86) 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 "Diagnostics Pane" on page 11-102 for more information) reveals that this model contains an algebraic loop involving its atomic subsystem.

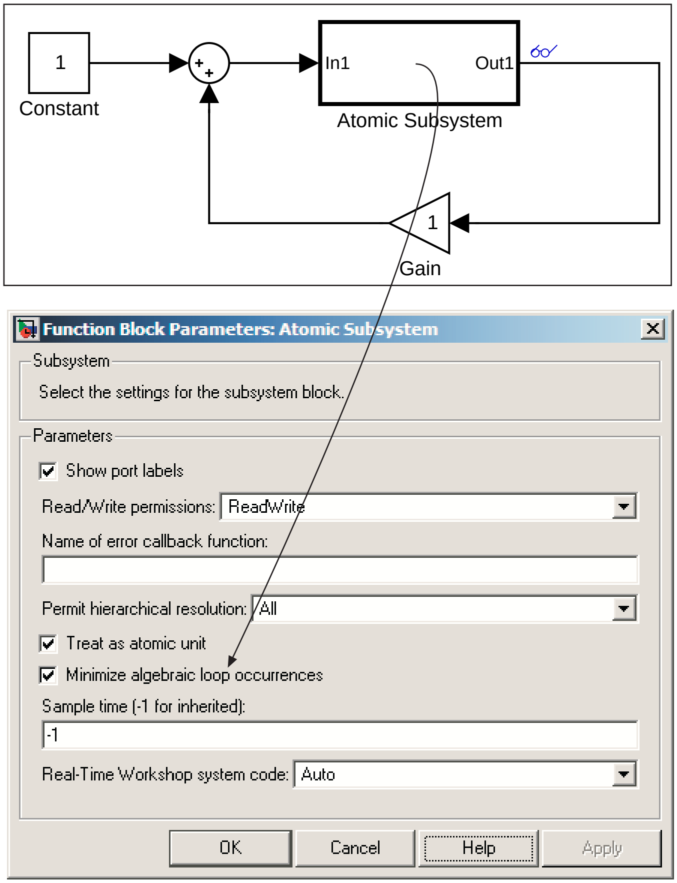


	Message	Source	Reported by	Summary
●	Model error	algebraic_l...	Simulink	Block diagram 'algebraic_loop_...
●	Block error	Atomic Sub...	Simulink	Algebraic loop error with 'algebr...
●	Block error	Gain	Simulink	Algebraic loop error with 'algebr...
●	Block error	Sum	Simulink	Algebraic loop error with 'algebr...
●	Model error	Unknown	Simulink	This algebraic loop may be reso...

algebraic_loop_elim_ex

Block diagram 'algebraic_loop_elim_ex' contains an algebraic loop. The algebraic loop solver is disabled because of the current setting for Algebraic loop option in the Diagnostics page of the Configuration Parameters Dialog.

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



Function Block Parameters: Atomic Subsystem

Subsystem

Select the settings for the subsystem block.

Parameters

☑ Show port labels

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

☑ Treat as atomic unit

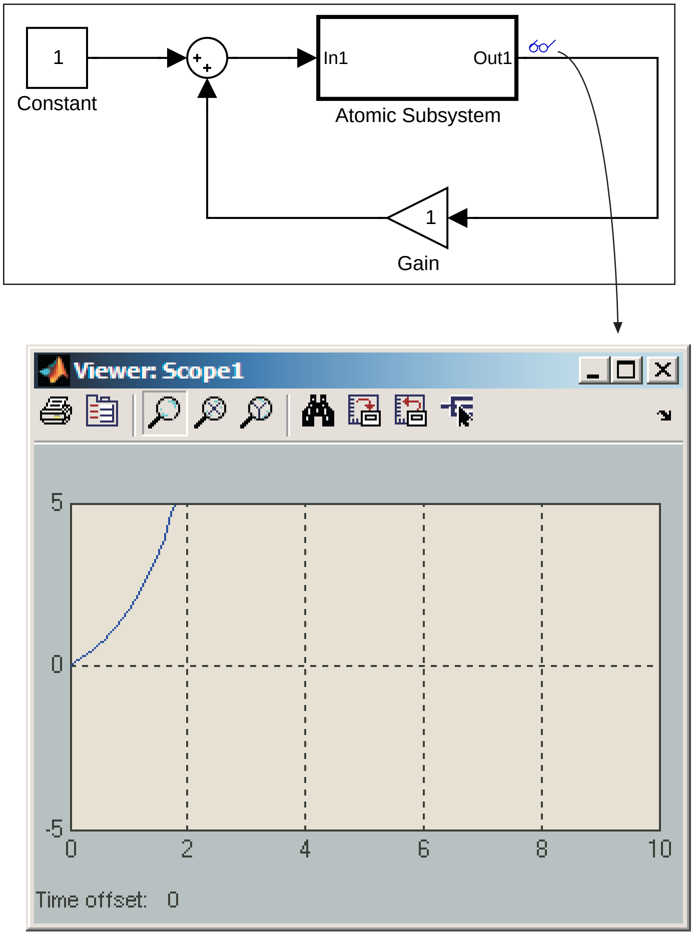
☑ Minimize algebraic loop occurrences

Sample time (-1 for inherited):

-1

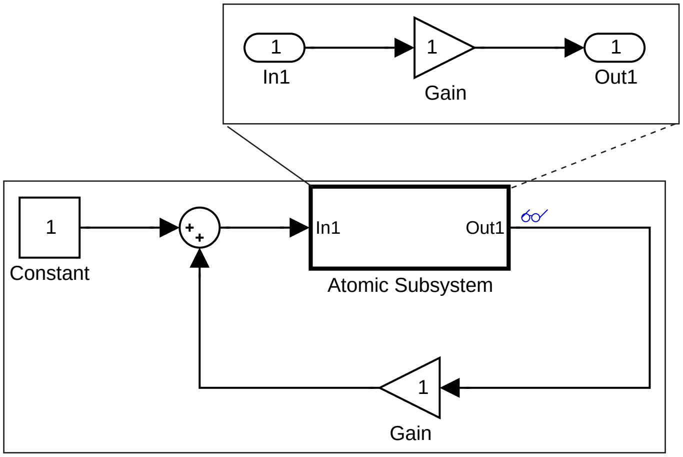
Real-Time Workshop system code: Auto

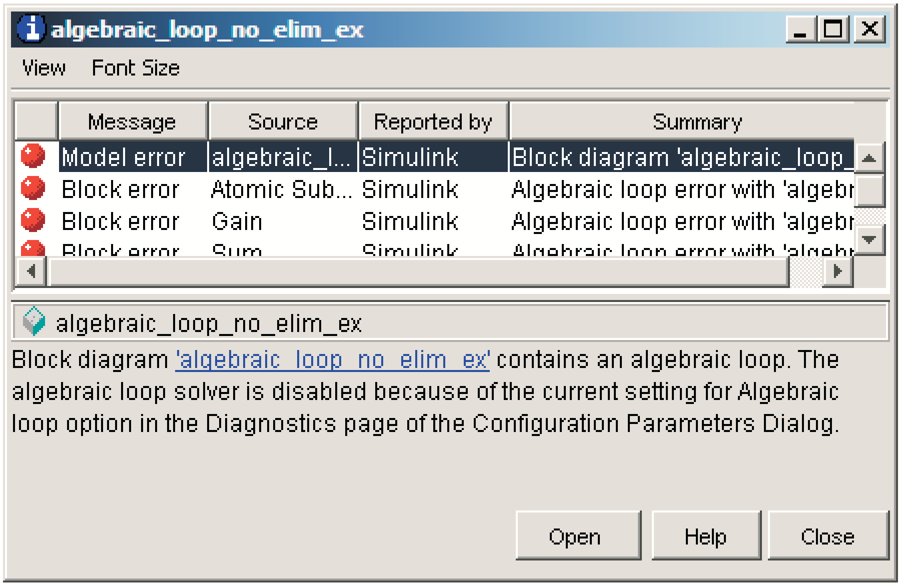
As a result, the model now simulates without error.



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.





Message	Source	Reported by	Summary
Model error	algebraic_l...	Simulink	Block diagram 'algebraic_loop_
Block error	Atomic Sub...	Simulink	Algebraic loop error with 'algebr
Block error	Gain	Simulink	Algebraic loop error with 'algebr
Block error	Sum	Simulink	Algebraic loop error with 'algebr

algebraic_loop_no_elim_ex

Block diagram 'algebraic_loop_no_elim_ex' contains an algebraic loop. The algebraic loop solver is disabled because of the current setting for Algebraic loop option in the Diagnostics page of the Configuration Parameters Dialog.

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 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 their output.

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
$[T_s, T_o]$ $0 < T_s < T_{sim}$ $\|T_o\| < T_p$	Specifies that updates occur at simulation times $t_n = n * T_s + \|T_o\|$ where `n` is an integer in the range $0..T_{sim}/T_s$ 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 1-20). 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.

Sample Time	Usage
`[-2, T`$_{vo}$`]`	Specifies that a block has a *variable sample time*, i.e., computes its output only at times when the output changes. Every block with variable sample time has a unique T_{vo} determined by Simulink. The only built-in Simulink block that can have variable sample time is the Pulse Generator block. The sample time color (see "Displaying Sample Time Colors" on page 3-10) for variable sample time is yellow.
`inf`	The meaning of this sample time depends on whether the active model configuration's inline parameters optimization (see "Inline parameters" on page 11-90) is enabled. If the inline parameters optimization is enabled, `inf` signifies that the block's output can never change (see "Constant Sample Time" on page 1-44). 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

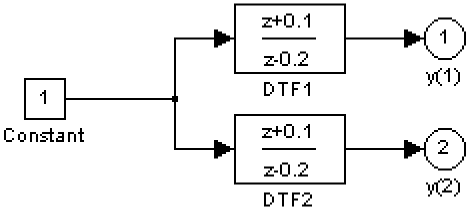
During the compilation phase of a simulation, Simulink determines the sample time of the block from its SampleTime parameter (if it has a SampleTime parameter), sample-time inheritance, or block type (Continuous blocks always have a continuous sample time). It is this compiled sample time that determines 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.

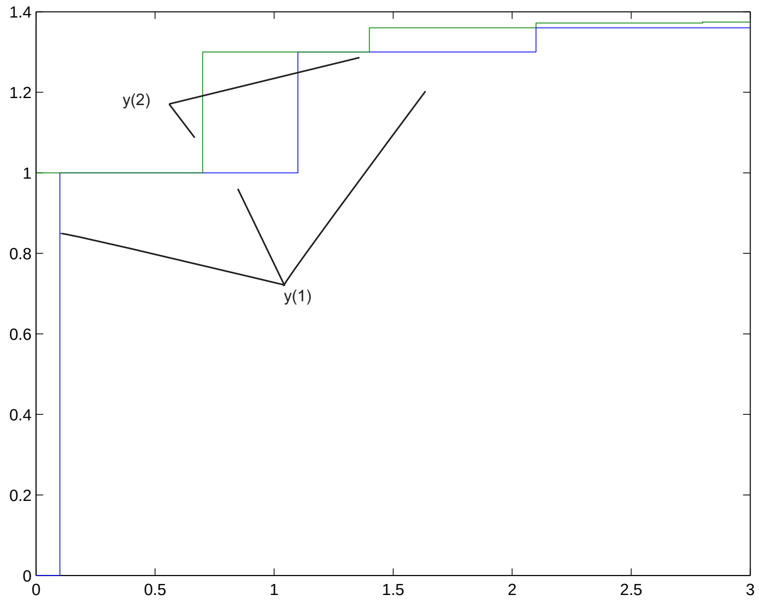


For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to `[1 0.1]`, which gives it an offset of `0.1`. The DTF2 Discrete Transfer Fcn block's **Sample time** is set to `0.7`, with no offset.

Starting the simulation and plotting the outputs using the stairs function

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

produces this plot



See "Running a Simulation Programmatically" on page 11-152 for information on the sim command.

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.

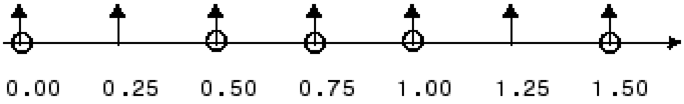
Determining Step Size for Discrete Systems

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.

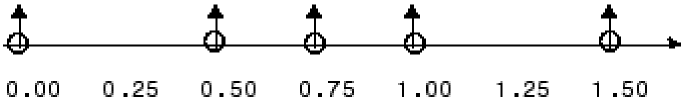
The *fundamental sample time* of a discrete system is the greatest integer divisor of the system's actual sample times. For example, suppose that a system has sample times of 0.25 and 0.5 second. The fundamental sample time in this case is 0.25 second. Suppose, instead, the sample times are 0.5 and 0.75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

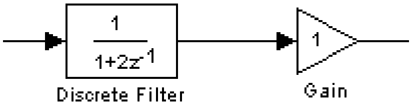


Variable-Step Solver

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, using the following rules.

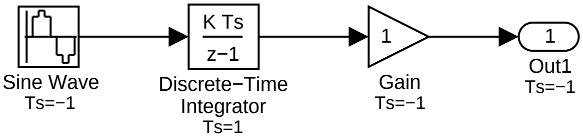
- If all the inputs have the same sample time, Simulink assigns that sample time to the block.
- If the inputs have different sample times and if all the input sample times are integer multiples of the fastest input sample time, the block is assigned the sample time of the fastest input.
- If the inputs have different sample times and some of the input sample times are not integer multiples of the fastest sample time and a variable-step solver is being used, the block is assigned continuous sample time.

- If the inputs have different sample times and some of the input sample times are not integer multiples of the fastest sample time and a fixed-step solver is being used, and the greatest common divisor of the sample times (the fundamental sample time) can be computed, the block is assigned the fundamental sample time; otherwise, in this case, the block is assigned continuous sample time.

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 3-75 for more information).

Propagating Sample Times Back to Source Blocks

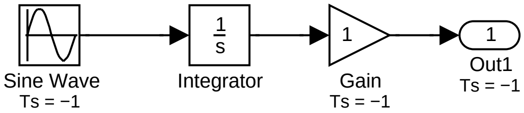
When you update or simulate a model that specifies a source block's sample time as inherited (-1), Simulink may back propagate the source block's sample time, i.e., it sets the source block's sample time to be the same as the sample time specified or inherited by the block to which the source block is connected. Simulink does this only if it can do so without changing the results of simulating the model. For example, in the model below, Simulink recognizes that the Sine Wave block is driving a Discrete-Time Integrator block whose sample time is 1, so it assigns the Sine Wave block a sample time of 1.



You can verify this 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 Sine Wave and Gain blocks to change to continuous blocks, as indicated by their being colored black.



Note Back propagation makes the sample times of a model's sources dependent on block connectivity. If you change the connectivity of a model whose sources inherit sample times, you can inadvertently change the source sample times. For this reason, when you update or simulate a model, Simulink by default displays warnings at the MATLAB command line if the model contains sources that inherit their sample times. See "Source block specifies -1 sample time" on page 11-109 for more information.

Constant Sample Time

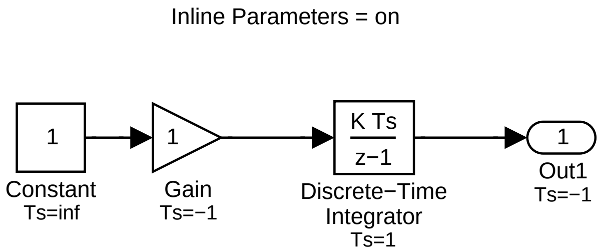
A block whose output cannot change from its initial value during a simulation is said to have *constant sample time*. A block has constant sample time if it satisfies both of the following conditions:

- All of its parameters are nontunable, either because they are inherently nontunable or because they have been inlined (see "Inline parameters" on page 11-90).
- The block's sample time has been declared infinite (`inf`) or its sample time is declared to be inherited and it inherits a constant sample time from another block to which it is connected.

When Simulink updates a model, for example, at the beginning of a simulation, Simulink determines which blocks, if any, have constant sample time, and computes the initial values of the output ports. During the simulation, Simulink uses the initial values whenever the outputs of blocks with constant sample time are required, thus avoiding unnecessary computations.

You can determine which blocks have constant sample time by selecting **Sample Time Colors** from the **Format** menu and updating the model. Blocks with constant sample time are colored magenta.

For example, in this model, both the Constant and Gain blocks have constant sample time.



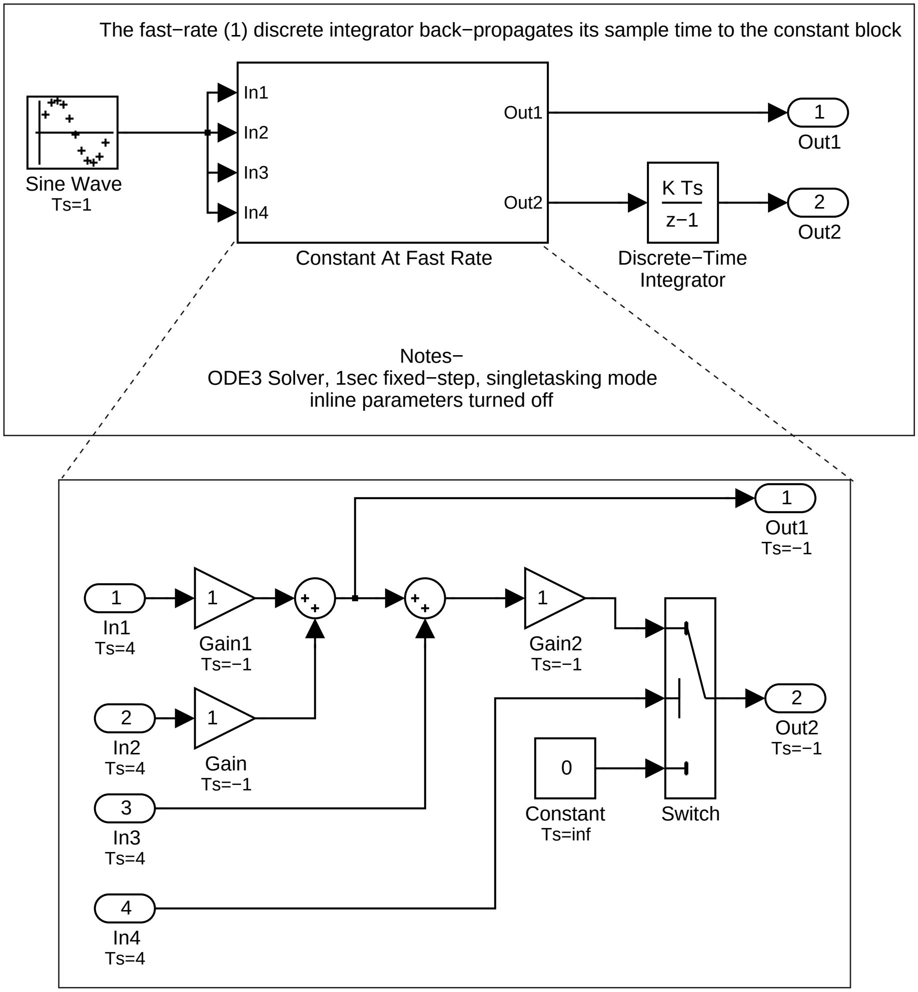
The Gain block has constant sample time because it inherits its sample time from the Constant block and all of the model's parameters are inlined, i.e., nontunable.

Note The Simulink block library includes a few blocks, e.g., the S-Function, Level-2 M-File S-Function, Rate Transition, and Model block, whose ports can produce outputs at different sample rates. It is possible for some of the ports of such blocks to inherit a constant sample time. The ports with constant sample time produce output only once, at the beginning of the simulation. The other ports produce outputs at their sample rates.

How Simulink Treats Blocks with Infinite Sample Times and Tunable Parameters

A block that has tunable parameters cannot have constant sample time even if its sample time is specified to be infinite. This is because the fact that a block has one or more tunable parameters means that you can change the values of its parameters during simulation and hence the value of its outputs. In this case, Simulink uses sample time propagation (see "Sample Time Propagation" on page 1-42) to determine the block's actual sample time.

For example, consider the following model.



In this example, although the Constant block's sample time is specified to be infinite, it cannot have constant sample time because the inlined parameters option is off for this model and therefore the block's **Constant value** parameter is tunable. Since the Constant block's output can change during the simulation, Simulink has to determine a sample time for the block that ensures accurate simulation results. It does this by treating the Constant block's sample time as inherited and using sample time propagation to determine its sample time. The first nonvirtual block in the diagram branch to which the Constant block is connected is the Discrete-Time Integrator block. As a result, the block inherits its sample time (1 sec) via back propagation from the Discrete-Time Integrator block.

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.

Simulink Basics

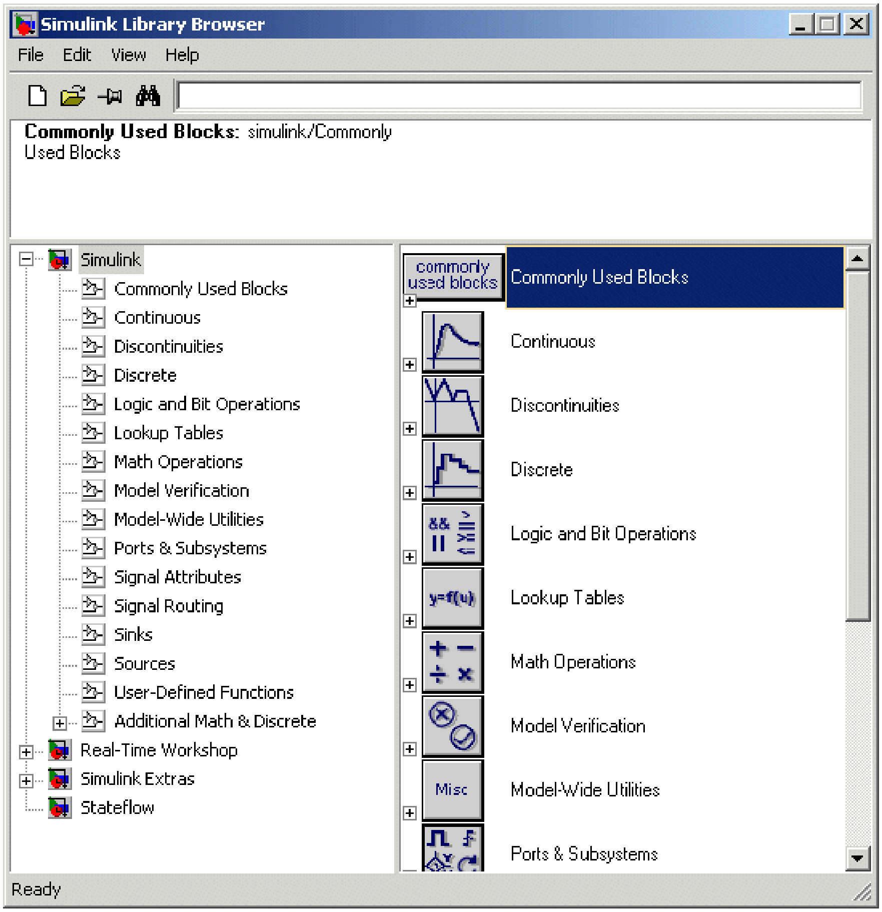
The following sections explain how to perform basic Simulink tasks.

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.

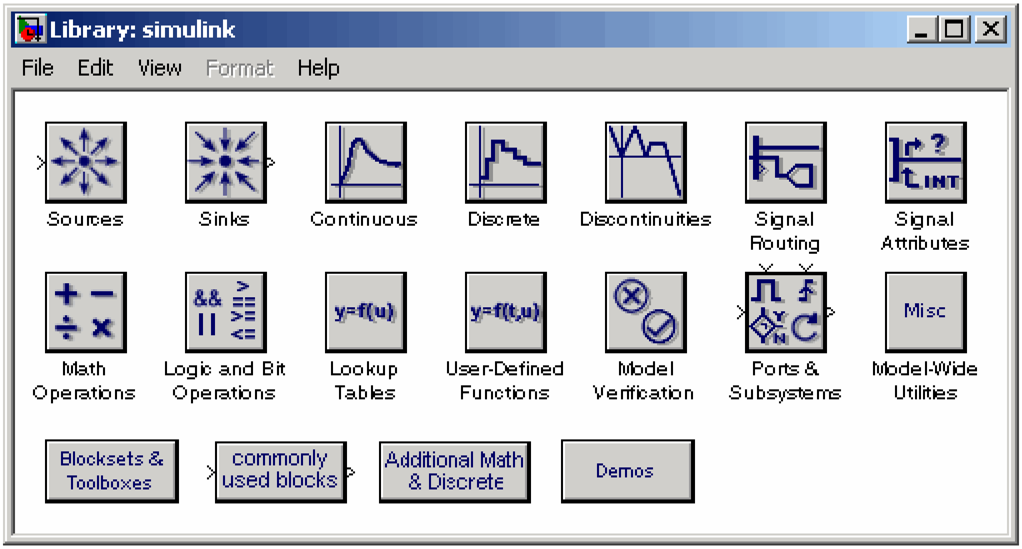


Commonly Used Blocks: simulink/Commonly Used Blocks

Ready

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 4-4).

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



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

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

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 you have an earlier version of Simulink, and you want to open a model that was created in a later version of Simulink, you must first use the later version of Simulink to save the model in a format compatible with the earlier version of Simulink. You can then open the model in the earlier version of Simulink. See "Saving a Model in Earlier Formats" on page 2-17 for details.

Opening Models with Different Character Encodings

If you open a model created in a MATLAB session configured to support one character set encoding, for example, Shift_JIS, in a MATLAB session configured to support another character encoding, for example, US_ASCII, Simulink displays a warning or an error message, depending on whether it can or cannot encode the model, using the current character encoding, respectively. The warning or error message specifies the encoding of the current session and the encoding used to create the model. To avoid corrupting the model (see "Saving Models with Different Character Encodings" on page 2-17) and ensure correct display of the model's text, you should:

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

You can now safely edit and save the model.

Avoiding Initial Model Open Delay

You may notice that the first model that you open in a MATLAB 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:

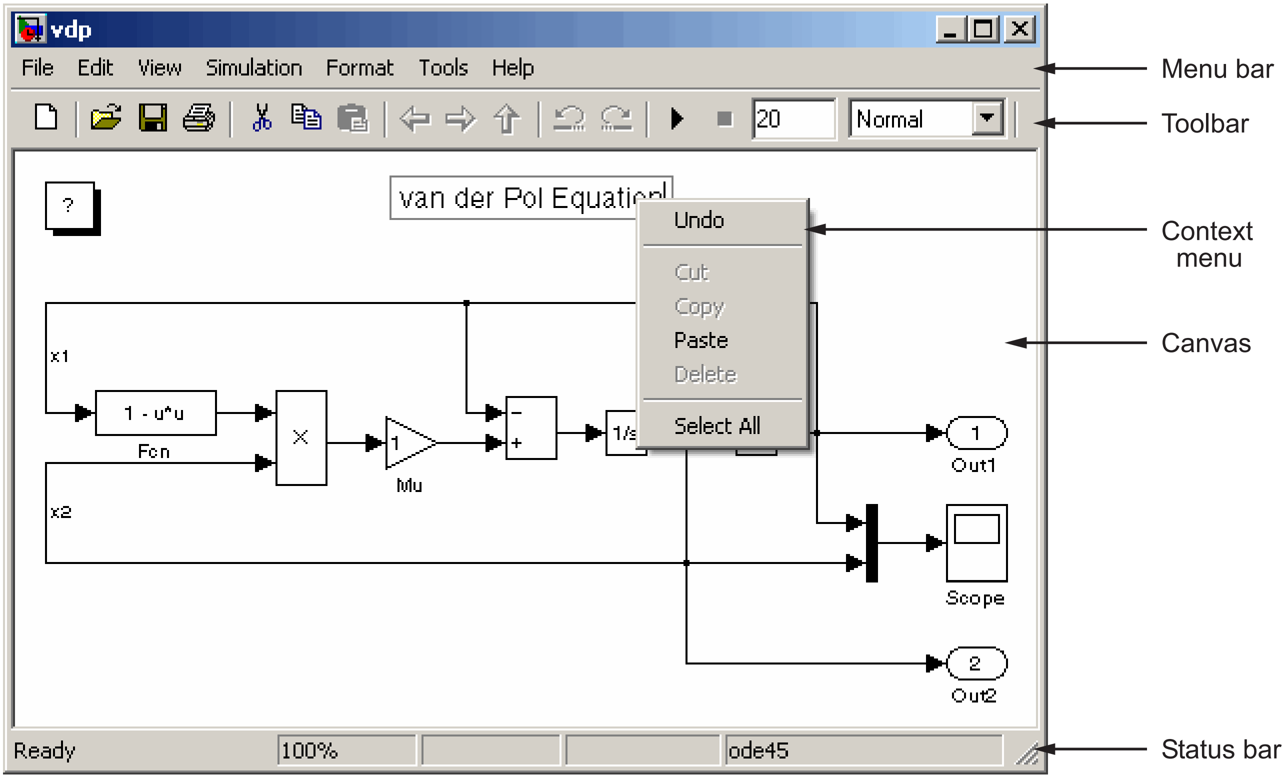
```
matlabroot\bin\win32\matlab.exe -r load_simulink
```

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

```
matlab -r load_simulink
```

Model Editor

When you open a Simulink model or library, Simulink displays the model or library in an instance of the Model Editor.



- “Editor Components” on page 2-8
- “Undoing a Command” on page 2-9
- “Zooming Block Diagrams” on page 2-9
- “Panning Block Diagrams” on page 2-10
- “View Command History” on page 2-11
- “Bringing the MATLAB Desktop Forward” on page 2-12

- "Copying Models to Third-Party Applications" on page 2-12

Editor Components

The Model Editor includes the following components.

Menu Bar

The Simulink menu bar contains commands for creating, editing, viewing, printing, and simulating models. The menu commands apply to the model displayed in the editor. See Chapter 3, "Creating a Model" and Chapter 11, "Running Simulations" for more information.

Toolbar

The toolbar allows you to execute Simulink's most frequently used Simulink commands with a click of a mouse button. For example, to open a Simulink model, click the open folder icon on the toolbar. Letting the mouse cursor hover over a toolbar button or control causes a tooltip to appear. The tooltip describes the purpose of the button or control. You can hide the toolbar by clearing the **Toolbar** option on the Simulink **View** menu.

Canvas

The canvas displays the model's block diagram. The canvas allows you to edit the block diagram. You can use your system's mouse and keyboard to create and connect blocks, select and move blocks, edit block labels, display block dialog boxes, and so on. See Chapter 4, "Working with Blocks" for more information.

Context Menus

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

Status Bar

The status bar appears only in the Windows version of the Model Editor. 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. Regardless of the simulation state, the status bar also displays the zoom factor of the model editor window expressed as a percentage of normal (100%). You can display or hide the status bar by selecting or clearing the **Status Bar** option on the Simulink **View** menu.

Undoing a Command

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

- Adding, deleting, or moving a block
- Adding, deleting, or moving a line
- Adding, deleting, or moving a model annotation
- Editing a block name
- Creating a subsystem (see "Undoing Subsystem Creation" on page 3-35 for more information)

You can reverse the effects of an **Undo** command by choosing **Redo** from the **Edit** 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 (or type **1**) to view the diagram at actual size.

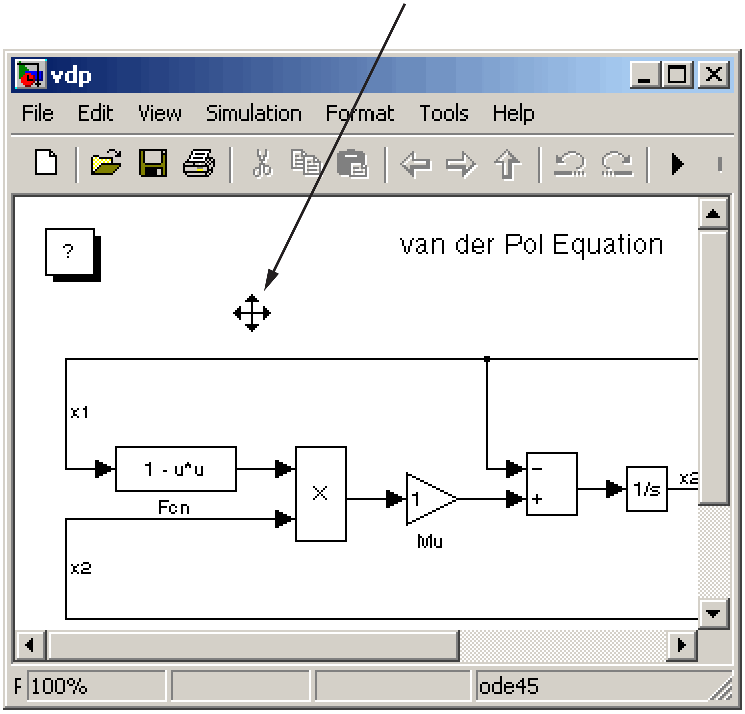
By default, Simulink fits a block diagram to view when you open the diagram either in the model browser's content pane or in a separate window. If you change a diagram's zoom setting, Simulink saves the setting when you close the diagram and restores the setting the next time you open the diagram. If 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 your keyboard alone (see "Model Viewing Shortcuts" on page 2-32) or in combination with your mouse to pan model diagrams that are too large to fit in the Model Editor's window. To use the keyboard and the mouse, position the mouse over the diagram, hold down the **p** or **q** key on the keyboard, then hold down the left mouse button.

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

A pan cursor appears.



Pan cursor

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

View Command History

Simulink maintains a history of the modeling viewing commands, i.e., pan and zoom, that you execute for each model window. The history allows you to quickly return to a previous view in a window, using the following commands, accessible from the Model Editor's **View** menu and tool bar:

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

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

Bringing the MATLAB Desktop Forward

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

Copying Models to Third-Party Applications

On a Microsoft Windows system only, you can copy a Simulink model to the Windows clipboard, then paste it to a third-party application such as Microsoft Word. Simulink allows you to copy a model in either bitmap or metafile format. You can then paste the clipboard model to an application that accepts figures in bitmap or metafile format. See "Exporting to the Windows Clipboard" in the MATLAB Graphics documentation for a description of how to set up the figure settings and save a figure to the clipboard.

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

1 Set the figure copying options.

   a In MATLAB, select **File > Preferences**. The Preferences dialog box appears.

   b Under the **Figure Copy Template** node, select **Copy Options**.

   c In the Clipboard format pane on the right, select **Preserve information (metafile if possible)**.

   With this setting, MATLAB selects the figure format for you, and uses the metafile format whenever possible.

   d. Click **OK**.

2. Click **OK**.
3. Open the `vdp` model.
4. In the Model Editor, select **Edit > Copy Model to Clipboard**.
5. Open a document in Microsoft Word and paste the contents of the clipboard.

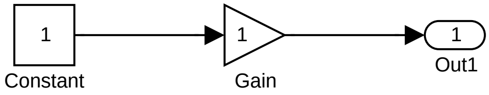
Updating a Block Diagram

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

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

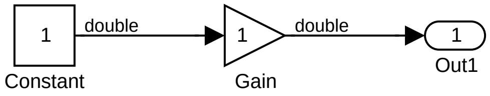
For example:

1 Create the following model.



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

Simulink displays the data types of the output ports of the Constant and Gain blocks. Note that the data type of both ports is `double`, the default value.

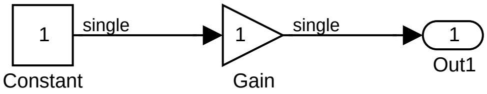


3 Set the **Signal Data Type** parameter of the Constant block (see Constant) to `single`.

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

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

Simulink updates the block diagram to reflect the change that you made previously.



Note that Simulink has inferred a data type for the output of the Gain block. This is because you did not specify a data type for the block. The data type inferred by Simulink is `single` because single precision is all that is necessary to simulate the model accurately, given that the precision of the block's input is single.

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 2-17).

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 Models with Different Character Encodings

When Simulink saves a model, it uses the character encoding in effect when the model was created (the original encoding) to encode the text stored in the model's `.mdl` file, regardless of the character encoding in effect when the model is saved. This can lead to model corruption if you save a model whose original encoding differs from encoding currently in effect in the MATLAB session.

For example, it's possible you could have introduced characters that cannot be represented in the model's original encoding. If this is the case, Simulink saves the model as **`model`**`.err` where **`model`** is the model's name, leaving the original model file unchanged. Simulink also displays an error message that specifies the line and column number of the first unrepresentable character. To recover from this error without losing all the changes you've made to the model in the current session, use the following procedure. First, use a text editor to find the character in the `.err` file at the position specified by the save error message. Then, returning to Simulink, find and delete the corresponding character in the open model and resave the model . Repeat this process until you are able to save the model without error.

It's possible that your model's original encoding can represent all the text changes that you've made in the current session, albeit incorrectly. For example, suppose you open a model whose original encoding is A in a MATLAB session whose current encoding is B. Further, suppose you edit the model to include a character that has different encodings in A and B and then save the model. For example, suppose that the encoding for x in B is the same as the coding for y in A and you insert x in the model while B is in effect, save the model, and then reopen the model with A in effect. In this scenario, Simulink will display x as y. To alert you to the possibility of such corruptions, Simulink displays a warning message when you save a model and the current and original encoding differ but the original encoding can encode, possibly incorrectly, all the characters to be saved in the model file.

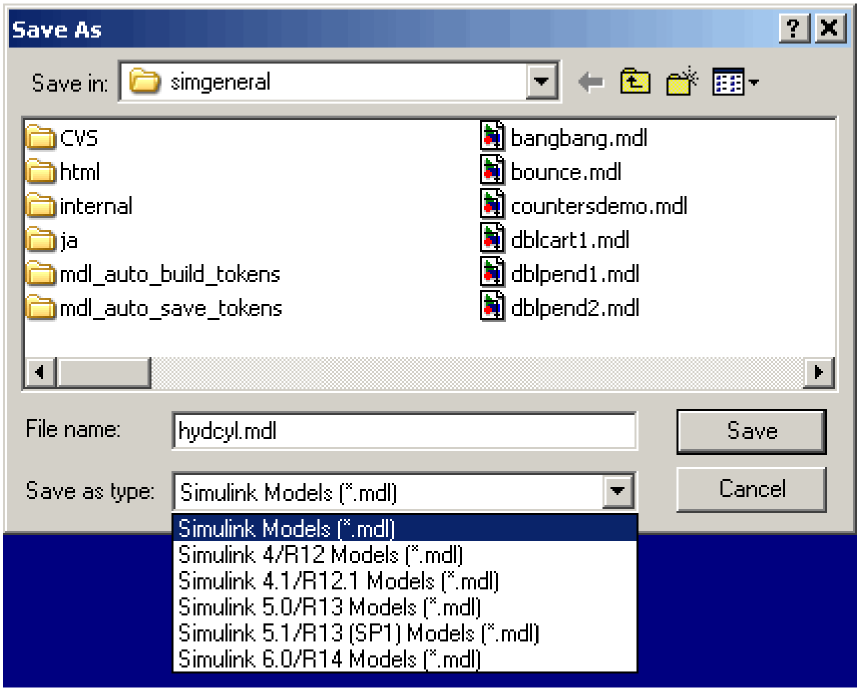
Saving a Model in Earlier Formats

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 4 (Release 12), Simulink 4.1 (Release 12.1), Simulink 5 (Release 13), Simulink 5.1 (Release 13SP1), and Simulink 6 (Release 14, compatible with Release 14, Release 14SP1, and Release 14SP2). 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 As

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 or by using the `print` command in the MATLAB Command Window.

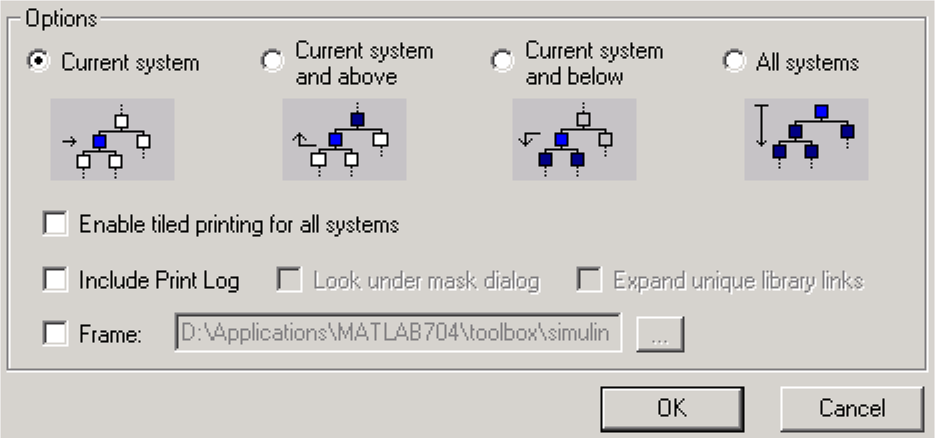
- "Print Dialog Box" on page 2-20
- "Specifying Paper Size and Orientation" on page 2-22
- "Positioning and Sizing a Diagram" on page 2-22
- "Tiled Printing" on page 2-23
- "Print Command" on page 2-27

Print Dialog Box

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

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

The portion of the **Print** dialog box that supports selective printing is similar on supported platforms. This figure shows how it looks on a 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

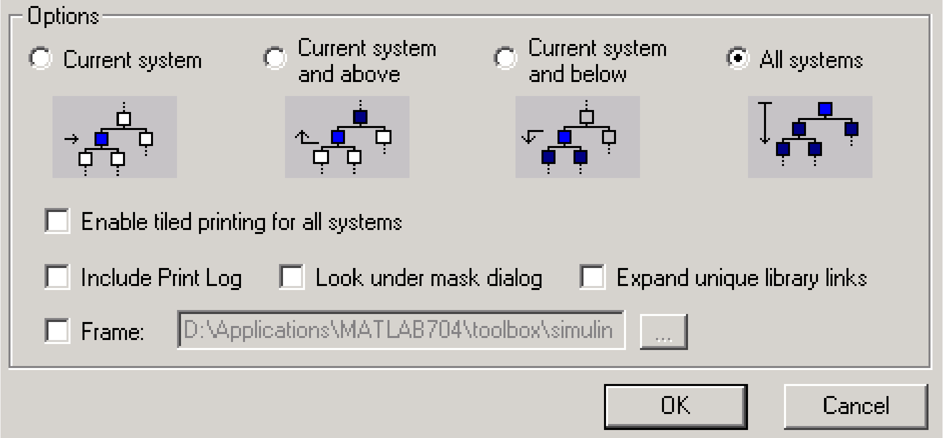
□ Enable tiled printing for all systems

□ Include Print Log □ Look under mask dialog □ Expand unique library links

□ Frame: D:\Applications\MATLAB704\toolbox\simulin ...

OK Cancel

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

□ Enable tiled printing for all systems

□ Include Print Log □ Look under mask dialog □ Expand unique library links

□ Frame: D:\Applications\MATLAB704\toolbox\simulin ...

OK Cancel

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 4-35.

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

Selecting the **Enable tiled printing for all systems** check box overrides the tiled-print settings for individual subsystems in a model. See "Tiled Printing" on page 2-23 for more information.

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

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 Simulink reference), 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.

Tiled Printing

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

By contrast, tiled printing enables you to print even the largest block diagrams without sacrificing clarity and detail. Tiled printing allows you to distribute a block diagram over multiple pages. You can control the number of pages over which Simulink prints the block diagram, and hence, the total size of the printed diagram.

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

Enabling Tiled Printing

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

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

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

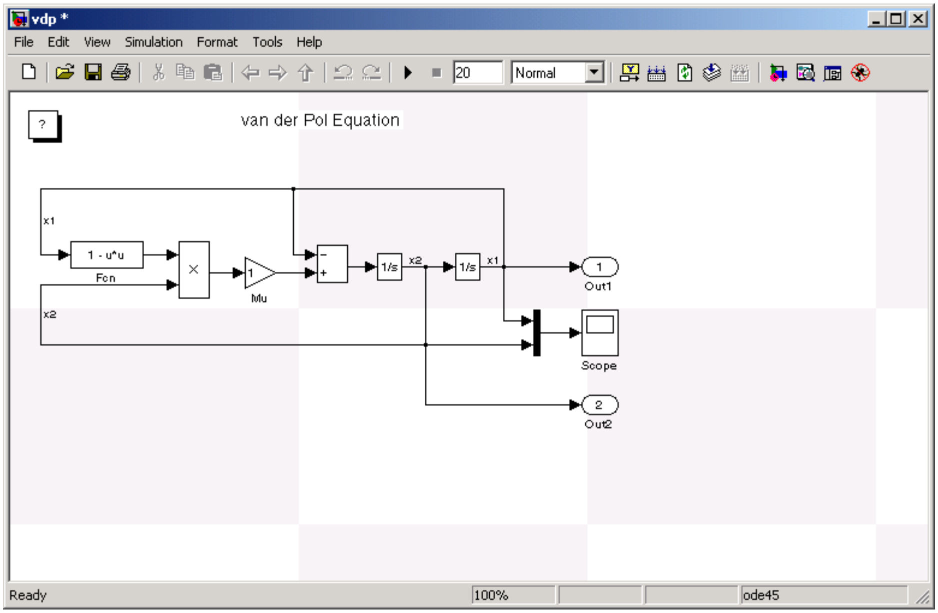
open the `f14` demo model and enable tiled printing for the `Controller` subsystem.

To enable tiled printing for all systems in your model, select the **Enable tiled printing for all systems** check box on the **Print** dialog box (see "Print Dialog Box" on page 2-20). If you select this option, Simulink overrides the individual tiled-print settings for all systems in your model.

Displaying Page Boundaries

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

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



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

Specifying Tiled Print Settings

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

The `TiledPageScale` parameter scales the block diagram so that more or less of it appears on a single tiled page. By default, its value is `1`. Values greater than `1` proportionally scale the diagram such that it occupies a smaller percentage of the tiled page, while values between `0` and `1` proportionally scale the diagram such that it occupies a larger percentage of the tiled page. For example, a `TiledPageScale` of `0.5` makes the printed diagram appear twice its size on a tiled page, while a `TiledPageScale` of `2` makes the printed diagram appear half its size on a tiled page.

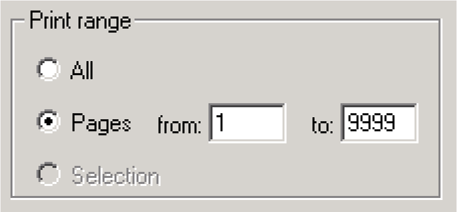
Simulink lets you specify the margin sizes associated with tiled pages using the `TiledPaperMargins` parameter. The value of `TiledPaperMargins` is a vector of form `[left top right bottom]`. Each element specifies the size of the margin at a particular edge of the page. Simulink uses the value of the `PaperUnits` parameter to determine its units of measurement. By default, Simulink sets each margin to 0.5 inches. By decreasing the margin sizes, you can increase the printable area of the tiled pages.

Printing Tiled Pages

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

Alternatively, you can specify the range of tiled page numbers that Simulink prints as follows:

- On a Microsoft Windows system, you can specify a range of tiled page numbers to be printed using the **Print range** portion of the **Print** dialog box. This field is accessible if you select both the **Current system** and **Enable tiled printing for all systems** options (see "Print Dialog Box" on page 2-20).



Print range

○ All

● Pages from: 1 to: 9999

○ Selection

- On all platforms, you can specify a range of tiled page numbers to be printed using the `print` command at the MATLAB prompt. The print command's `tileall` option enables tiled printing for the system, and its `pages` option indicates the range of tiled page numbers to be printed (see "Print Command" on page 2-27). For example, the following commands

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

  open the `vdp` demo model, enable tiled printing, display the page boundaries, and scale the tiled pages such that the block diagram spans multiple pages. You can print the second, third, and fourth pages by issuing the following command at the MATLAB prompt:

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

Note Simulink uses a row-major scheme to number tiled pages. For example, the first page of the first row is 1, the second page of the first row is 2, etc.

Print Command

The format of the `print` command is

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

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

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

`tileall` specifies the tiled printing option (see "Tiled Printing" on page 2-23).

`p` is a two-element vector specifying the range of tiled page numbers to be printed. The vector must be preceded by the `pages` switch identifier. This option is valid only when you enable tiled printing using the `tileall` switch. For an example of its usage, see "Printing Tiled Pages" on page 2-26.

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

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

```
print -suntitled
```

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

```
print -sSub1
```

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

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

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

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

To print the currently selected subsystem, enter

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

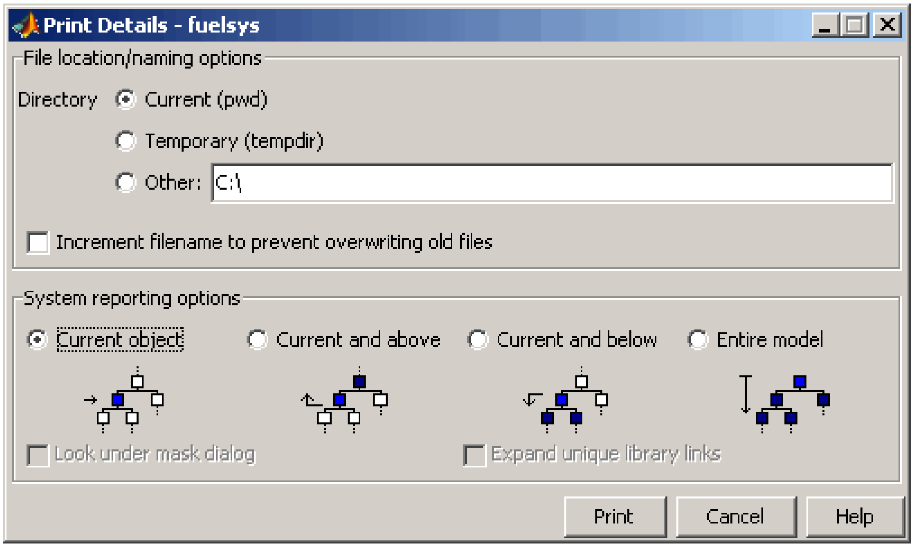
Generating a Model Report

A 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.



Print Details - fuelsys

File location/naming options

Directory ● Current (pwd)

○ Temporary (tempdir)

○ Other: C:\

□ 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

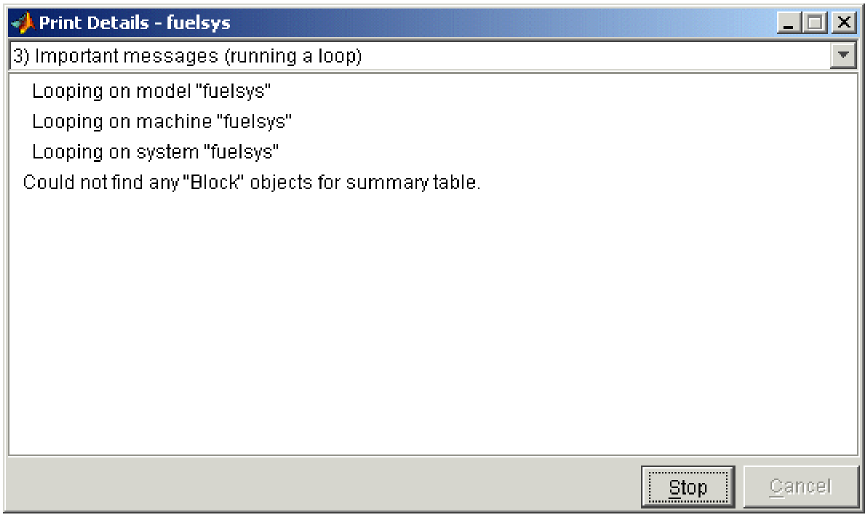
The dialog box allows you to select various report options (see "Model Report Options" on page 2-30).

2 Select the desired report options on the dialog box.

3 Select **Print**.

Simulink generates the HTML report and displays the report 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

Simulink provides mouse and keyboard shortcuts for many of its commands. The following tables summarize these shortcuts.

- "Model Viewing Shortcuts" on page 2-32
- "Block Editing Shortcuts" on page 2-33
- "Line Editing Shortcuts" on page 2-34
- "Signal Label Editing Shortcuts" on page 2-34
- "Annotation Editing Shortcuts" on page 2-35

LMB means press the left mouse button; CMB, the center mouse button; and RMB, the right mouse button.

Model Viewing Shortcuts

The following table lists keyboard shortcuts for viewing models.

Task	Microsoft Windows	UNIX
Zoom in	**r**	**r**
Zoom out	**v**	**v**
Zoom to normal (100%)	**1**	**1**
Pan left	**d** or **Ctrl+Left Arrow**	**d** or **Ctrl+Left Arrow**
Pan right	**g** or **Ctrl+Right Arrow**	**g** or **Ctrl+Right Arrow**
Pan up	**e** or **Ctrl+Up Arrow**	**e** or **Ctrl+Up Arrow**
Pan down	**c** or **Ctrl+Down Arrow**	**c** or **Ctrl+Down Arrow**
Fit selection to screen	**f**	**f**
Fit diagram to screen	**Space**	**Space**
Pan with mouse	Hold down **p** or **q** and drag mouse	Hold down **p** or **q** and drag mouse

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

Block Editing Shortcuts

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

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

Line Editing Shortcuts

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

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

Signal Label Editing Shortcuts

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

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

Annotation Editing Shortcuts

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

Action	Microsoft Windows	UNIX
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 UNIX system: **Quit MATLAB**

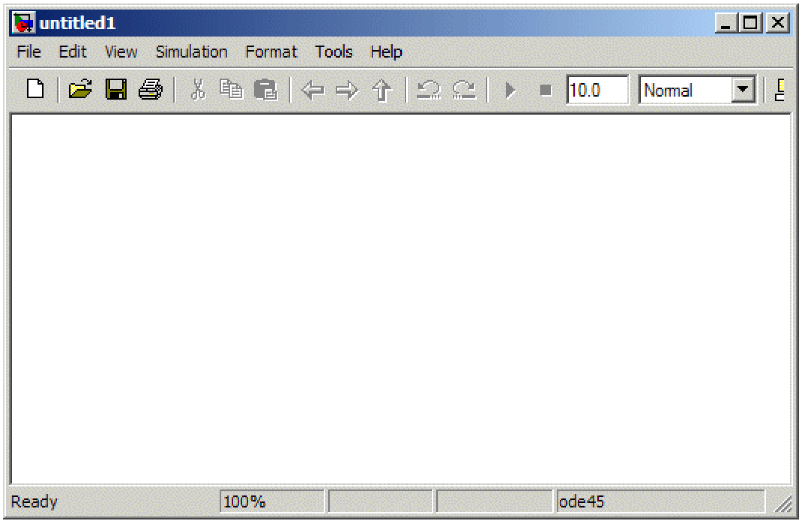
3

Creating a Model

The following sections explain how to perform tasks required to create Simulink models.

Creating an Empty Model

To create an empty 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**. Simulink creates an empty model in memory and displays it in a new model editor window.



Creating a Model Template

When creating a model, Simulink uses defaults for many of its configuration parameters. For instance, by default, new models have a white canvas, the `ode45` solver, and a visible toolbar. If these defaults are not to your liking, use Simulink model construction commands (see "Model Construction" in the online Simulink reference) to write a function that creates a model with the defaults you prefer.

For example, the following function creates a Simulink model that has a green canvas and a hidden toolbar and uses the `ode3` solver:

```
function new_model(modelname)
% NEW_MODEL Create a new, empty Simulink model
%    NEW_MODEL('MODELNAME') creates a new model with
%    the name 'MODELNAME'. Without the 'MODELNAME'
%    argument, the new model is named 'my_untitled'.

if nargin == 0
     modelname = 'my_untitled';
end

% create and open the model
open_system(new_system(modelname));

% set default screen color
set_param(modelname, 'ScreenColor', 'green');

% set default solver
set_param(modelname, 'Solver', 'ode3');

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

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

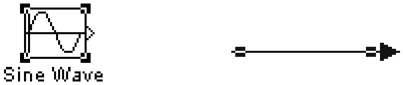
Selecting Objects

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). The following sections describe selecting objects:

- "Selecting an Object" on page 3-5
- "Selecting Multiple Objects" on page 3-5

Selecting an Object

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



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

Selecting Multiple Objects

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

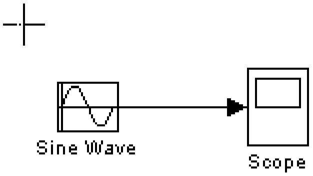
Selecting Multiple Objects One at a Time

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

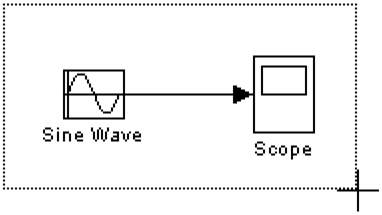
Selecting Multiple Objects Using a Bounding Box

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

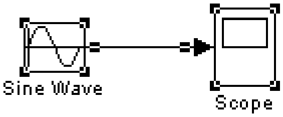
1 Define the starting corner of a bounding box by positioning the pointer at one corner of the box, then pressing and holding down the mouse button. Notice the shape of the cursor.



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



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



Selecting All Objects

To select all objects in the active window, select **Select All** from the **Edit** menu. You cannot create a subsystem by selecting blocks and lines in this way. For more information, see “Creating Subsystems” on page 3-33.

Specifying Block Diagram Colors

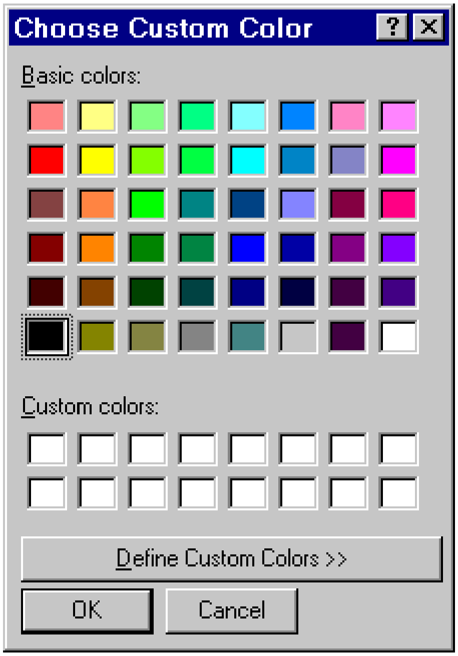
Simulink allows you to specify the foreground and background colors of any block or annotation in a diagram, as well as the diagram's background color. To set the background color of a block diagram, select **Screen color** from the Simulink **Format** menu. To set the background color of a block or 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. For more information on specifying foreground and background colors, see:

- "Choosing a Custom Color" on page 3-8
- "Defining a Custom Color" on page 3-8
- "Specifying Colors Programmatically" on page 3-9
- "Displaying Sample Time Colors" on page 3-10

Choosing a Custom Color

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

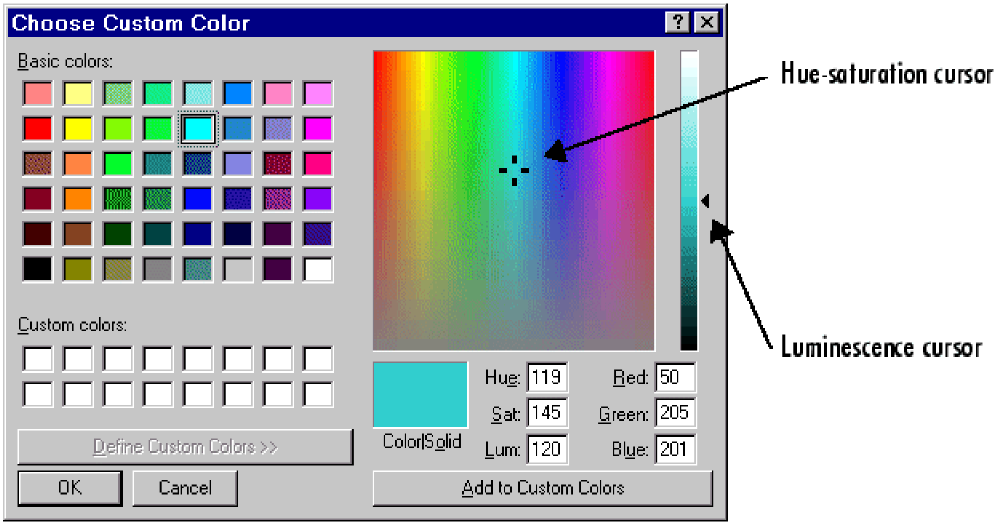


The dialog box displays a palette of basic colors and a palette of custom colors that you previously defined. If you have not previously created any custom colors, the custom color palette is all white. To choose a color from either palette, click the color, and then click the **OK** button.

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 sample time
Magenta	Constant sample time
Red	Fastest discrete sample time
Green	Second fastest discrete sample time
Blue	Third fastest discrete sample time
Light Blue	Fourth fastest discrete sample time

Color	Use
Dark Green	Fifth fastest discrete sample time
Orange	Sixth fastest discrete sample time
Yellow	Can indicate one of the following: • A block with hybrid sample time, e.g., subsystems grouping blocks and Mux or Demux blocks grouping signals with different sample times, Data Store Memory blocks updated and read by different tasks. • Variable sample time. See the Pulse Generator block and "Specifying Sample Time" on page 1-36 for more information. • A block with the seventh, eighth, etc., 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 top-level model and in the models that it references (see "Referencing Models" on page 3-63). 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 top-level 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 Chapter 5, “Working with Signals” 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.

For more information on connecting blocks, see:

- “Automatically Connecting Blocks” on page 3-13
- “Manually Connecting Blocks” on page 3-16
- “Disconnecting Blocks” on page 3-21

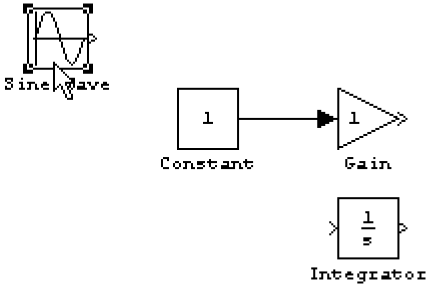
Automatically Connecting Blocks

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

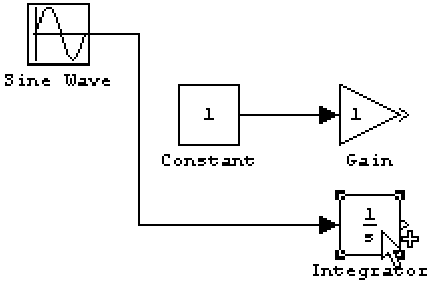
To autoconnect two blocks:

1 Select the source block.

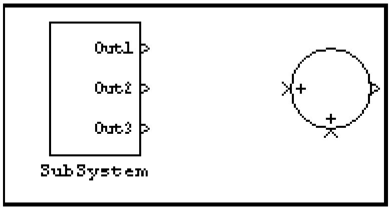


2 Hold down **Ctrl** and left-click the destination block.

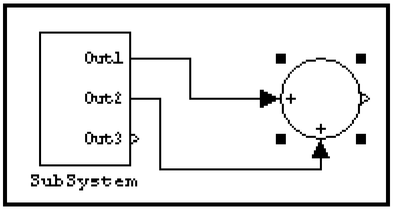
Simulink connects the source block to the destination block, routing the line around intervening blocks if necessary.



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



Before autoconnect



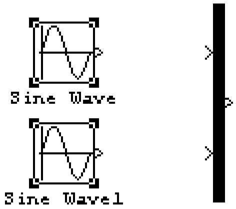
After autoconnect

Connecting Groups of Blocks

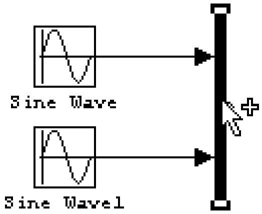
Simulink can connect a group of source blocks to a destination block or a source block to a group of destination blocks.

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

1 Select the source blocks.

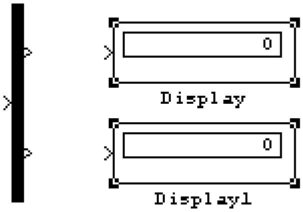


2 Hold down **Ctrl** and left-click the destination block.

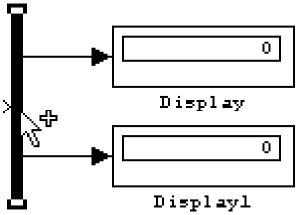


To connect a source block to a group of destination blocks:

1 Select the *destination* blocks.



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



Manually Connecting Blocks

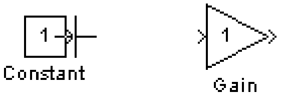
Simulink allows you to draw lines manually between blocks or between lines and blocks. You might want to do this if you need to control the path of the line or to create a branch line.

Drawing a Line Between Blocks

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

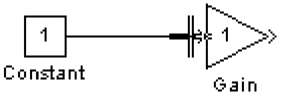
1. Position the cursor over the first block's output port. It is not necessary to position the cursor precisely on the port.

   The cursor shape changes to crosshairs.



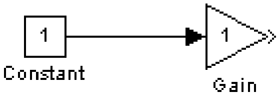
2. Press and hold down the mouse button.
3. Drag the pointer to the second block's input port. You can position the cursor on or near the port or in the block. If you position the cursor in the block, the line is connected to the closest input port.

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

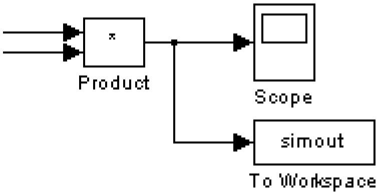


Simulink draws connecting lines using horizontal and vertical line segments. To draw a diagonal line, hold down the **Shift** key while drawing the line.

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 represent the same signal. Using branch lines enables you to connect a signal to more than one block.

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



To add a branch line:

1 Position the pointer on the line where you want the branch line to start.

2 While holding down the **Ctrl** key, press and hold down the left mouse button.

3 Drag the pointer to the input port of the target block, then release the mouse button and the **Ctrl** key.

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

Drawing a Line Segment

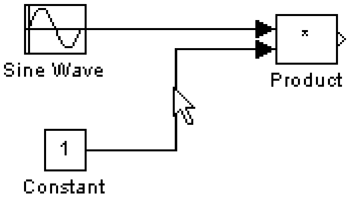
You might want to draw a line with segments exactly where you want them instead of where Simulink draws them. Or you might want to draw a line before you copy the block to which the line is connected. You can do either by drawing line segments.

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.

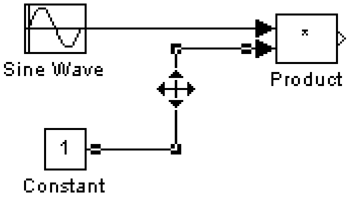
Moving a Line Segment

To move a line segment:

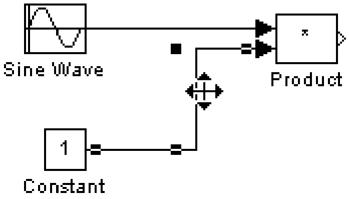
1 Position the pointer on the segment you want to move.



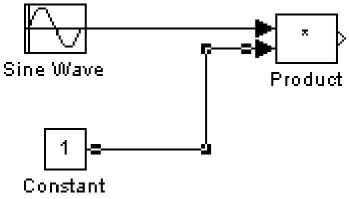
2 Press and hold down the left mouse button.



3 Drag the pointer to the desired location.



4 Release the mouse button.



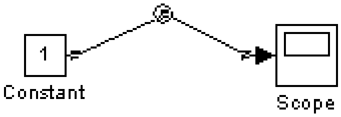
To move the segment connected to an input port, position the pointer over the port and drag the end of the segment to the new location. You cannot move the segment connected to an output port.

Moving a Line Vertex

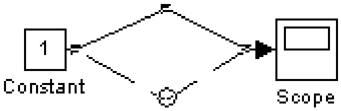
To move a vertex of a line:

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

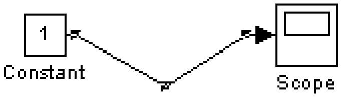
The cursor changes to a circle that encloses the vertex.



2 Drag the pointer to the desired location.



3 Release the mouse button.

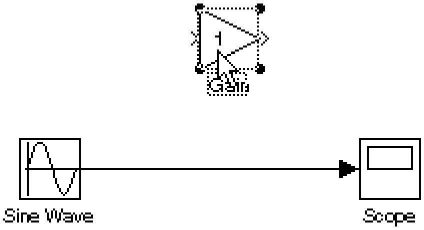


Inserting Blocks in a Line

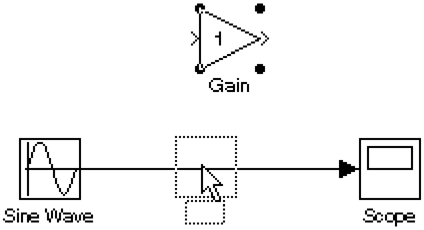
You can insert a block in a line by dropping the block on the line. Simulink inserts the block for you at the point where you drop the block. The block that you insert can have only one input and one output.

To insert a block in a line:

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

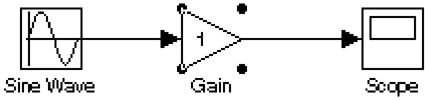


2 Drag the block over the line in which you want to insert the block.



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

Simulink inserts the block where you dropped it.



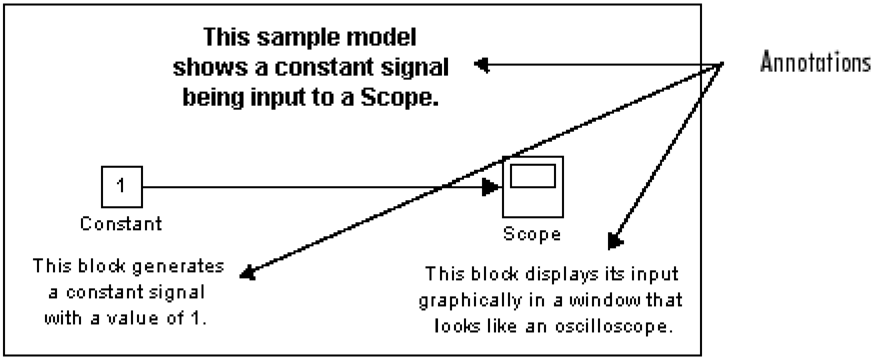
Disconnecting Blocks

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

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

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 in the rectangle that surrounds the annotation.

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

To edit an annotation, select it:

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

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

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

The following topics provide more information on annotations:

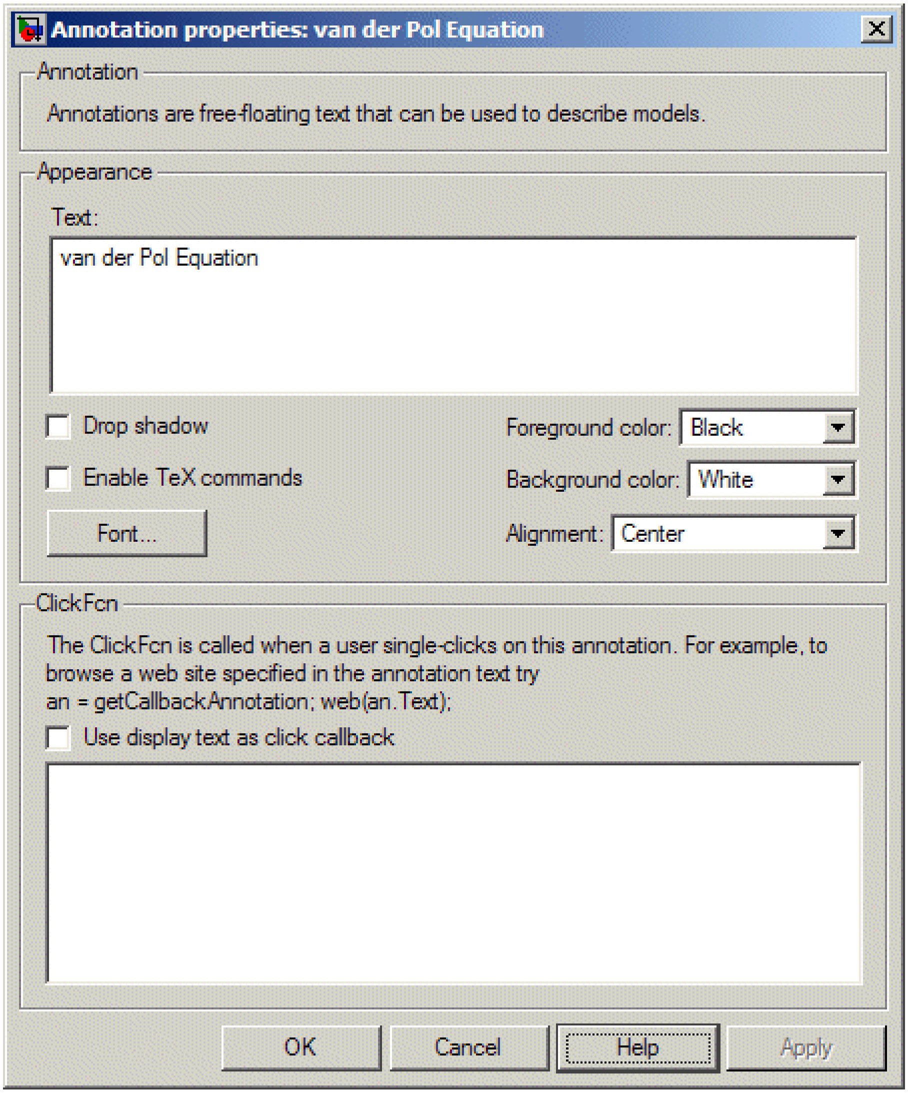
- "Annotations Properties Dialog Box" on page 3-23
- "Annotation Callback Functions" on page 3-26
- "Associating Click Functions with Annotations" on page 3-27
- "Annotations API" on page 3-29
- "Using TeX Formatting Commands in Annotations" on page 3-29
- "Creating Annotations Programmatically" on page 3-31

Annotations Properties Dialog Box

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

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

The dialog box appears.



Annotation properties: van der Pol Equation

Annotation

Annotations are free-floating text that can be used to describe models.

Appearance

Text:

van der Pol Equation

□ Drop shadow

Foreground color: Black

□ Enable TeX commands

Background color: White

Alignment: Center

ClickFcn

The ClickFcn is called when a user single-clicks on this annotation. For example, to browse a web site specified in the annotation text try
an = getCallbackAnnotation; web(an.Text);

□ Use display text as click callback

The dialog box includes the following controls.

Text

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

Drop shadow

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

Enable TeX commands

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

Font

Clicking this button displays a font chooser dialog box. Use the font chooser to change the font used to render the annotation’s text.

Foreground color

Specifies the color of the annotation text.

Background color

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

Alignment

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

ClickFcn

Specifies MATLAB code to be executed when a user single-clicks this annotation. Simulink stores the code entered in this field with the model. See “Associating Click Functions with Annotations” on page 3-27 for more information.

Use display text as click callback

Checking this option causes Simulink to treat the text in the **Text** field as the annotation's click function. The specified text must be a valid MATLAB expression comprising symbols that are defined in the MATLAB workspace when the user clicks this annotation. See "Associating Click Functions with Annotations" on page 3-27 for more information. Note that selecting this option disables the **ClickFcn** edit field.

Annotation Callback Functions

Simulink allows you to associate the following callback functions with annotations.

Click Function

A click function is an M function that Simulink invokes when a user single-clicks an annotation. Simulink allows you to associate a click function with any of a model's annotations (see "Associating Click Functions with Annotations" on page 3-27). Simulink uses the color blue to display the text of annotations associated with click functions. This allows the user to see at a glance which annotations are associated with click functions. Click functions allow you to add hyperlinks and custom command "buttons" to your model's block diagram. For example, you can use click functions to allow a user to display the values of workspace variables referenced by the model or to open related models simply by clicking on annotations displayed on the block diagram.

Load Function

Simulink invokes this function when it loads the model containing the associated annotation. To associate a load function with an annotation, set the LoadFcn property of the annotation to the desired function (see "Annotations API" on page 3-29).

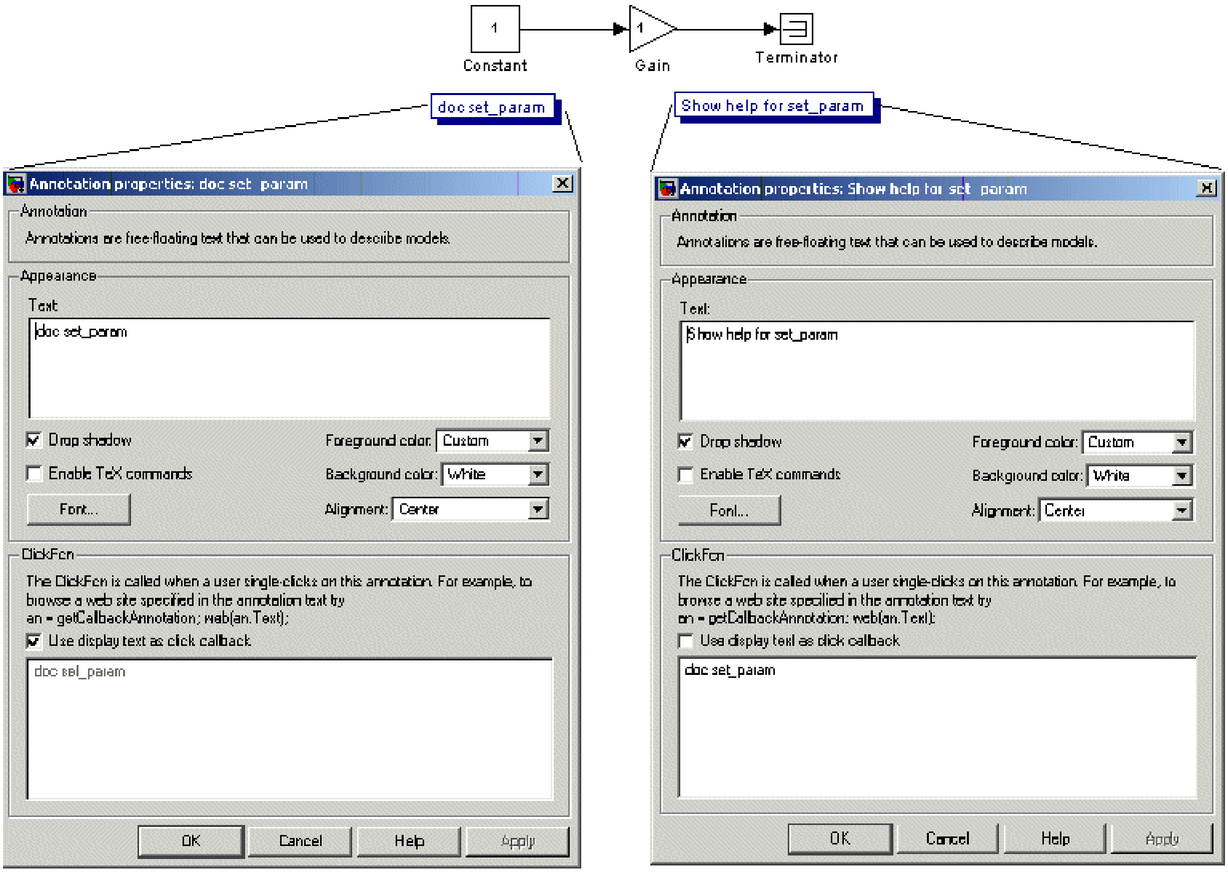
Delete function

Simulink invokes this function before deleting the associated annotation. To associate a delete function with an annotation, set the DeleteFcn property of the annotation to the desired function (see "Annotations API" on page 3-29).

Associating Click Functions with Annotations

Simulink provides two ways to associate a click function with an annotation via the annotation's properties dialog box (see "Annotations Properties Dialog Box" on page 3-23). You can specify either the annotation itself as the click function or a separately defined function. To specify the annotation itself as the click function, enter a valid MATLAB expression in the dialog box's **Text** field and check the **Use display text** as callback option. To specify a separately defined click function, enter the M-code that defines the click function in the dialog box's **ClickFcn** edit field.

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



Annotation properties: doc set_param

Annotation
Annotations are free-floating text that can be used to describe models.

Appearance
Text:
doc set_param

☑ Drop shadow — Foreground color: Custom
□ Enable TeX commands — Background color: White
Font... — Alignment: Center

ClickFcn
The ClickFcn is called when a user single-clicks on this annotation. For example, to browse a web site specified in the annotation text by
an = getCallbackAnnotation; web(an.Text);
☑ Use display text as click callback
doc set_param

OK | Cancel | Help | Apply

Annotation text as the click function

Annotation properties: Show help for set_param

Annotation
Annotations are free-floating text that can be used to describe models.

Appearance
Text:
Show help for set_param

☑ Drop shadow — Foreground color: Custom
□ Enable TeX commands — Background color: White
Font... — Alignment: Center

ClickFcn
The ClickFcn is called when a user single-clicks on this annotation. For example, to browse a web site specified in the annotation text by
an = getCallbackAnnotation; web(an.Text);
□ Use display text as click callback
doc set_param

OK | Cancel | Help | Apply

Separately defined click function

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

Note You can also use M-code to associate a click function with an annotation. See “Annotations API” on page 3-29 for more information.

Selecting and Editing Annotations Associated with Click Functions

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

Annotations API

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

- `Simulink.Annotation` class

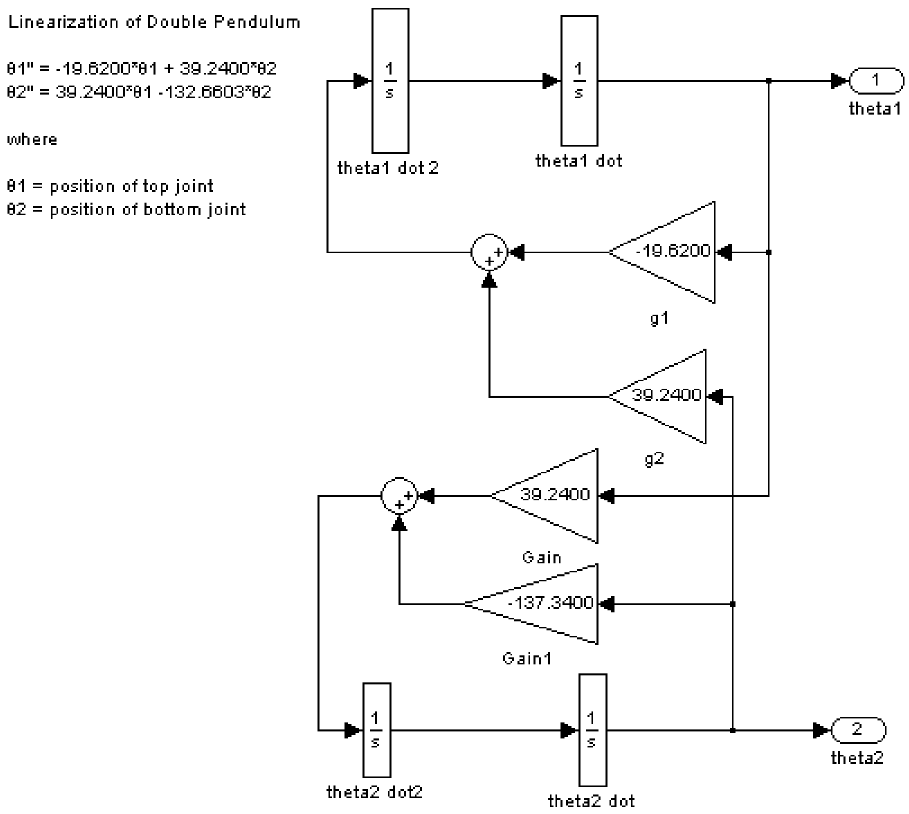
  Allows M-code, e.g., annotation load functions (see “Load Function” on page 3-26), to set the properties of annotations

- `getCallbackAnnotation` function

  Gets the `Simulink.Annotation` object for the annotation associated with the currently executing annotation callback function

Using TeX Formatting Commands in Annotations

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



Linearization of Double Pendulum

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

where

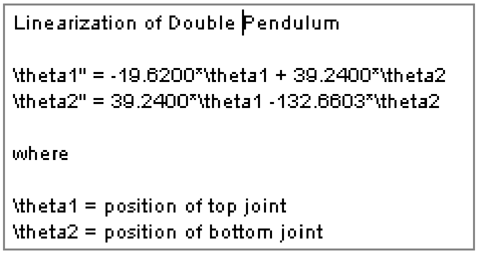
θ1 = position of top joint
θ2 = position of bottom joint

To use TeX commands in an annotation:

1 Select the annotation.

2 Select **Enable TeX Commands** from model editor’s **Format** menu.

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

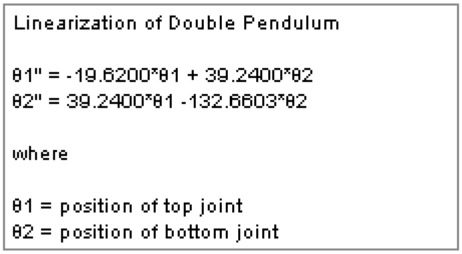


```
Linearization of Double Pendulum

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

where

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

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

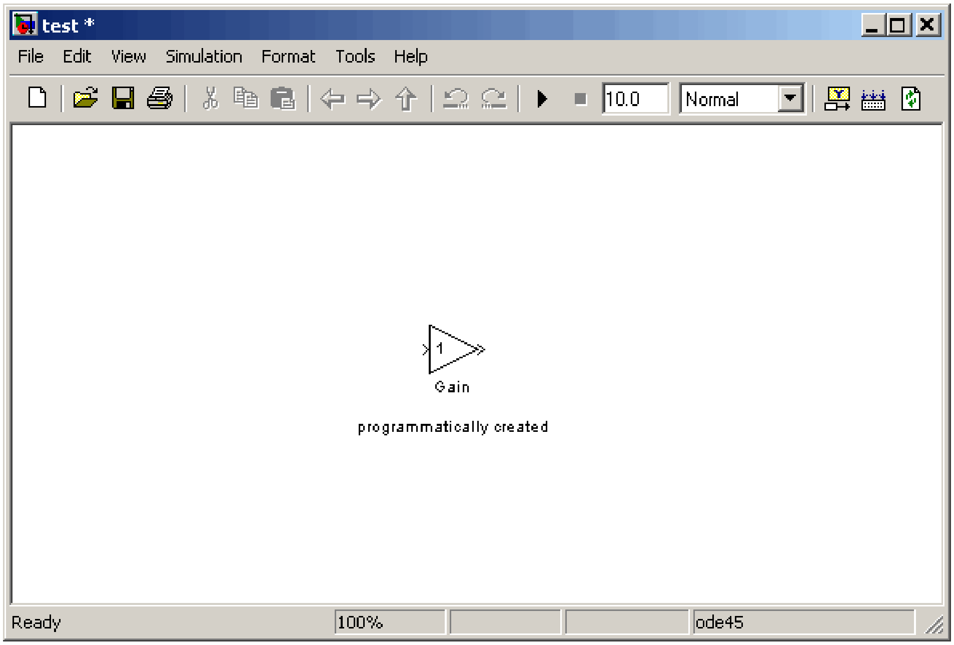
You can use the Simulink `add_block` command to create annotations at the command line or in an M-file program. Use the following syntax to create the annotation:

```
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:



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

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

Creating Subsystems

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.

For more information, see:

- "Creating a Subsystem by Adding the Subsystem Block" on page 3-33
- "Creating a Subsystem by Grouping Existing Blocks" on page 3-34
- "Model Navigation Commands" on page 3-36
- "Window Reuse" on page 3-36
- "Labeling Subsystem Ports" on page 3-37
- "Controlling Access to Subsystems" on page 3-38
- "Interconverting Subsystems and Block Diagrams" on page 3-38
- "Emptying Subsystems and Block Diagrams" on page 3-39

Creating a Subsystem by Adding the Subsystem Block

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

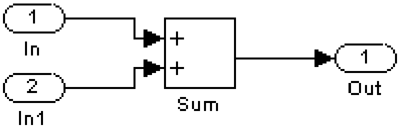
1 Copy the Subsystem block from the Ports & Subsystems library into your model.

2 Open the Subsystem block by double-clicking it.

   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 3-36).

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.

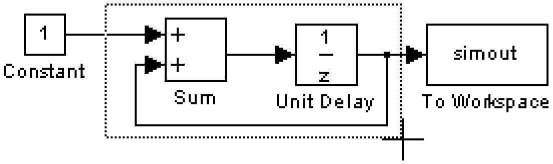


Creating a Subsystem by Grouping Existing Blocks

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 3-5.

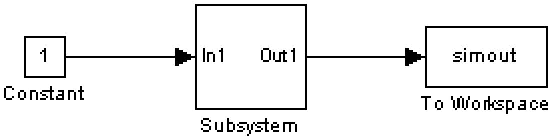
   For example, this figure shows a model that represents a counter. The Sum and Unit Delay blocks are selected within a bounding box.



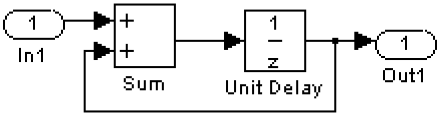
When you release the mouse button, the two blocks and all the connecting lines are selected.

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 13, "Creating Block Masks".

Undoing Subsystem Creation

To undo creation of a subsystem by grouping blocks, select **Undo** from the **Edit** menu. You can undo creation of a subsystem that you have subsequently edited. However, the **Undo** command does not undo any nongraphical changes that you made to the blocks, such as changing the value of a block parameter or the name of a block. 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 10-29) and/or the following model navigation commands:

- **Open Block**

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

- **Open Block In New Window**

  Opens the currently selected subsystem regardless of the Simulink window reuse settings (see "Window Reuse" on page 3-36). To execute the command, select **Open Block In New Window** from the subsystem's context (right-click) menu.

- **Go To Parent**

  The **Go To Parent** command displays the parent of the subsystem displayed in the current window. To execute the command, press **Esc** or select **Go To Parent** from the Simulink **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 Type	Open Action	Go to Parent (Esc) Action
`reuse`	Subsystem replaces the parent in the current window.	Parent window replaces subsystem in current window
`replace`	Subsystem appears in a new window. Parent window disappears.	Parent window appears. Subsystem window disappears.
`mixed`	Subsystem appears in its own window.	Parent window rises to front. Subsystem window disappears.

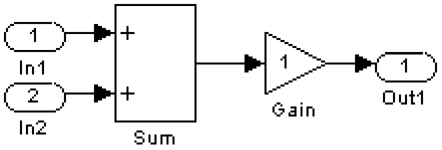
Labeling Subsystem Ports

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

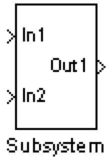
You can hide (or show) the port labels by

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

This figure shows two models.



Subsystem with Inport and Outport blocks



Subsystem with labeled ports

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

Controlling Access to Subsystems

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

To restrict access to a library subsystem, open the subsystem's parameter dialog box and set its **Read/Write permissions** parameter to either `ReadOnly` or `NoReadOrWrite`. The first option allows a user to view the contents of the library subsystem but prevents the user from modifying the reference subsystem without first disabling its library link or changing its **Read/Write permissions** parameter to `ReadWrite`. The second option prevents the user from viewing the contents of the library subsystem, modifying the reference subsystem, and changing the reference subsystem's permissions. Note that both options allow a user to use the library subsystem in models by creating links (see "Working with Block Libraries" on page 4-35). See the Subsystem block in the *Simulink Reference* guide for more information on subsystem access options.

Note If you attempt to view the contents of a subsystem whose **Read/Write permissions** parameter is set to `NoReadOrWrite`, Simulink does not respond. For example, when double-clicking such a subsystem, Simulink neither opens the subsystem nor displays any messages.

Interconverting Subsystems and Block Diagrams

Simulink provides these functions that you can use to interconvert subsystems and block diagrams:

`Simulink.SubSystem.copyContentsToBlockDiagram`
    Copies the contents of a subsystem to an empty block diagram.

`Simulink.BlockDiagram.copyContentsToSubSystem`
    Copies the contents of a block diagram to an empty subsystem.

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

Emptying Subsystems and Block Diagrams

Simulink provides these functions that you can use to empty subsystems and block diagrams:

`Simulink.SubSystem.deleteContents`
: Deletes the contents of a subsystem.

`Simulink.BlockDiagram.deleteContents`
: Deletes the contents of a block diagram.

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

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 3-41.
- 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 3-49.
- 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 3-52 for more information.
- A *control flow subsystem* executes one or more times at the current time step when enabled by a control flow block that implements control logic similar to that expressed by programming language control flow statements (e.g., `if-then`, `while`, `do`, and `for`. See "Modeling Control Flow Logic" on page 3-90 for more information.

Note Simulink imposes restrictions on connecting blocks with a constant sample time to the output port of a conditionally executed subsystem. See "Using Blocks with Constant Sample Times in Enabled Subsystems" on page 3-46 for more information.

For more information on conditionally executed subsystems, see:

- “Enabled Subsystems” on page 3-41
- “Triggered Subsystems” on page 3-49
- “Triggered and Enabled Subsystems” on page 3-52
- “Function-Call Subsystems” on page 3-56
- “Conditional Execution Behavior” on page 3-57

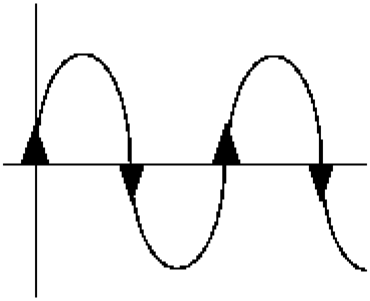
Enabled Subsystems

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.

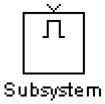
For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled, as shown in this figure. An up arrow signifies enable, a down arrow disable.



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

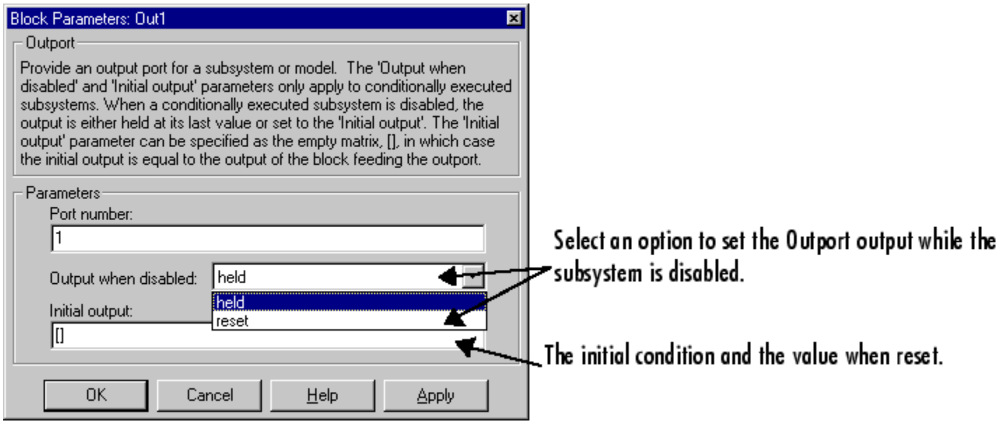
You create an enabled subsystem by copying an Enable block from the Ports & Subsystems library into a subsystem. Simulink adds an enable symbol and an enable control input port to the Subsystem block.



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.



Block Parameters: Out1

Outport

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.

Parameters

Port number: 1

Output when disabled: held

Initial output: []

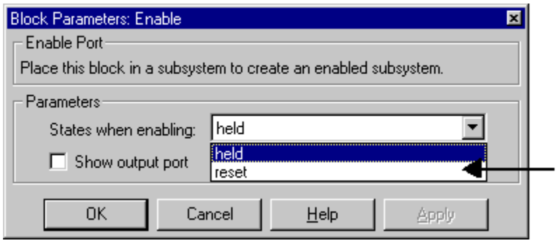
Select an option to set the Outport output while the subsystem is disabled.

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.


Block Parameters: Enable

Enable Port

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

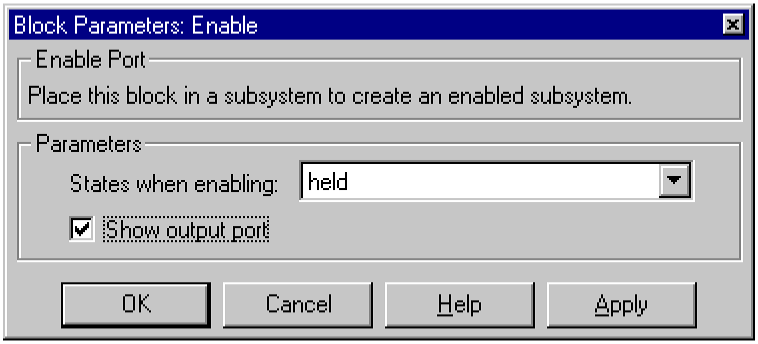
Parameters

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

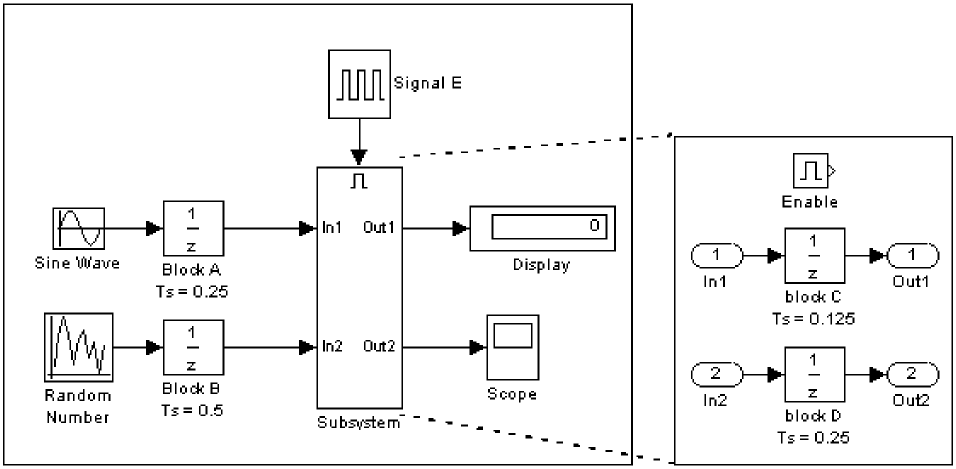
An enabled subsystem can contain any block, whether continuous or discrete. Discrete blocks in an enabled subsystem execute only when the subsystem executes, and only when their sample times are synchronized with the simulation sample time. Enabled subsystems and the model use a common clock.

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.

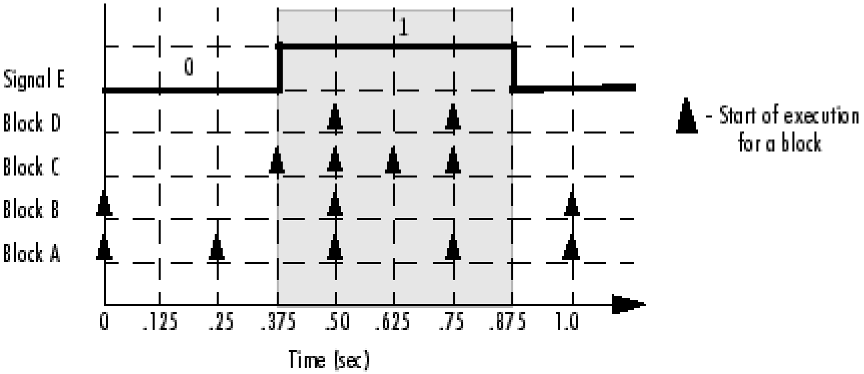
For example, this system contains four discrete blocks and a control signal. The discrete blocks are

- 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.

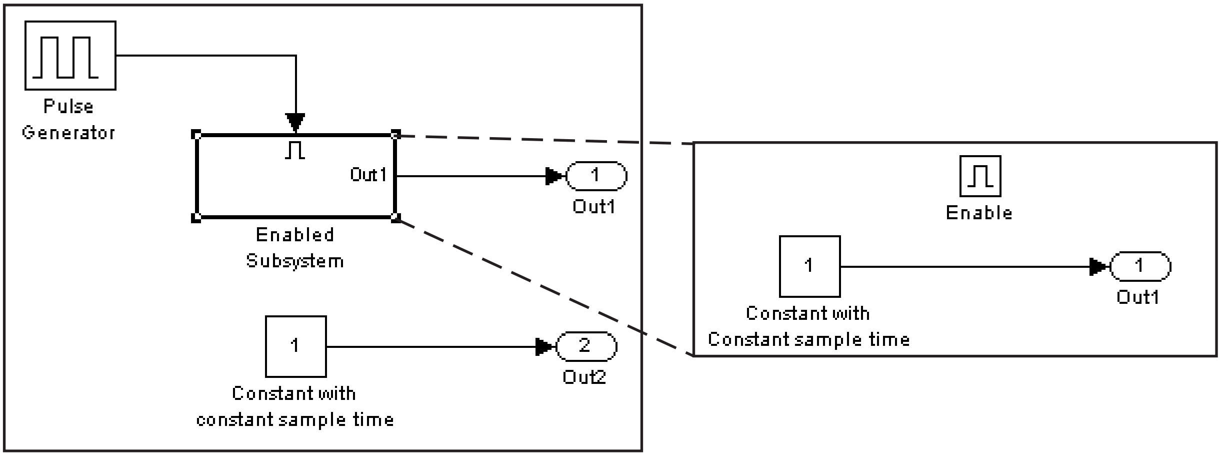
Using Blocks with Constant Sample Times in Enabled Subsystems

Simulink places certain restrictions on connecting blocks with constant sample times (see “Constant Sample Time” on page 1-44) to the output port of a conditionally executed subsystem.

- Simulink displays an error if you connect a Model or S-Function block with constant sample time to the output port of a conditionally executed subsystem.
- Simulink converts the sample time of any built-in block with a constant sample time to a different sample time, such as the fastest discrete rate in the conditionally executed subsystem.

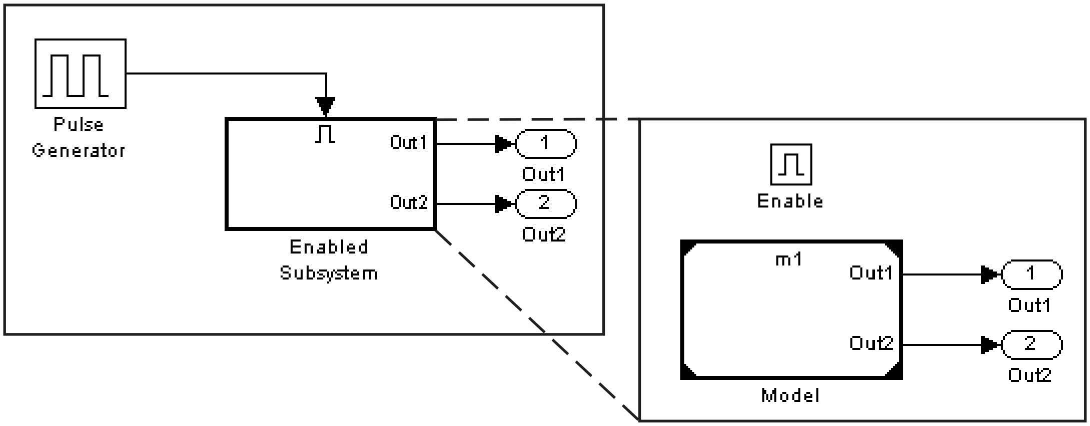
To avoid the error or conversion, either manually change the sample time of the block to a non-constant sample time or use a Signal Conversion block. The example below shows how to use the Signal Conversion block to avoid these errors.

Consider the following Simulink model `m1.mdl`.


Pulse Generator
Out1
Enabled Subsystem
1
Out1
1
2
Constant with constant sample time
Out2
Enable
1
Constant with Constant sample time
1
Out1

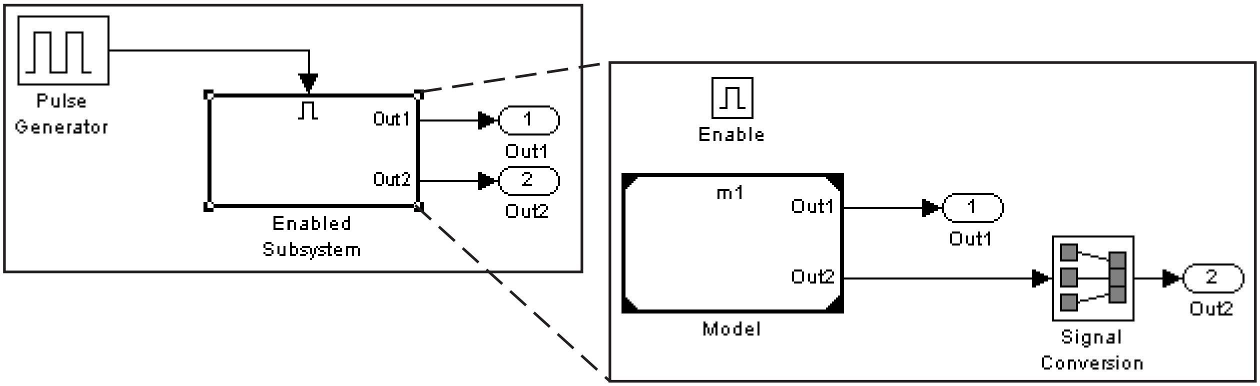
The two Constant blocks in this model have constant sample times. When you simulate the model, Simulink converts the sample time of the Constant block inside the enabled subsystem to the rate of the Pulse Generator. If you simulate the model with sample time colors displayed (see “Displaying Sample Time Colors” on page 3-10), Simulink shows the Pulse Generator and Enabled Subsystem in red. However, Simulink colors the Constant and Outport blocks outside of the enabled subsystem magenta, indicating that these blocks still have a constant sample time.

Suppose the model above is referenced from a Model block inside an enabled subsystem in a top-level model, as shown below.



Simulink invokes an error when you try to simulate the top model, indicating that the second output of the Model block may not be wired directly to the enabled subsystem's output port because it has a constant sample time.

To avoid this error, insert a Signal Conversion block between the second output of the Model block and the enabled subsystem's Outport block, as shown below.



Simulink runs this model with no errors. With sample time colors displayed, Simulink colors the Model and Enabled Subsystem blocks yellow, indicating that these are hybrid systems, i.e., systems that contain multiple sample times.

Triggered Subsystems

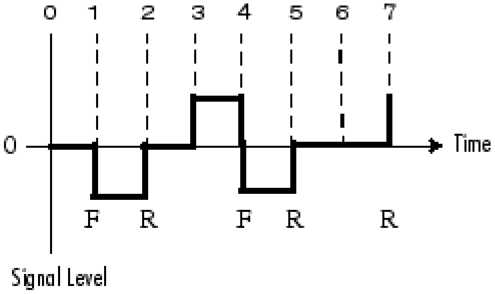
Triggered subsystems are subsystems that execute each time a trigger event occurs.

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

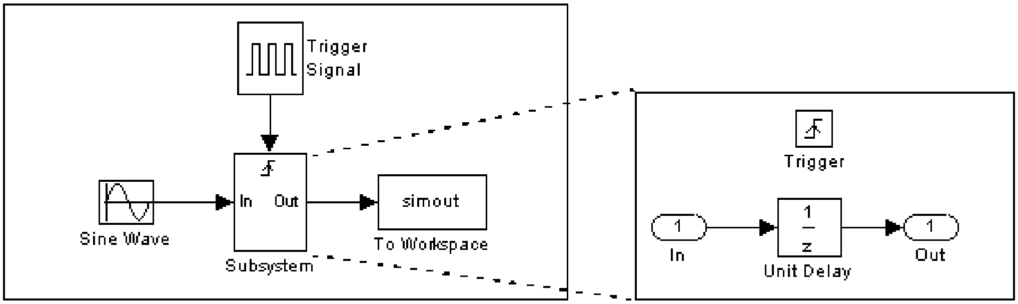
- `rising` triggers execution of the subsystem when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative).
- `falling` triggers execution of the subsystem when the control signal falls from a positive or a zero value to a negative value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

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

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



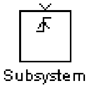
A simple example of a triggered subsystem is illustrated.



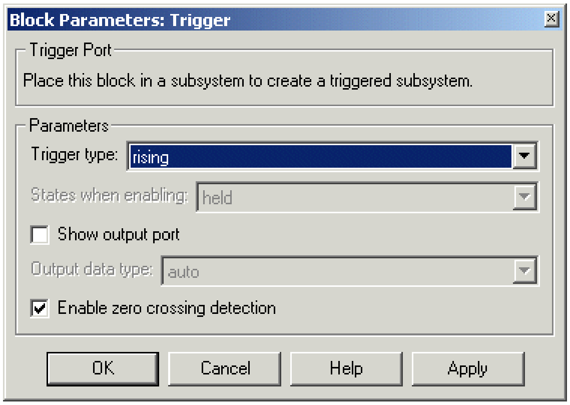
In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Creating a Triggered Subsystem

You create a triggered subsystem by copying the Trigger block from the Ports & Subsystems library into a subsystem. 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

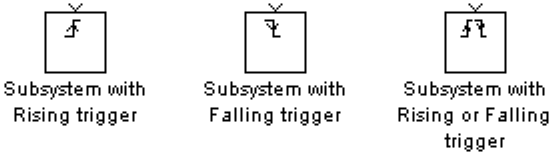
□ Show output port

Output data type: auto

☑ Enable zero crossing detection

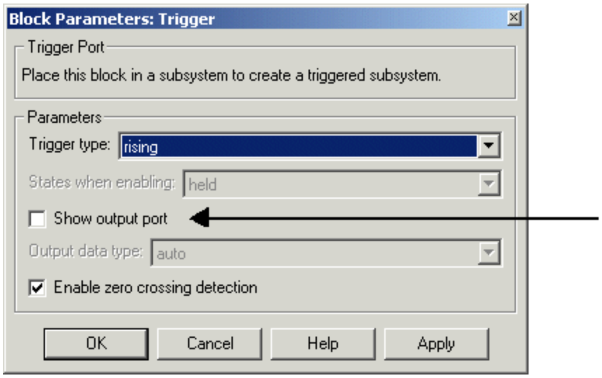
Select the trigger type.

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.



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.

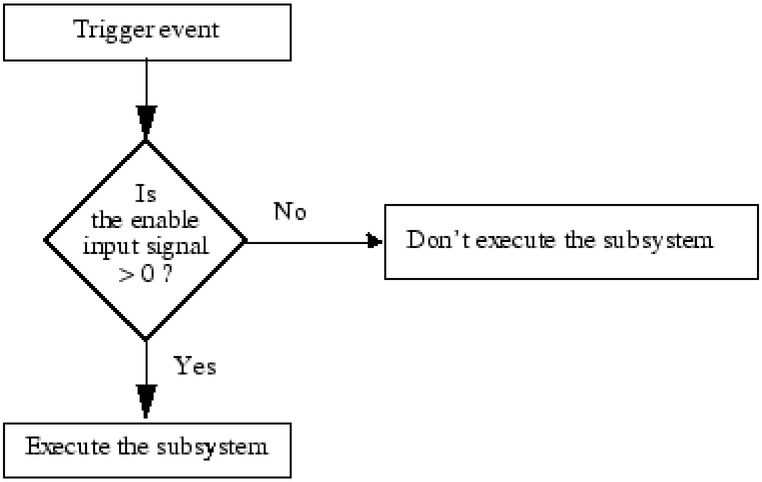
The **Output data type** field allows you to specify the data type of the output signal as `auto`, `int8`, or `double`. The `auto` option causes the data type of the output signal to be set to the data type (either `int8` or `double`) of the port to which the signal is connected.

Blocks That a Triggered Subsystem Can Contain

All blocks in a triggered subsystem must have either inherited (`-1`) or constant (`inf`) sample time. This is to indicate that the blocks in the triggered subsystem run only when the triggered subsystem itself runs, i.e., when it is triggered. This requirement means that a triggered subsystem cannot contain continuous blocks, such as the Integrator 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.

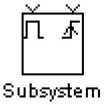


A triggered and enabled subsystem contains both an enable input port and a trigger input port. When the trigger event occurs, Simulink checks the enable input port to evaluate the enable control signal. If its value is greater than zero, Simulink executes the subsystem. If both inputs are vectors, the 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 Ports & Subsystems library into an existing subsystem. Simulink adds enable and trigger symbols and enable and trigger and enable control inputs to the Subsystem block.

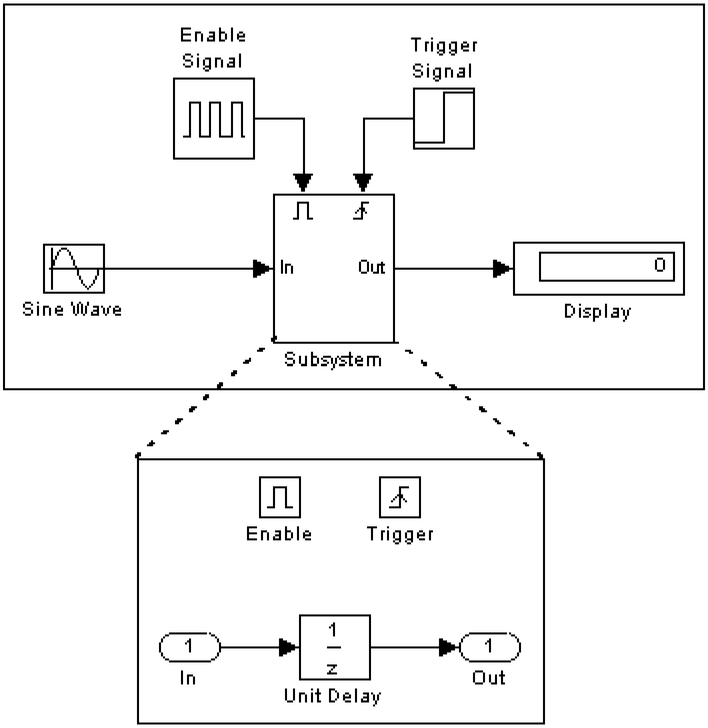


You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see “Setting Output Values While the Subsystem Is Disabled” on page 3-42. Also, you can specify what the values of the states are when the subsystem is reenabled. See “Setting States When the Subsystem Becomes Reenabled” on page 3-43.

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

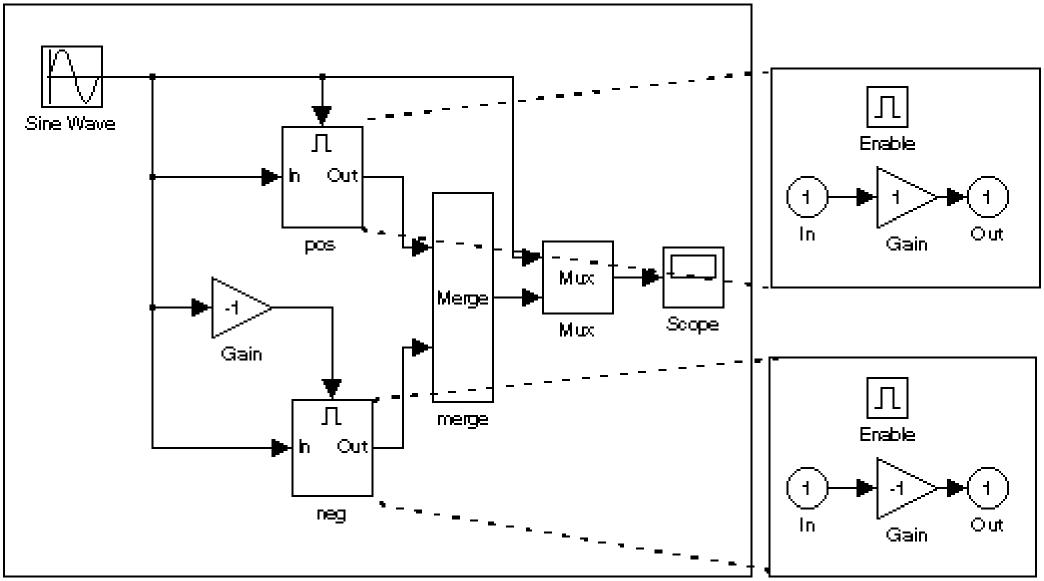
A simple example of a triggered and enabled subsystem is illustrated in the model below.



Creating Alternately Executing Subsystems

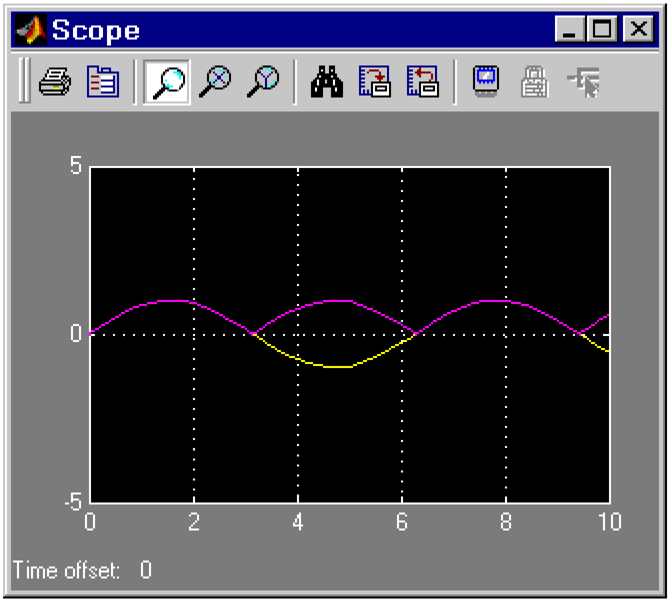
You can use conditionally executed subsystems in combination with Merge blocks to create sets of subsystems that execute alternately, depending on the current state of the model.

The following figure shows a model that uses two enabled blocks and a Merge block to model a full-wave rectifier – a device that converts AC current to pulsating DC current.



The block labeled “pos” is enabled when the AC waveform is positive; it passes the waveform unchanged to its output. The block labeled “neg” is enabled when the waveform is negative; it inverts the waveform. The Merge block passes the output of the currently enabled block to the Mux block, which passes the output, along with the original waveform, to the Scope block.

The Scope creates the following display.



Function-Call Subsystems

A function-call subsystem is a subsystem that another block can invoke directly during a simulation. It is analogous to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods (see “Block Methods” on page 1-12) of the blocks that the subsystem contains in sorted order (see “How Simulink Determines the Sorted Order” on page 4-31). The block that invokes a function-call subsystem is called the function-call initiator. Stateflow®, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

To create a function-call subsystem, drag a Function-Call Subsystem block from the Simulink Ports & Subsystems library into your model and connect a function-call initiator to the function-call port displayed on top of the subsystem. You can also create a function-call subsystem from scratch by first creating a Subsystem block in your model and then creating a Trigger block in the subsystem and setting the Trigger block’s **Trigger type** to `function-call`.

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

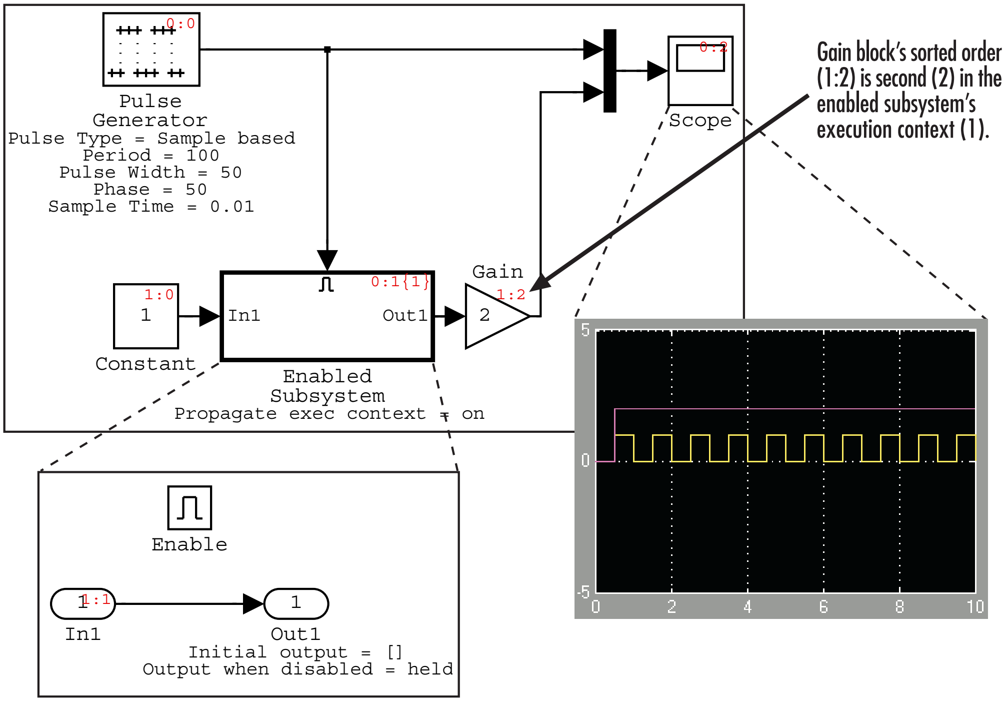
A function-call initiator can invoke a periodic function-call subsystem only once per time step and must invoke the subsystem periodically. If the initiator invokes a periodic function-call subsystem aperiodically, Simulink halts the simulation and displays an error message. The blocks in a periodic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

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

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 3-61).

The following model illustrates conditional execution behavior.



Pulse Type = Sample based
Period = 100
Pulse Width = 50
Phase = 50
Sample Time = 0.01

Propagate exec context = on

Initial output = []
Output when disabled = held

Gain block's sorted order (1:2) is second (2) in the enabled subsystem's execution context (1).

Simulink computes the outputs of the Constant block and Gain block only while the enabled subsystem is enabled (i.e., at time steps 0.5 to 1.0, 1.5 to 2.0, and so on). This is because the output of the Constant block is required and the input of the Gain block changes only while the enabled subsystem is enabled. When CE behavior is off, 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. Simulink indicates this in the sorted order labels displayed on the diagram for the Constant and Gain blocks. The notations list the subsystem's (id = 1) as the execution context for the blocks even though the blocks exist graphically at the model's root level (id = 0).

Propagating Execution Contexts

In general, 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 5-56).
- 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`).

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 3-61), 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.

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

Behavior for Switch Blocks

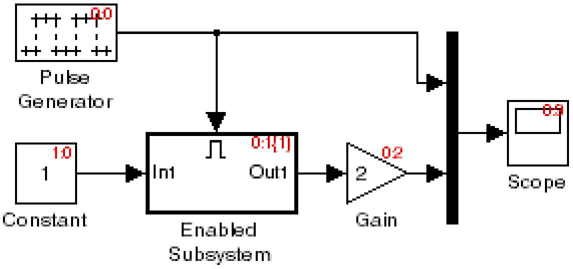
This behavior treats the input branches of a Switch or Multiport Switch block as invisible, conditionally executed subsystems, each of which has its own execution context that is enabled only when the switch's control input selects the corresponding data input. As a result, switch branches execute only 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 belongs and `b` is an index that indicates the block's sorted order in the subsystem's execution context, e.g., 0:0 indicates that the block is the first block in the root subsystem's execution context.

If a bus is connected to the block's input, Simulink displays the block's sorted order as **s**:B, e.g., 0:B indicates that the block belongs to the root system's execution context and has a bus connected to its input.

Simulink 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

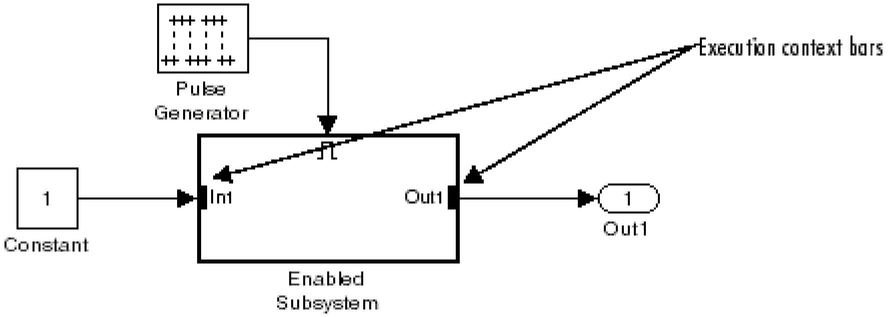
To disable conditional execution behavior for all Switch and Multiport Switch blocks in a model, turn off the `Conditional input branch execution` optimization on the **Optimization** pane of the **Configuration Parameters** dialog box (see "Optimization Pane" on page 11-86). To disable conditional execution behavior 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 model editor's **Format > Block Displays** menu.

Referencing Models

Simulink allows you to include models in other models as blocks, a feature called model referencing. See the following topics for information on using referencing models:

Overview of Model Referencing

You can include one model in another by using Model blocks. Each instance of a Model block represents a reference to another model, called a *referenced model* or a *submodel*. For simulation and code generation, the referenced model effectively replaces the Model block that references it. A referenced model can contain Model blocks and thus reference lower-level models, and so on to any depth. The topmost model in a hierarchy of model references is called the *top model*, *parent model*, or *root 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. 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 specifies different values for variables that define the model's behavior. See "Parameterizing Model References" on page 3-69 for more information.

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 3-79 for more information).

Real-Time Workshop® generates library modules called *Real-Time Workshop targets* for referenced models, and a stand-alone executable for the root model. The parent target invokes the referenced model targets as needed to compute the referenced model outputs. See "Building Subsystems and Working with Referenced Models" in the Real-Time Workshop documentation for more information.

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 3-83 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.

- Independent configuration sets

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

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 3-85. Simulink also provides a command, `find_mdlrefs`, to find all models directly or indirectly referenced by a given model.

Model Referencing Demos

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

1. In the MATLAB Command Window, type

   ```
   demos
   ```

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

2. From the left side of the window, select **Simulink**.

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

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

   This category contains most of the model referencing demos.

Creating a Model Reference

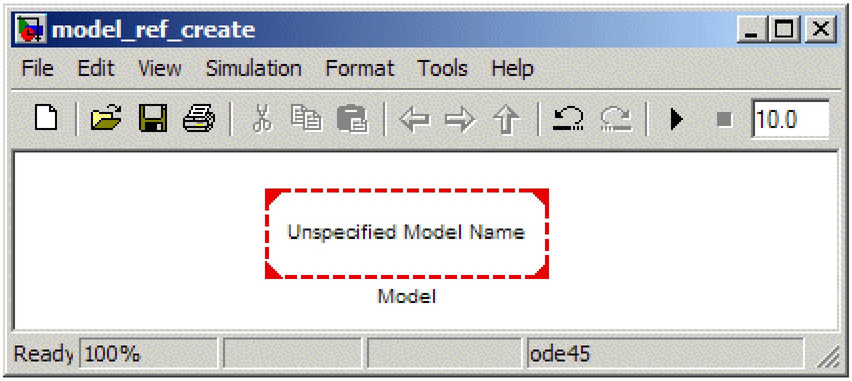
To create a reference to a model in another model:

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

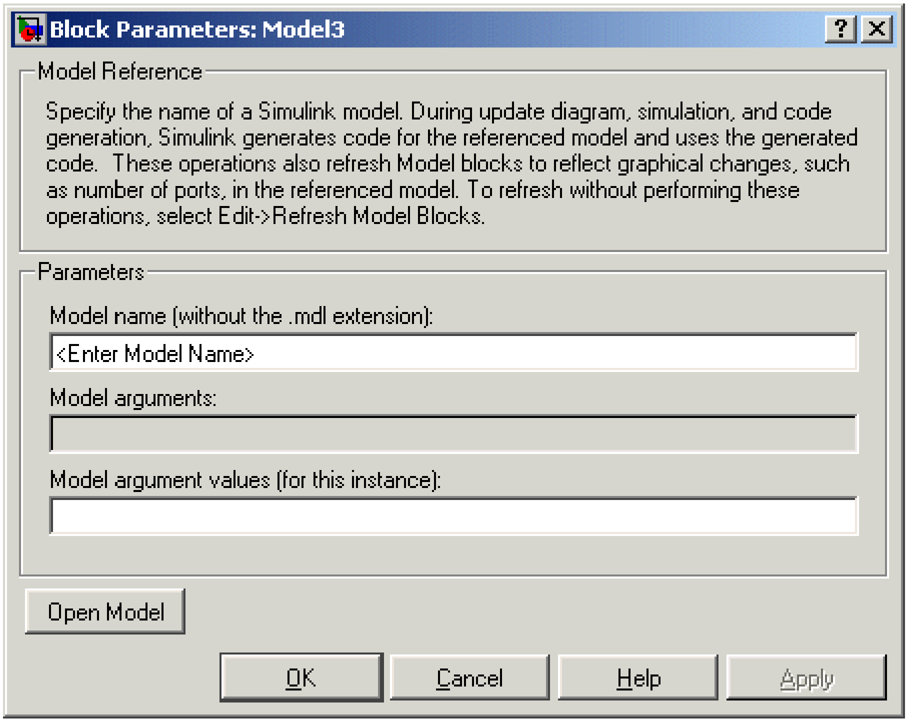
2 If the model in which you want to create the reference (the parent model) is itself referenced by another model, enable the parent model's **Inline parameters** optimization if it is not already enabled (see "Inline parameters" on page 11-90).

Note You must enable the **Inline parameters** optimization for all models in a model reference hierarchy except the hierarchy's top model (see "Model Referencing and the Inline Parameters Optimization" on page 3-69 for more information).

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. When prompted, type `Yes` to access the dialog box and enter a valid model name.



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.

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

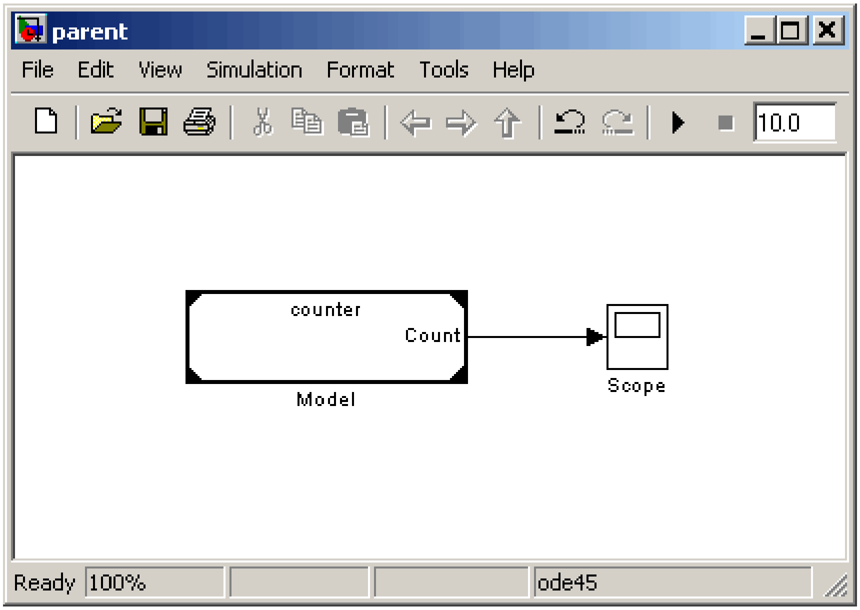
Model arguments:

Model argument values (for this instance):

5 Enter the name of the referenced model in the parameter dialog box's **Model name** field.

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 3-77 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 model editor’s **Edit** menu or from the block’s context menu.

Referenced Model Configuration Sets

A referenced model uses configuration sets in the same way that any other model does. All models in a hierarchy of referenced models can use the same configuration set, or any or all models can have their own sets. When more than one Model block uses the same referenced model, all instances of the referenced model use the same configuration set. See “Configuration Sets” on page 11-37 and “Referencing Configuration Sets” on page 11-49 for details.

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 "Using Model Arguments" on page 3-70).

Model Referencing and the Inline Parameters Optimization

Simulink supports the `off` setting of the inline parameters optimization (see "Inline parameters" on page 11-90) only for the top model in a model reference hierarchy. Simulink ignores the setting for the inline parameters optimization for all other models in the hierarchy and assumes the optimization is enabled. Further, Simulink ignores the settings in the **Tunable Parameter** dialog box (see "Model Parameter Configuration Dialog Box" on page 11-100) for both the top model and referenced models. This means that you cannot use this dialog box to override the inline parameters optimization for selected parameters and thereby permit them to be tuned. As a result of these constraints, the only parameters that you can tune in a model reference hierarchy are the top model's parameters and any global tunable parameters (see "Parameterizing Model References" on page 3-69) used by referenced models.

If you want your model to reference existing models and need to tune their parameters during simulation, you must convert the parameters to global tunable parameters. To facilitate this task, Simulink provides a command that converts tunable parameters specified in the **Tunable Parameter** dialog box to global tunable parameters. 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 `sldemo_mdlref_paramargs` demo to see parameterized model references in action.

Using model arguments requires that you

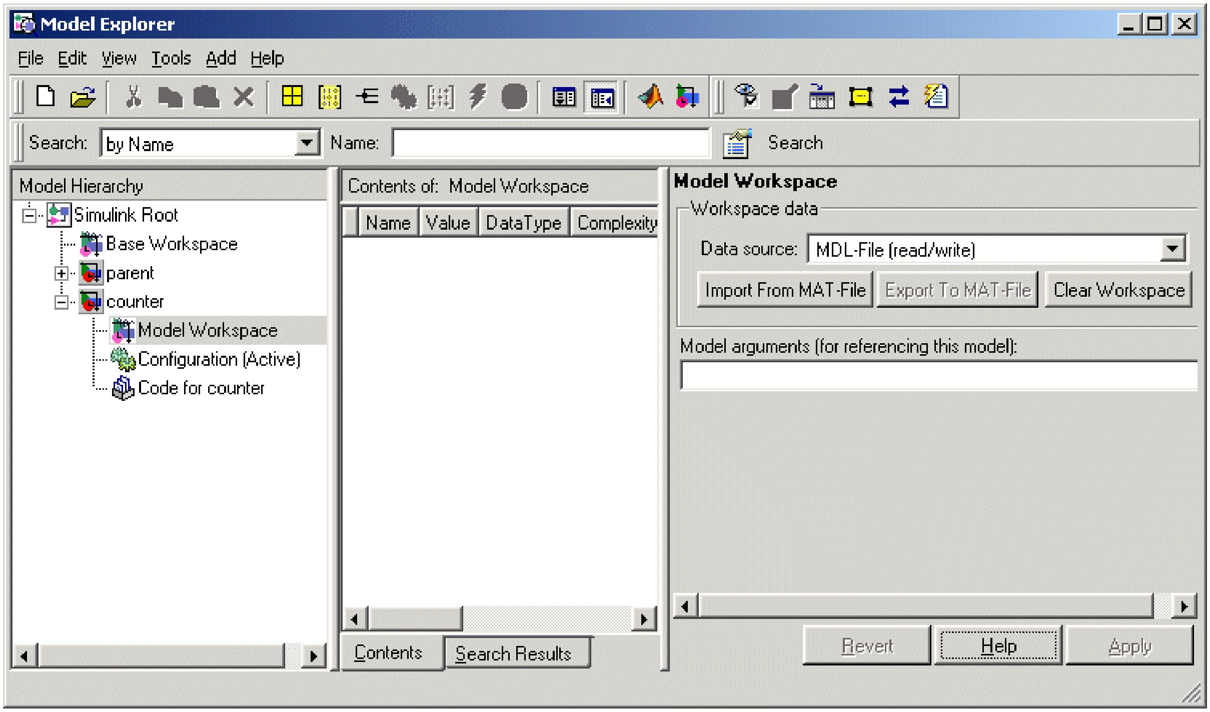
- Declare model workspace variables that determine the model's behavior as model arguments
- Assign values to the model arguments in each reference to the parameterized model

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 Model Explorer.
3. Select the model's workspace in Model Explorer.



Search: by Name

Name:

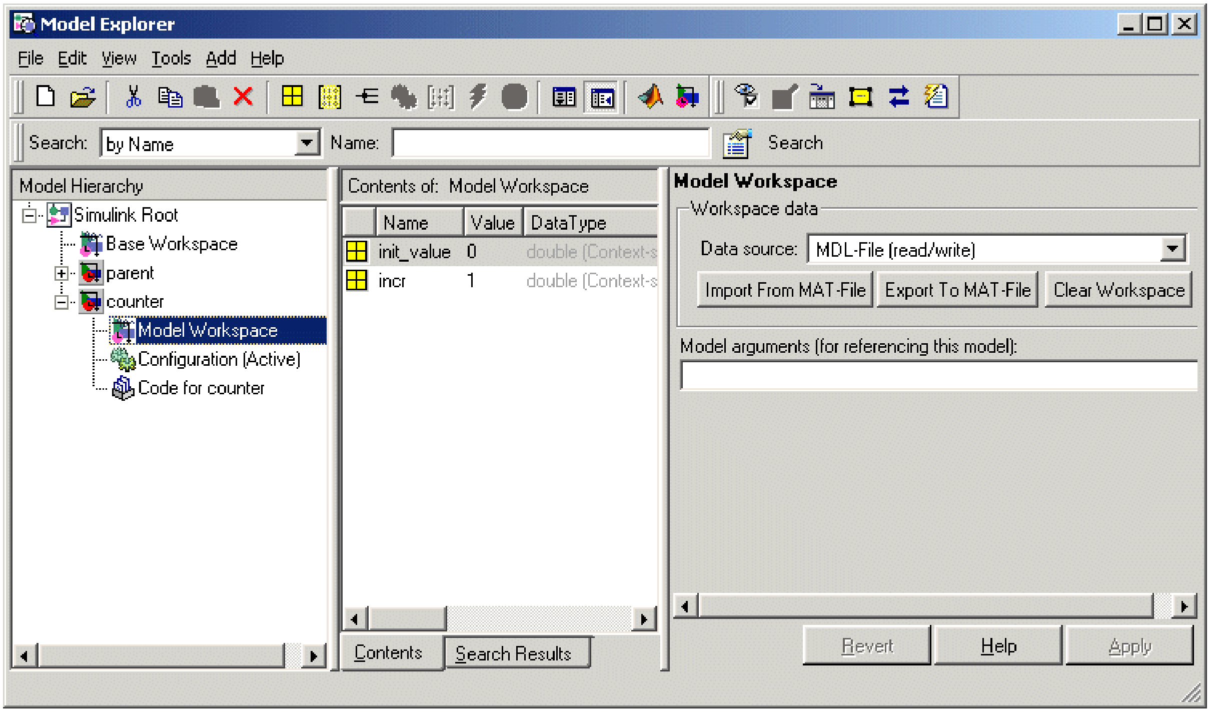
Contents of: Model Workspace

Model Workspace

Data source: MDL-File (read/write)

Model arguments (for referencing this model):

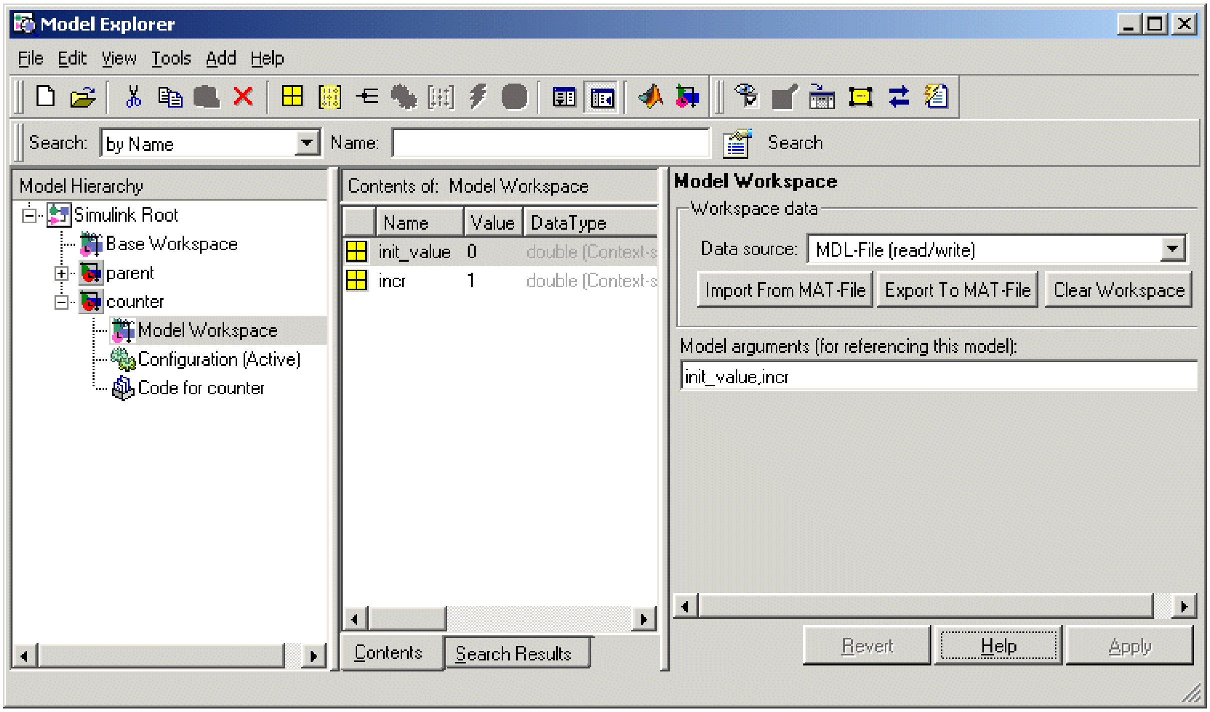
4 If you have not already done so, use Model Explorer to create MATLAB variables in the model's workspace that determine the model's behavior. For example, to create a variable in the model's workspace using Model Explorer, first select the model's workspace in Model Explorer's **Model Hierarchy** pane. Then, from Model Explorer's **Add** menu, select **MATLAB Variable**.



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

Data source: MDL-File (read/write)

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.



Search: by Name Name: Search

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.

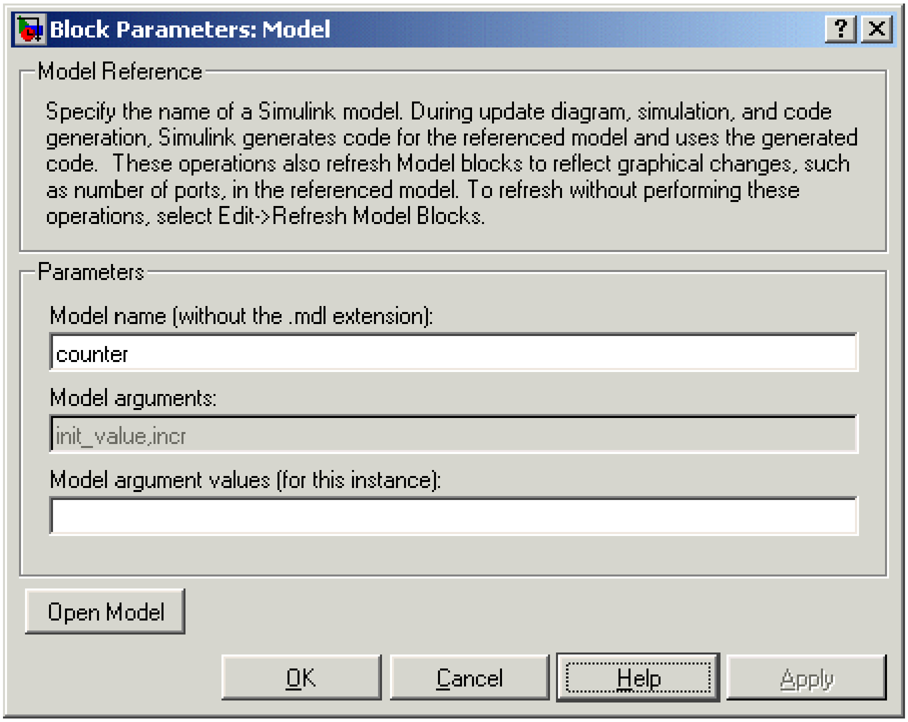
Note If a model does not declare a variable in its workspace as a model argument, the variable has the same value in every reference to the model and cannot be tuned from a parent model. For example, suppose that a model defines a variable `k` in its workspace but does not declare it as a model argument. 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 referenced model's arguments:

1 Open the Model block's parameter dialog box by double-clicking it.



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.

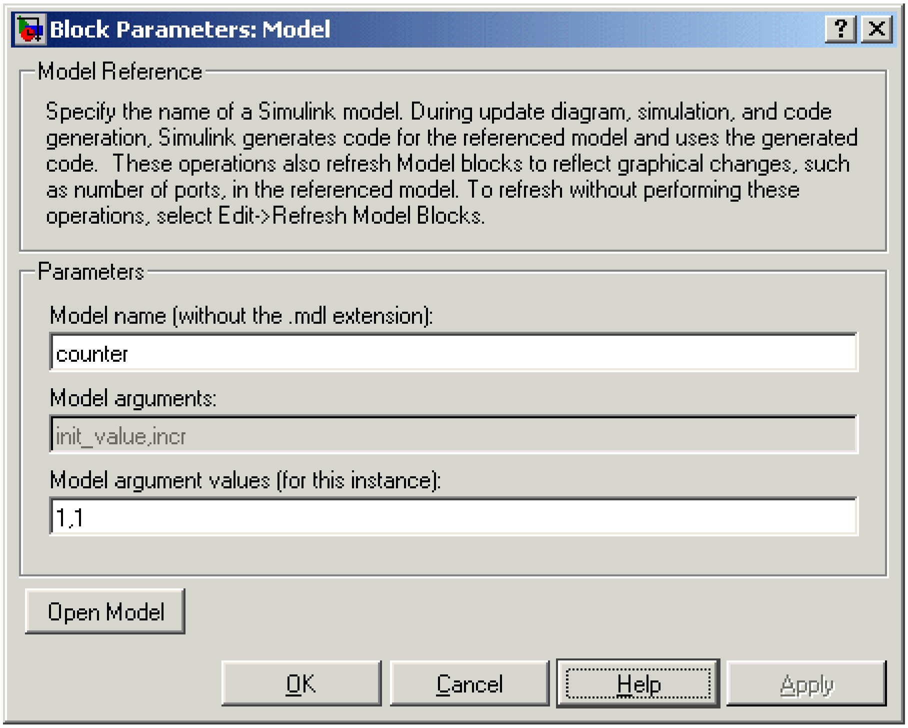
Parameters

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.



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 3-76)

You can use a referenced model that inherits its sample time anywhere in a parent model. By contrast, you cannot use a referenced model that has intrinsic sample times in a triggered, function call, or 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 11-76). 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 11-76).

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 11-105).

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.

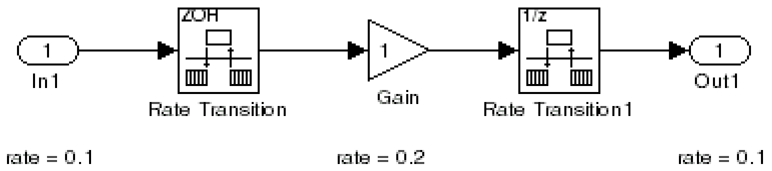
Similarly, the bus connected to a bus input port of a referenced model must be defined by the same bus object that defines the bus input, i.e., the bus must be created by a Bus Creator block whose Bus object parameter is set to the bus object as is the Inport of the referenced model. This explains why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



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 11-140).

Note If the referenced model contains an S-function that should be inlined using a Target Language Compiler file, the S-function must use the `ssSetOptions` macro to set the SS_OPTION_USE_TLC_WITH_ACCELERATOR option in its `mdlInitializeSizes` method. The simulation target will not inline the S-function unless this flag is set. See "Inlining S-Functions" in the Real-Time Workshop Target Language Compiler Reference Guide for more information on inlining S-functions.

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.

Function-Call Models

Simulink allows certain blocks, such as a Function-Call Generator or an appropriately configured custom S-function, to control execution of a referenced model during a time step, using a function-call signal (see "Function-Call Subsystems" on page 3-56 for more information). A referenced model capable of being invoked in this way is called a *function-call model*.

- "Function-Call Model Demo" on page 3-80
- "Creating a Function-Call Model" on page 3-81
- "Referencing a Function-Call Model" on page 3-81
- "Function-Call Model Limitations" on page 3-82

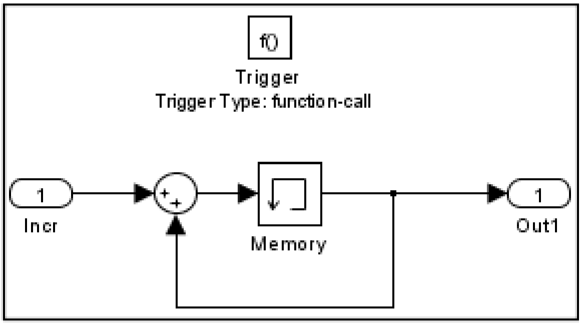
Function-Call Model Demo

To view a function-call model demo, select **Simulink > Modeling Features > Model Reference > Model Reference Function-Call** from the **Demos** pane of the MATLAB Help Browser or execute sldemo_mdlref_fcncall at the MATLAB command line.

Creating a Function-Call Model

To create a function-call model:

1. Insert a Trigger block at the root level of the model.
2. Set the Trigger block's **Trigger type** parameter to `function-call`
3. Create and connect any other blocks required to implement the model.



4. Ensure that the model satisfies the conditions imposed on function-call models (see "Function-Call Model Limitations" on page 3-82).

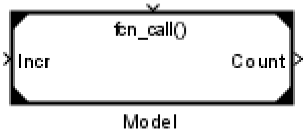
You can now simulate the function-call model either by itself or by running a model that references the function-call model directly or indirectly.

Referencing a Function-Call Model

To create a reference to a function-call model:

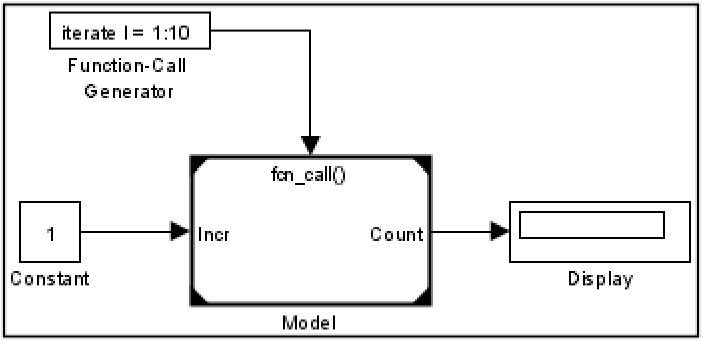
1. Create a Model block in the referencing model that references the function-call model (see "Creating a Model Reference" on page 3-65).

   The top of the Model block displays a function-call port corresponding to the function-call trigger port in the function-call model.



2 Connect a Stateflow chart, Function-Call Generator block, or other function-call-generating block to the Model block's function-call port.

3 Connect the Model blocks inputs and outputs if any to the appropriate blocks in the referencing model.



4 Create and connect any other blocks required to implement the referencing model.

5 Ensure that the referencing model satisfies the conditions for a model to reference other models (see "Model Referencing Limitations" on page 3-85).

You can now simulate the model that references the function-call model.

Function-Call Model Limitations

To be a function-call model, a referenced model must meet the following conditions in addition to the conditions that any referenced model must meet.

- If the function-call model specifies a fixed-step solver and contains one or more blocks that use absolute or elapsed time, the referencing model must trigger the function-call model at the rate specified by the `'Fixed-step size'` option on the **Solver** page of the **Configuration Parameters** dialog box. Otherwise, the referencing model may trigger the function-call model at any rate.
- If the **Sample time type** is `periodic`, the sample-time period cannot contain an offset.

- The model cannot have direct internal connections between its root-level input and output ports.

Note Simulink does not honor the `None` and `Warning` settings for the **Invalid root Inport/Outport block connection** diagnostic for a referenced function-call model. It reports all invalid root port connections as errors.

A limitation also exists on the function-call signal itself, namely, you cannot use vector function-call signals to control a function-call model. The function-call signal must be a scalar.

Model Interfaces

A referenced model's interface consists of its input and output ports (and trigger port in the case of function-call models) 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

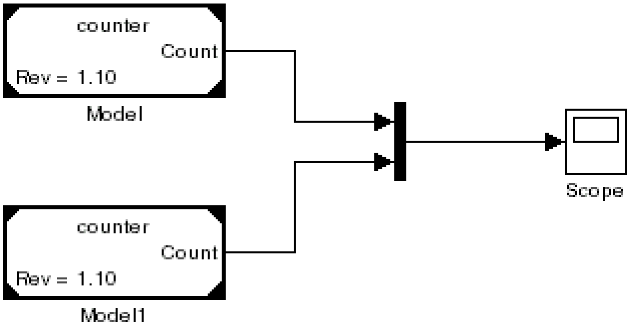
Edit menu. To update a specific Model block, select **Refresh** from the block's context (pop-up) menu.

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 11-131)
- I/O port and parameter mismatch (see "Port and parameter mismatch" on page 11-131)

Displaying Referenced Model Version Numbers

To display the version numbers of the models referenced by a model (see "Managing Model Versions" on page 3-129), 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.

Browsing Model Reference Dependencies

You can use the Simulink Model Dependency Viewer to quickly find and open any model referenced directly or indirectly by another model. See "Model Dependency Viewer" on page 10-39 for more information.

Converting Subsystems to Model References

To convert an atomic subsystem in the current model to a model reference, select **Convert to Model Block** from the subsystem's context (right-click) menu. Simulink creates a new model containing a Model block having the same name as the subsystem. The Model block references a new model having the same name and content as the subsystem.

Note If the MATLAB path includes a file or variable with the same name as the subsystem's, Simulink generates a new name for the new model based on the subsystem's name, e.g., `mysubsystem0` for a model converted from a subsystem named `mysubsystem`.

You can use `Simulink.SubSystem.convertToModelReference` to convert subsystems to model references programmatically. See the Simulink reference documentation for details.

Model Referencing Limitations

This section lists limitations on the use of model referencing. For example, models must meet certain conditions to reference other models or be referenced by other models. There are also restrictions on the use of model referencing with some features of Simulink and products based on Simulink. This section summarizes some of the major limitations placed on model referencing.

Referencing and Referenced Model Limitations

The following limitations apply both to models that reference other models (referencing models) and models referenced by other models.

- In both normal and accelerated simulation, the start time of all referencing and referenced models must be the same. The start time need not be zero unless a code generation target has been specified that requires a zero start time.

- This release ignores the setting of the inline parameters optimization for all models in a reference model except the top model. It assumes that the optimization is enabled for all referenced models.
- Referenced models must use `Simulink.Parameter` objects to specify the tunability of parameters.

  This release ignores tunable parameter information specified by the model's **Model Parameter Configuration** dialog box for all models in a model reference hierarchy, including the top model. This release provides a utility function, `tunablevars2parameterobjects`, to facilitate conversion of parameter tunability information from dialog to `Simulink.Parameter` object form.
- A signal that is propagated into a referenced model from a subsystem of that model cannot propagate out of the model to its parent model. The signal must be explicitly defined in the referenced model; it can then propagate to the parent model.

Referencing Model Limitations

The following limitations apply to models that reference other models:

- Model Browser does not display Model blocks in its tree view.

  Use Model Explorer to browse models referenced by a model.
- Tools that require access to a model's internal data or configuration, including the Model Coverage Tool, Report Generator, the Simulink and Stateflow debuggers, and the Profiler, do not work when invoked from a top model on models that the top model references. This is because the referenced models appear as black boxes to the top model. For example, you cannot use the Simulink Debugger to step from a top model into a referenced model.
- You cannot print a referenced model from a top model.
- If you want to initialize the states of a model that references other models with states, you must specify the initial states in structure format.
- To Workspace and Scope blocks in models referenced by a top model do not log data when you simulate or run code generated from a top model.

- A continuous sample time cannot be propagated to a Model block that is sample-time independent.
- Linearization is not supported for models containing Model blocks.
- Right-clicking a subsystem to build an S-function from it works for subsystems containing Model blocks only if the model is configured to use `ert.tlc`.
- Microsoft Windows platforms impose a limit on the number of distinct models that can be referenced by a parent model. The limit applies only to referenced models configured to be instantiable multiple times, i.e., whose **Total number of instances allowed per top model** configuration parameter is set to `multiple`. The limit depends on the length of the names of the such models referenced by your model. For example, if the average model name length is 25 characters, the limits are approximately 50 models on Microsoft Windows 2000 and 400 models on Microsoft Windows XP. You can use the `find_mdlrefs` command to determine how many distinct models your top model references directly or indirectly.

  The limit does not apply to multiple instances of the same model. For example, suppose that your model references 50 instances of model A and 75 instances of model B. In this case, models A and B count as only two against the referenced model limit, despite the fact that your model references 125 instances of these two models. To avoid exceeding this limit, configure all models that are referenced only once in your model hierarchy as singly instantiable models. To configure a model as singly instantiable, set its **Total number of instances allowed per top model** configuration parameter to `single`.

Referenced Model Limitations

The following limitations apply to models referenced by other models:

- The model must specify that it can be referenced (see "Total number of instances allowed per top model" on page 11-142in the online Simulink documentation).
- This release places some restrictions on the sample times that a referenced model can inherit from the model that references it. See "Model Block Sample Times" on page 3-75in the online Simulink documentation.

- This release places some limitations on I/O connections in referenced models.
- A referenced model must use `Simulink.Bus` objects to specify any buses that it inputs and outputs (see “Bus I/O Limitations” on page 3-77in the online Simulink documentation).
- A referenced model can input or output only those user-defined data types that are fixed-point or defined by `Simulink.DataType` or `Simulink.Bus` objects.
- Goto/From blocks cannot cross model reference boundaries.
- Function-Call signals cannot pass through model reference input and output ports. They can pass through only a function-call control port (see Function-Call Models).
- Real-Time Workshop cannot generate standalone executable (Real-Time Workshop targets) for models that reference models that include noninlined S-functions.
- A referenced model cannot use noninlined S-functions in the following cases:
  - The model uses a variable-step solver.
  - The model is referenced more than once in the model reference hierarchy. The workaround in this case is to make copies of the referenced model, assign different names to the copies, and reference the copies in the model reference hierarchy.
  - The S-function was generated by Real-Time Workshop.
  - The S-function supports use of fixed-point numbers as inputs, outputs, or parameters.
- This release ignores custom code settings in the **Configuration Parameter** dialog box and custom code blocks when generating the simulation target for a referenced model.
- This release does not include Stateflow target custom code in simulation targets generated for referenced models.
- This release does not support referenced models that have asynchronous rates.

- A referenced model can input or output indices. However, Simulink may not be able to detect a 0-based index that is connected to a model port expecting or outputting a 1-based index, or vice versa. See “Index I/O Limitations” on page 3-78in the online Simulink documentation.
- Model blocks referencing models that contain assignment blocks that are not in an iterator subsystem, cannot be placed in an iterator subsystem.
- The Real-Time Workshop S-function target and `grt_malloc`-based target do not support model reference.
- When generating a simulation target for a referenced model that contains an S-function with a TLC file, Simulink inlines the S-function only if the S-function sets the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` flag.
- This release places some restrictions on the use of custom storage classes in referenced models. See Real-Time Workshop documentation for details.
- You cannot log the output of a Ground block in a referenced model even if you testpoint it.
- A model that contains a Stateflow chart cannot be referenced multiple times in the same model reference hierarchy if
  - The Stateflow chart contains exported graphical functions.
  - The Stateflow model contains machine-parented events.

Modeling Control Flow Logic

Simulink allows you to use block diagrams to model control flow logic equivalent to the following C programming language statements:

- `for`
- `if-else`
- `switch`
- `while`

Modeling Conditional Control Flow Logic

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

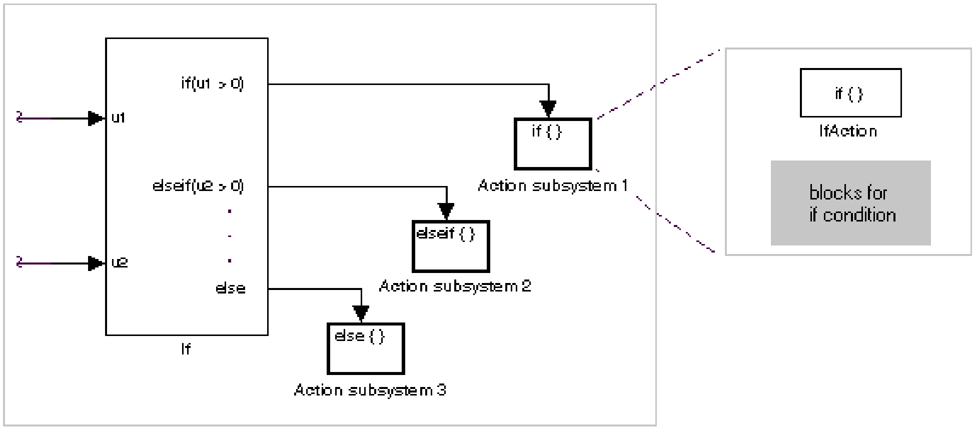
C Statement	Equivalent Simulink Blocks
`if-else`	If, If Action Subsystem
`switch`	Switch Case, Switch Case Action Subsystem

See the following sections for more information:

- "Modeling If-Else Control Flow" on page 3-90
- "Modeling Switch Control Flow" on page 3-92

Modeling If-Else Control Flow

The following diagram models `if-else` control flow.



Construct an `if-else` control flow diagram as follows:

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

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

- Connect each condition output port to an Action subsystem.

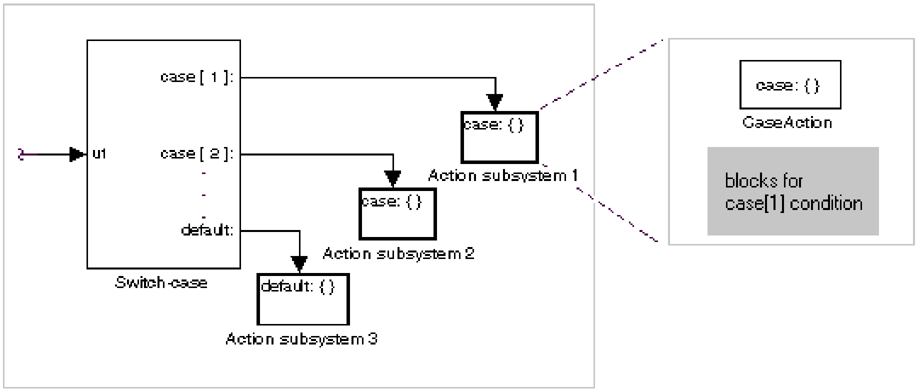
  Each if, elseif, and else condition output port on the If block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing an Action Port block in a subsystem. This creates an atomic Action subsystem with a port named Action, which you then connect to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

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

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

Modeling Switch Control Flow

The following diagram models `switch` control flow.



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

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

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

For more detailed information, see *Simulink Reference* for the Switch Case and Action Port blocks.

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

Modeling While and For Loops

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

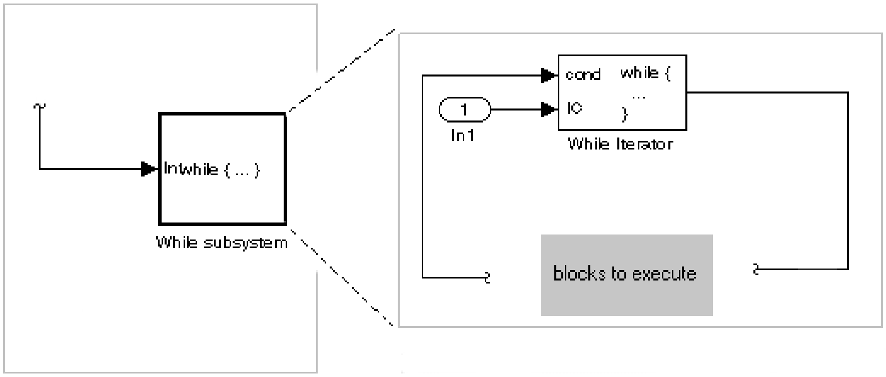
C Statement	Equivalent Simulink Blocks
`do-while`	While Iterator Subsystem
`for`	For Iterator Subsystem
`while`	While Iterator Subsystem

See the following sections for more information:

- “Modeling While Loops” on page 3-93
- “Modeling For Loops” on page 3-96

Modeling While Loops

The following diagram illustrates a Simulink `while` loop.



In this example, Simulink repeatedly executes the contents of the While subsystem at each time step until a condition specified by the While Iterator block is satisfied. In particular, for each iteration of the loop specified by the While Iterator block, Simulink invokes the update and output methods of all the blocks in the While subsystem in the same order that the methods would be invoked if they were in a noniterated atomic subsystem.

Note Simulation time does not advance during execution of a While subsystem's iterations. Nevertheless, blocks in a While subsystem treat each iteration as a time step. As a result, in a While subsystem, the output of a block with states, i.e., a block whose output depends on its previous input, reflects the value of its input at the previous iteration of the `while` loop—not, as one might expect, its input at the previous simulation time step. For example, a Unit Delay block in a While subsystem outputs the value of its input at the previous iteration of the `while` loop—not the value at the previous simulation time step.

Construct a Simulink `while` loop as follows:

- Place a While Iterator block in a subsystem.

  The host subsystem's label changes to `while {...}` to indicate that it is modeling a while loop. These subsystems behave like triggered subsystems.

This subsystem is host to the block programming you want to iterate with the While Iterator block.

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

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

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

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

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

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

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

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

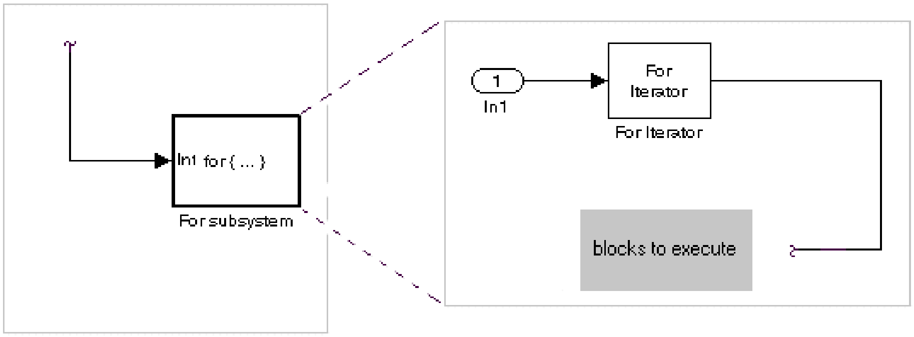
- Create a block diagram in the subsystem that defines the subsystem's outputs.

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

For more information, see the While Iterator block.

Modeling For Loops

The following diagram models a `for` loop:



In this example, Simulink executes the contents of the For subsystem multiples times at each time step with the number of iterations being specified by the input to the For Iterator block. In particular, for each iteration of the `for` loop, Simulink invokes the update and output methods of all the blocks in the For subsystem in the same order that the methods would be invoked if they were in a noniterated atomic subsystem.

Note Simulation time does not advance during execution of a For subsystem's iterations. Nevertheless, blocks in a For subsystem treat each iteration as a time step. As a result, in a For subsystem, the output of a block with states, i.e., a block whose output depends on its previous input, reflects the value of its input at the previous iteration of the `for` loop—not, as one might expect, its input at the previous simulation time step. For example, a Unit Delay block in a For subsystem outputs the value of its input at the previous iteration of the `for` loop—not the value at the previous simulation time step.

Construct a Simulink `for` loop as follows:

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

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

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

  You can also set the number of iterations directly in the properties dialog.

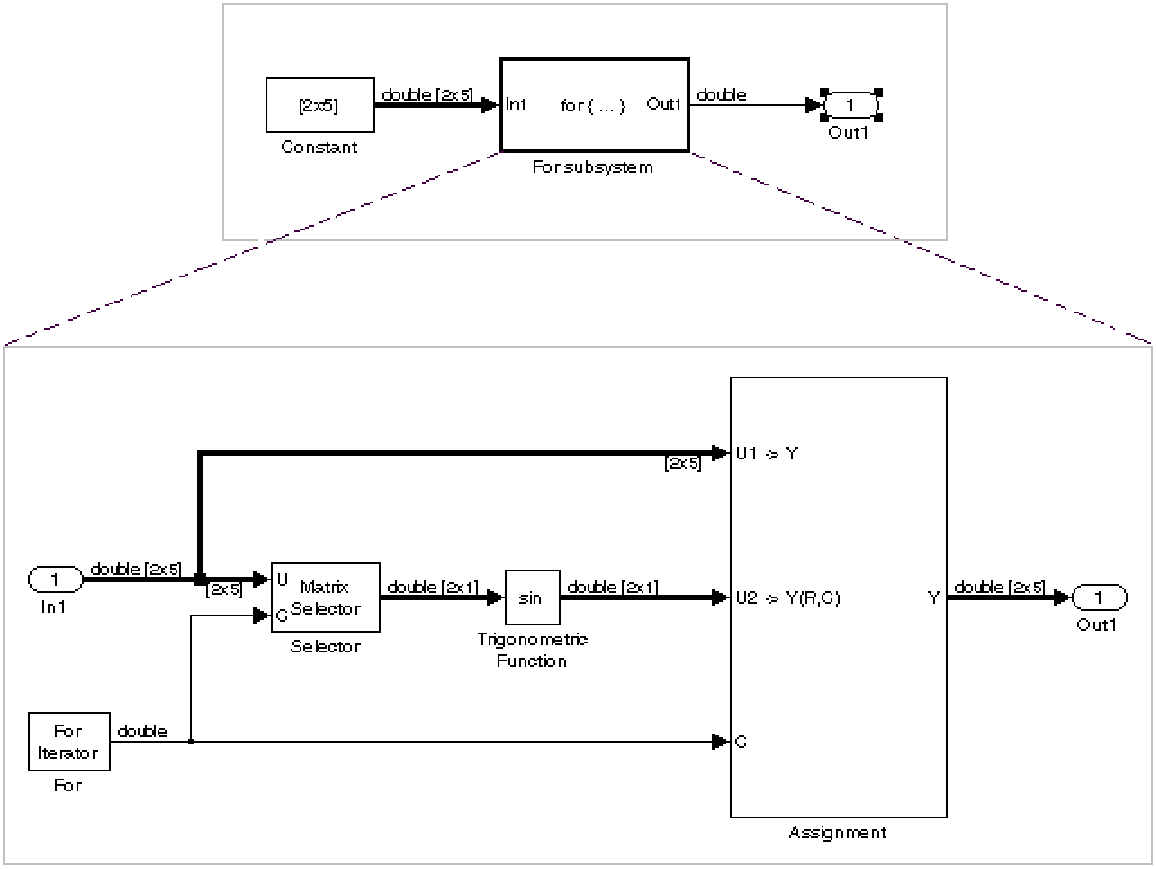
- You can set the For Iterator block to output its iterator value for use in the block programming of the For Iterator Subsystem.

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

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

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

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



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

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

4 The sine value 2-by-1 matrix is passed to an Assignment block.

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

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

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

Using Callback Functions

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 functions*, are specified by block, port, or model parameters. For example, the function specified by a block's `NameChangeFcn` parameter is executed when you double-click that block's name or its path changes.

For more information on callbacks, see:

- "Tracing Callbacks" on page 3-100
- "Creating Model Callback Functions" on page 3-100
- "Creating Block Callback Functions" on page 3-102
- "Port Callback Parameters" on page 3-106

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 for Simulink, select the **Callback tracing** option on the Simulink **Preferences** dialog box or execute `set_param(0, 'CallbackTracing', 'on')`. This option causes Simulink to list callbacks in the MATLAB Command Window as they are invoked. This option applies to all of Simulink, not just one model.

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 3-133) to create model callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the model parameter corresponding to the callback (see "Model Callback Functions" on page 3-101).

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

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

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

- PreLoadFcn
- PostLoadFcn
- StartFcn
- StopFcn
- CloseFcn

Model Callback Functions

The following table describes callback functions associated with models.

Parameter	When Executed
CloseFcn	Before the block diagram is closed. Any ModelCloseFcn and DeleteFcn callbacks set on blocks in the model are called prior to the model's CloseFcn. The DestroyFcn callback of any blocks in the model is called after the model's CloseFcn.
PostLoadFcn	After the model is loaded. Defining a callback routine for this parameter might be useful for generating an interface that requires that the model has already been loaded.
InitFcn	Called at start of model simulation.
PostSaveFcn	After the model is saved.
PreLoadFcn	Before the model is loaded. Defining a callback routine for this parameter might be useful for loading variables used by the model.
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 block callback functions interactively or programmatically. Use the **Callbacks** pane of the block's **Block Properties** dialog box (see "Callbacks Pane" on page 4-20) to create block callbacks interactively. To create a callback programmatically, use the `set_param` command to assign a MATLAB expression that implements the function to the block parameter corresponding to the callback (see "Block Callback Parameters" on page 3-102).

Note A callback for a masked subsystem cannot directly reference the parameters of the masked subsystem (see "About Masks" on page 13-2). 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 model containing the block is closed interactively or by 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.
DeleteChildFcn	After a block is deleted from a subsystem. Only Subsystem blocks have a DeleteChildFcn callback.
DeleteFcn	After a block is graphically deleted, e.g., when the user graphically deletes the block, invokes delete_block on the block, or closes the model containing the block. If the block is deleted by invoking delete_block or by closing the model, the block's DestroyFcn is called after the DeleteFcn. When the DeleteFcn is called, the block handle is still valid and can be accessed using get_param. This callback is recursive for Subsystem blocks.
DestroyFcn	When the block has been destroyed, e.g., when the user invokes delete_block on either the block or a subsystem containing the block or closes the model containing the block. If the block was not previously graphically deleted, the block's DeleteFcn is called prior to the DestroyFcn.
InitFcn	Before the block diagram is compiled and before block parameters are evaluated.

Parameter	When Executed
ErrorFcn	When an error has occurred in a subsystem. The callback function should have the following form: errorMsg = errorHandler(subsys, errorType) where errorHandler is the name of the callback function, subsys is a handle to the subsystem in which the error occurred, errorType is a string that indicates the type of error that occurred, and errorMsg is a string specifying the text of an error message to be displayed to the user. Simulink displays the error message returned by the callback function.
LoadFcn	After the block diagram is loaded. This callback is recursive for Subsystem blocks.
ModelCloseFcn	Before the block diagram is closed. When the model is closed, the block's ModelCloseFcn is called prior to its DeleteFcn. This callback is recursive for Subsystem blocks.
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.

Parameter	When Executed
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 online Simulink reference) or the **Create Subsystem** item in model editor's **Edit** menu. The ParentCloseFcn of blocks at the root model level is not called when the model is closed.
PostSaveFcn	After the block diagram is saved. This callback is recursive for Subsystem blocks.
PreCopyFcn	Before a block is copied. The callback is recursive for Subsystem blocks (that is, if you copy a Subsystem block that contains a block for which the PreCopyFcn parameter is defined, that routine is also executed). The block's CopyFcn is called after all PreCopyFcn callbacks are executed, unless the PreCopyFcn invokes the error command either explicitly or via a command used in any PreCopyFcn. The PreCopyFcn is also executed if an add_block command is used to copy the block.
PreDeleteFcn	Before a block is graphically deleted, e.g., when the user graphically deletes the block or invokes delete_block on the block. The PreDeleteFcn is not called when the model containing the block is closed. The block's DeleteFcn is called after the PreDeleteFcn unless the PreDeleteFcn invokes the error command either explicitly or via a command used in the PreDeleteFcn.
PreSaveFcn	Before the block diagram is saved. This callback is recursive for Subsystem blocks.

Parameter	When Executed
StartFcn	After the block diagram is compiled and before the simulation starts. In the case of an S-Function block, StartFcn executes immediately before the first execution of the block's mdlProcessParameters function. See "S-Function Callback Methods" in the online Simulink documentation for more information.
StopFcn	At any termination of the simulation. In the case of an S-Function block, StopFcn executes after the block's mdlTerminate function executes. See "S-Function Callback Methods" in the online Simulink documentation for more information.
UndoDeleteFcn	When a block deletion 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 source.

  The data source can be the model's MDL-file, a MAT-file, or M-code stored in the model file (see "Data source" on page 3-111 for more information).

- You can interactively reload and save MAT-file and M-code data sources.
- The only kinds of Simulink data objects that a model workspace can contain are
  - `Simulink.Parameter` objects
  - `Simulink.Signal` objects whose storage class is `auto`
- In general, parameter variables in a model workspace are not tunable.

  However, you can tune model workspace variables declared as model arguments (see "Using Model Arguments" on page 3-70 for more information).

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 3-63), 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.

Note When you use a workspace variable as a block parameter, Simulink creates a copy of the variable during the compilation phase of the simulation and stores the variable in memory. This can cause your system to run out of memory during simulation, or in the process of generating code. Your system might run out of memory if

- You have large models with many parameters
- You have a model with parameters that have a large number of elements

This issue does not affect the amount of memory that is used to represent parameters in generated code.

See the following topics for information on model workspace:

- "Changing Model Workspace Data" on page 3-108
- "Model Workspace Dialog Box" on page 3-110

Changing Model Workspace Data

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

- "Changing Workspace Data Whose Source Is the Model File" on page 3-108
- "Changing Workspace Data Whose Source Is a MAT-File" on page 3-109
- "Changing Workspace Data Whose Source Is M-Code" on page 3-109

Changing Workspace Data Whose Source Is the Model File

If a model workspace's data source is data stored in the model, you can use Model Explorer (see "The Model Explorer" on page 10-2) or MATLAB commands to change the model's workspace (see "Using MATLAB Commands to Change Workspace Data" on page 3-109).

For example, to create a variable in a model workspace, using Model Explorer, first select the workspace in Model Explorer's **Model Hierarchy** pane. Then select **MATLAB Variable** from Model Explorer's **Add** menu or toolbar.

You can similarly use the Add menu or Model Explorer's toolbar to add a `Simulink.Parameter` object to a model workspace.

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

Changing Workspace Data Whose Source Is a MAT-File

You can also use Model Explorer or MATLAB commands to modify workspace data whose source is a MAT-file. In this case, if you want to make the changes permanent, you must save the changes to the MAT-file, using the **Save To Source** button on the model workspace dialog box (see "Model Workspace Dialog Box" on page 3-110). To discard changes to the workspace, use the **Reinitialize From Source** button on the model workspace's dialog box.

Changing Workspace Data Whose Source Is M-Code

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

Using MATLAB Commands to Change Workspace Data

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

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

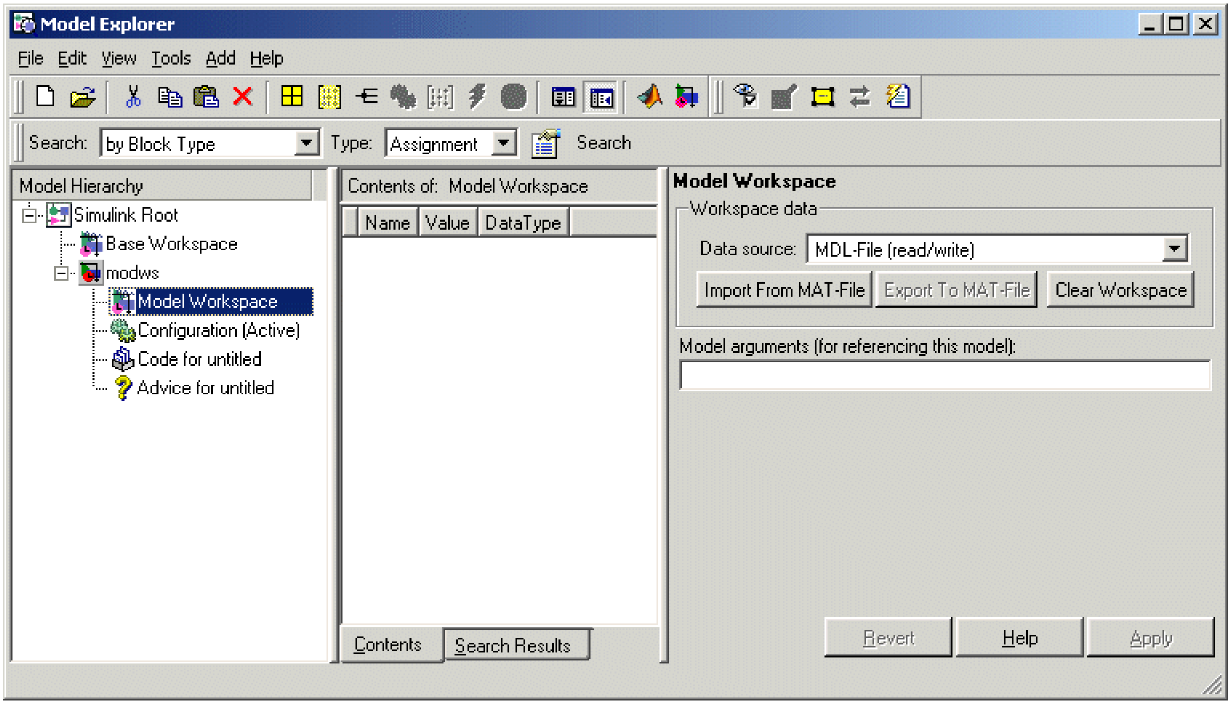
This command returns a handle to a `Simulink.ModelWorkspace` object whose properties specify the source of the data used to initialize the model workspace. Edit the properties to change the data source. Use the workspace's methods to list, set, and clear variables, evaluate expressions in, and save and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

For example, the following MATLAB sequence of commands creates variables specifying model parameters in the model's workspace, saves the parameters, modifies one of them, and then reloads the workspace to restore it to its previous state.

```
hws = get_param(bdroot, 'modelworkspace');
hws.DataSource = 'MAT-File';
hws.FileName = 'params';
hws.assignin('pitch', -10);
hws.assignin('roll', 30);
hws.assignin('yaw', -2);
hws.saveToSource;
hws.assignin('roll', 35);
hws.reload;
```

Model Workspace Dialog Box

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



Search: by Block Type

Type: Assignment

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 3-112).

- `MAT-File`

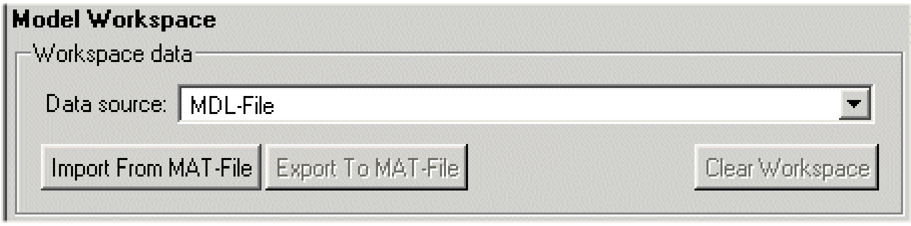
  Specifies that the data source is a MAT file. Selecting this option causes additional controls to appear (see "MAT-File Source Controls" on page 3-112).

- `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 3-113).

MDL-File Source Controls

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



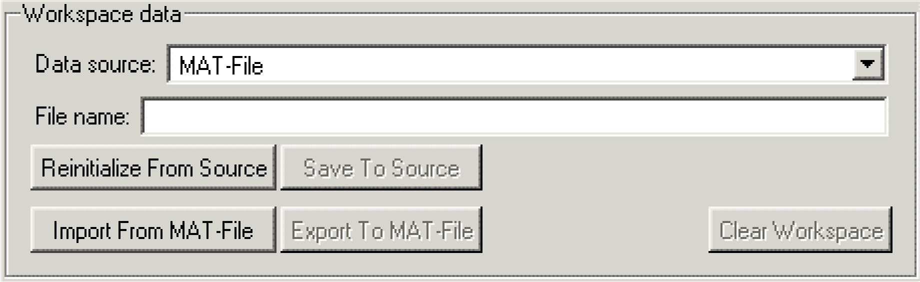
Import From MAT-File. This button lets you import data from a MAT-file. Selecting the button causes Simulink to display a file selection dialog box. Use the dialog box to select the MAT file that contains the data you want to import.

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



Workspace data

Data source: MAT-File

File name:

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

Reinitialize From Source. Clears the workspace and reloads the data from the MAT-file specified by the **File name** field.

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

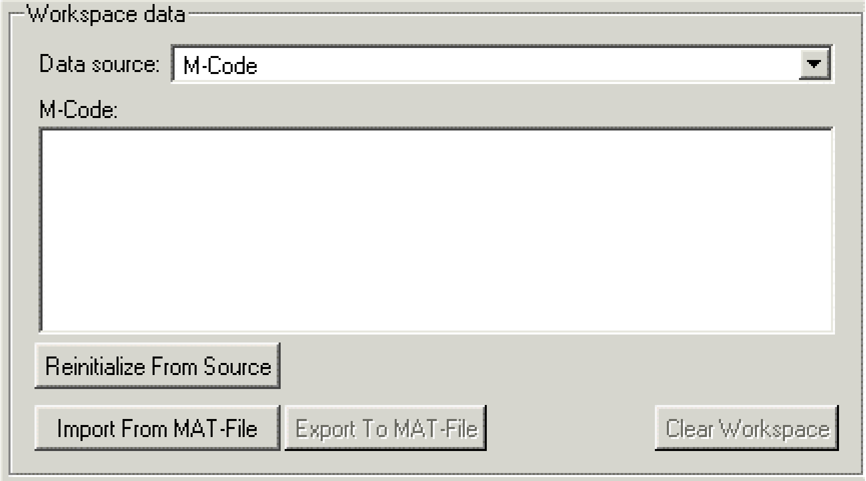
Import From MAT-File. Loads data from a specified MAT file into the selected model workspace without first clearing the workspace. Selecting this option causes Simulink to display a file selection dialog box. Use the dialog box to enter the name of the MAT-file that contains the data to be imported.

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

Clear Workspace. Clears the selected workspace.

M-Code Source Controls

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



Data source: M-Code

M-Code. Specifies M-code that initializes the selected workspace. To change the initialization code, edit this field, then select the **Reinitialize from source** button on the dialog box to clear the workspace and execute the modified code.

Reinitialize from Source. Clears the workspace and executes the contents of the **M-Code** field.

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

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

Clear Workspace. Clears the selected workspace.

Model Arguments

This field allows you to specify arguments that can be passed to instances of this model referenced by another model. See “Using Model Arguments” on page 3-70 for more information.

Working with Data Stores

Data stores are signals that are accessible at any point in a model hierarchy at or below the level in which they are defined. Because they are accessible across model levels, data stores allow subsystems and model references to share data without having to use I/O ports to pass the data from level to level (see “Data Store Examples” on page 3-118 for examples of using data stores to share data among subsystems and model references).

The following topics provide more information on Data Stores:

- “Defining Data Stores” on page 3-115
- “Accessing Data Stores” on page 3-117
- “Data Store Examples” on page 3-118

Defining Data Stores

Defining a data store entails creating an object whose properties specify the properties of the data store. You can use either Data Store Memory blocks or instances of `Simulink.Signal` class to define data stores. Each approach has advantages. Data Store Memory blocks give you more control over the scope of data stores within a model and allow initialization of data stores. `Simulink.Signal` objects avoid cluttering a model with blocks and allows data stores to be visible across model reference boundaries.

Using Data Store Memory Blocks to Define Data Stores

To use a Data Store Memory block to define a data store, drag an instance of the block into the model at the topmost level from which you want the data store to be visible. For example, to define a data store that is visible at every level in a model (except in model references), drag the Data Store Memory block into the root level of the model. To define a data store that is visible only in a particular subsystem (and the subsystems that it contains), drag the block into the subsystem. Once you have created the Data Store Memory block, use its parameter dialog box to define the data stores properties, including its name, data type, complexity.

Using Signal Objects to Define Data Stores

To use a signal object to define a data store, create an instance of `Simulink.Signal` object in a workspace that is visible to every model that needs to access the data store. For example, to define a data store that is visible to a top model and all the models that it references, use Model Explorer or MATLAB commands to create the signal object in the base (i.e., MATLAB) workspace. To define a data store that is visible only in a particular model, create the signal object in the model's workspace (see "Changing Model Workspace Data" on page 3-108). You can use `Simulink.Signal` objects to define data stores that are visible in only one model (a local data store) or in a top model and the models that the top model references (a global data store).

When creating the object, assign it to a workspace variable whose name is the name you want to be assigned to the data store. Once you have created the object, use Model Explorer or MATLAB commands to set the following properties of the signal object to the values that you want the corresponding data store property to have.

- `DataType`
- `Dimensions`
- `Complexity`
- `SampleTime`
- `SamplingMode`
- `StorageClass`

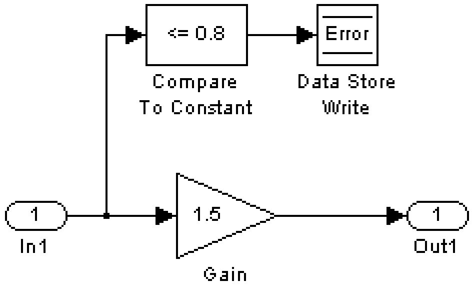
For example, the following commands define a data store named Error in the MATLAB workspace:

```
Error = Simulink.Signal;
Error.Description = 'Use to signal that subsystem output ...
is invalid';
Error.DataType = 'boolean';
Error.Complexity = 'real';
Error.Dimensions = 1;
Error.SamplingMode='Sample based';
Error.SampleTime = 0.1;
```

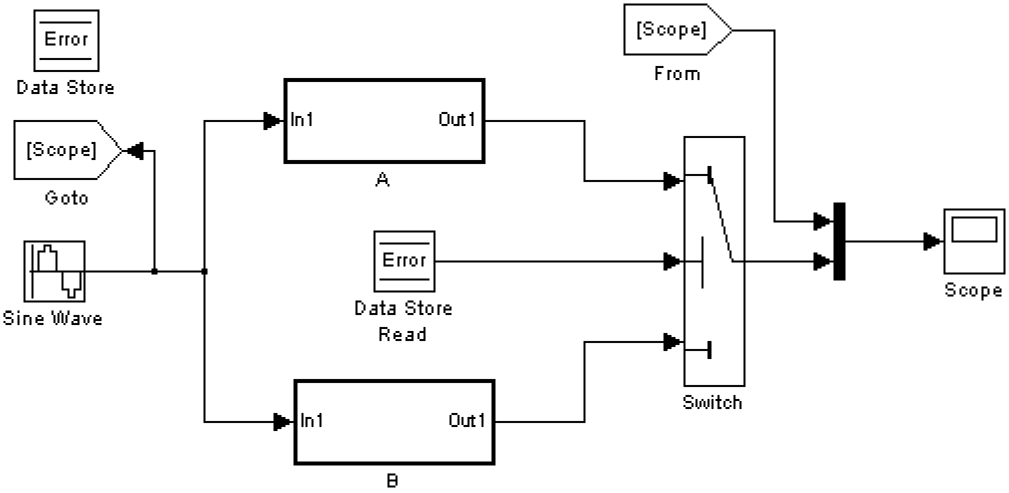
Note A signal object that defines a local store, i.e., that resides in a model workspace, must inherit the value of its `StorageClass` property, i.e., the value must be `auto` (the default). In the case of a signal object that defines a global store, i.e., that resides in the base workspace, the only properties that can inherit their values are `StorageClass` and `SampleTime`. You must specify explicit values for all of the other relevant properties of the object. In either case, when using a signal object to define a data store, you must specify the object's `SamplingMode` as `'Sample based'`.

Accessing Data Stores

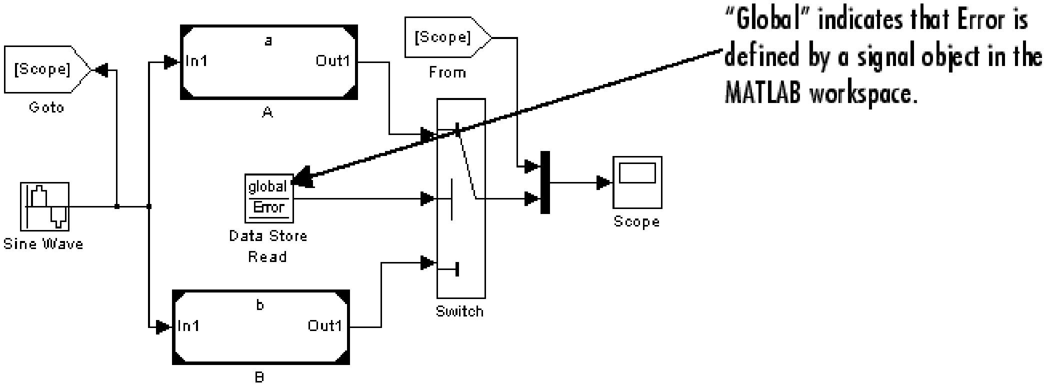
To set the value of a data store at each time step, create an instance of a Data Store Write block at the level of your model that computes the value, set its **Data store name** parameter to the name of the data store to be updated, and connect the output of the block that computes the value to the input of the Data Store Write block, e.g.,



To get the value of a data store at each time step, create an instance of a Data Store Read block at the level of your model that needs the value, set the block's **Data store name** parameter to the name of the data store to be read, and connect the output of the data store read block to the input of the block that need's the data store's value, e.g.,



When connected to a global data store, a data store access block displays the word Global above the data store's name.



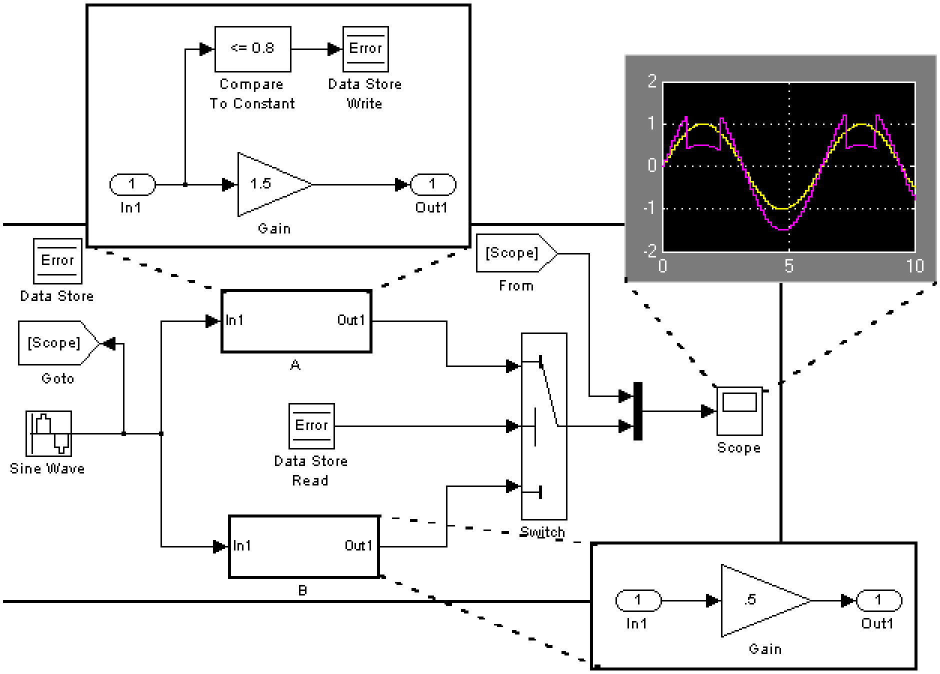
This is done to remind you that the data store is defined by a signal object in the MATLAB workspace rather than by a Data Store Memory block.

Data Store Examples

The following examples illustrate the use of these constructs to define and access data stores.

Local Data Store Example

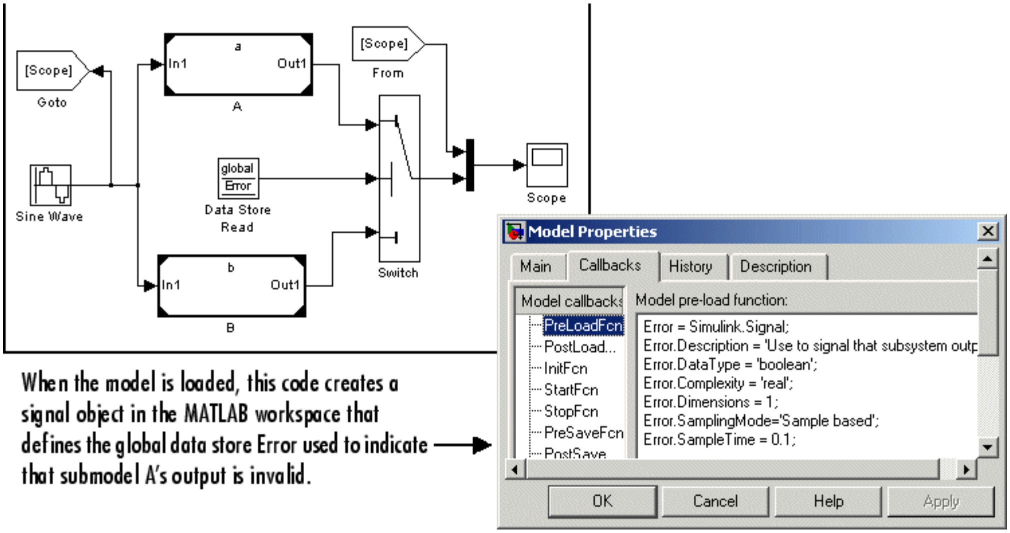
The following model illustrates creation and access of a local data store, i.e., a data store that is visible only in a model or particular subsystem.



This model uses a data store to permit subsystem A to signal that its output is invalid. If subsystem A's output is invalid, the model uses the output of subsystem B.

Global Data Store Example

The following model replaces the subsystems of the previous example with functionally identical submodels to illustrate use of a global data store to share data in a model reference hierarchy.



Model pre-load function:

```
Error = Simulink.Signal;
Error.Description = 'Use to signal that subsystem outp
Error.DataType = 'boolean';
Error.Complexity = 'real';
Error.Dimensions = 1;
Error.SamplingMode='Sample based';
Error.SampleTime = 0.1;
```

When the model is loaded, this code creates a signal object in the MATLAB workspace that defines the global data store Error used to indicate that submodel A's output is invalid.

In this example, the top model uses a signal object in the MATLAB workspace to define the error data store. This is necessary because data stores are visible across model boundaries only if they are defined by signal objects in the MATLAB workspace.

Consulting Model Advisor

Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation of the system represented by the model or generation of inefficient code from the model. It produces a report that lists all the suboptimal conditions or settings that it finds, suggesting better model configuration settings where appropriate. See the following topics for more information:

- "Launching Model Advisor" on page 3-121
- "Model Advisor Window" on page 3-122
- "Navigating Model Advisor Checks" on page 3-123
- "Checking Code-Generation Targets" on page 3-127
- "Model Advisor Demo" on page 3-127
- "Running Model Advisor Programmatically" on page 3-128

Launching Model Advisor

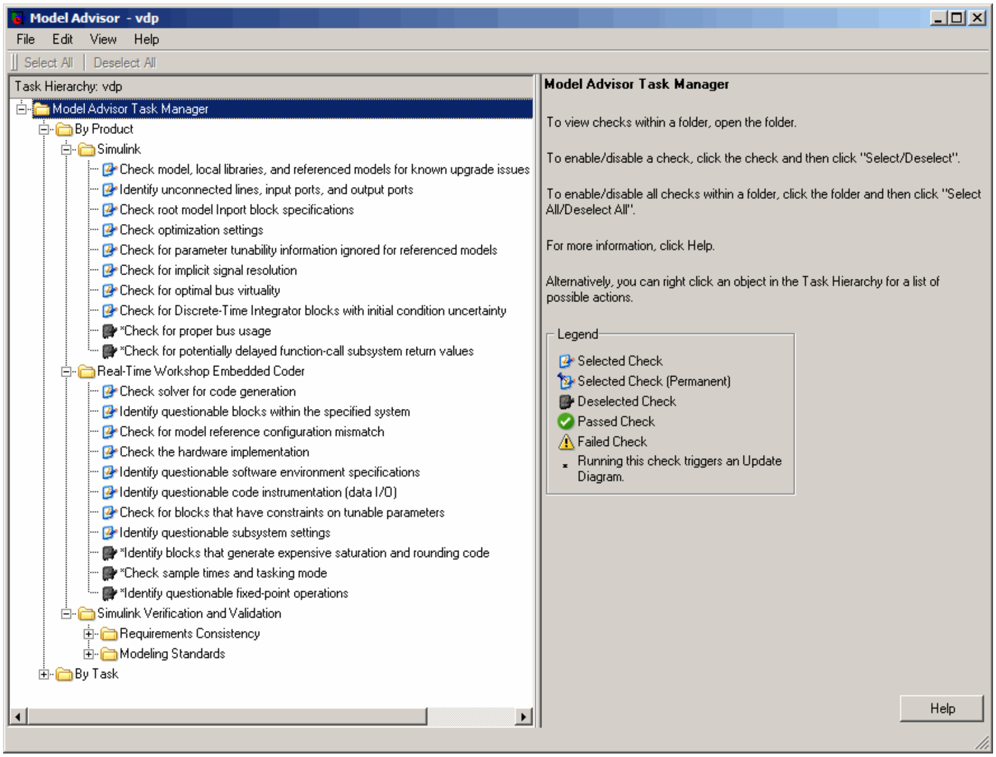
You can use any of the following methods to launch Model Advisor.

- Select **Model Advisor** from Model Editor's **Tools** menu.
- In the **Contents** pane of the Model Explorer (see "The Model Explorer" on page 10-2), select `Advice for` **`model`**, where **`model`** is the name of the model that you want to check.
- At the MATLAB prompt, enter `modeladvisor(`**`model`**`)`, where **`model`** is a handle or path of the model or subsystem you want to check (see `modeladvisor` for more information).
- Select **Model Advisor** from the context (right-click) menu of a subsystem that you want to check.

Note Model Advisor uses the Simulink project (`slprj`) directory (see "Project Directories" on page 3-80 for more information) in the current directory to store reports and other information. If such a directory does not exist in the current directory, Model Advisor creates it. For this reason, you should, before launching Model Advisor, ensure that the current directory is writable. If the directory is not writable, Simulink displays an error message when you attempt to launch Model Advisor.

Model Advisor Window

When you launch Model Advisor, the Model Advisor window is displayed. In the example below, Model Advisor is displaying checks for the `vdp` demo model.



Task Hierarchy: vdp

- Model Advisor Task Manager
  - By Product
    - Simulink
      - Check model, local libraries, and referenced models for known upgrade issues
      - Identify unconnected lines, input ports, and output ports
      - Check root model Inport block specifications
      - Check optimization settings
      - Check for parameter tunability information ignored for referenced models
      - Check for implicit signal resolution
      - Check for optimal bus virtuality
      - Check for Discrete-Time Integrator blocks with initial condition uncertainty
      - *Check for proper bus usage
      - *Check for potentially delayed function-call subsystem return values
    - Real-Time Workshop Embedded Coder
      - Check solver for code generation
      - Identify questionable blocks within the specified system
      - Check for model reference configuration mismatch
      - Check the hardware implementation
      - Identify questionable software environment specifications
      - Identify questionable code instrumentation (data I/O)
      - Check for blocks that have constraints on tunable parameters
      - Identify questionable subsystem settings
      - *Identify blocks that generate expensive saturation and rounding code
      - *Check sample times and tasking mode
      - *Identify questionable fixed-point operations
    - Simulink Verification and Validation
      - Requirements Consistency
      - Modeling Standards
  - By Task

Model Advisor Task Manager

To view checks within a folder, open the folder.

To enable/disable a check, click the check and then click "Select/Deselect".

To enable/disable all checks within a folder, click the folder and then click "Select All/Deselect All".

For more information, click Help.

Alternatively, you can right click an object in the Task Hierarchy for a list of possible actions.

Legend

- Selected Check
- Selected Check (Permanent)
- Deselected Check
- ✓ Passed Check
- ⚠ Failed Check
- * Running this check triggers an Update Diagram.

The left pane lists the checks that Model Advisor performs. By default, Model Advisor groups the checks by product. Select **By Task** to display checks related to specific tasks, such as updating the model to be compatible with the current Simulink version.

The right pane provides instructions on how to view, enable, and disable checks, and provides a legend explaining the displayed symbols.

You can select some or all of the checks and then perform an individual check or all selected checks. The results of the checks will be displayed in the Model Advisor window. Additionally, you can opt to generate an HTML report of the check results that will be displayed in a separate browser window.

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

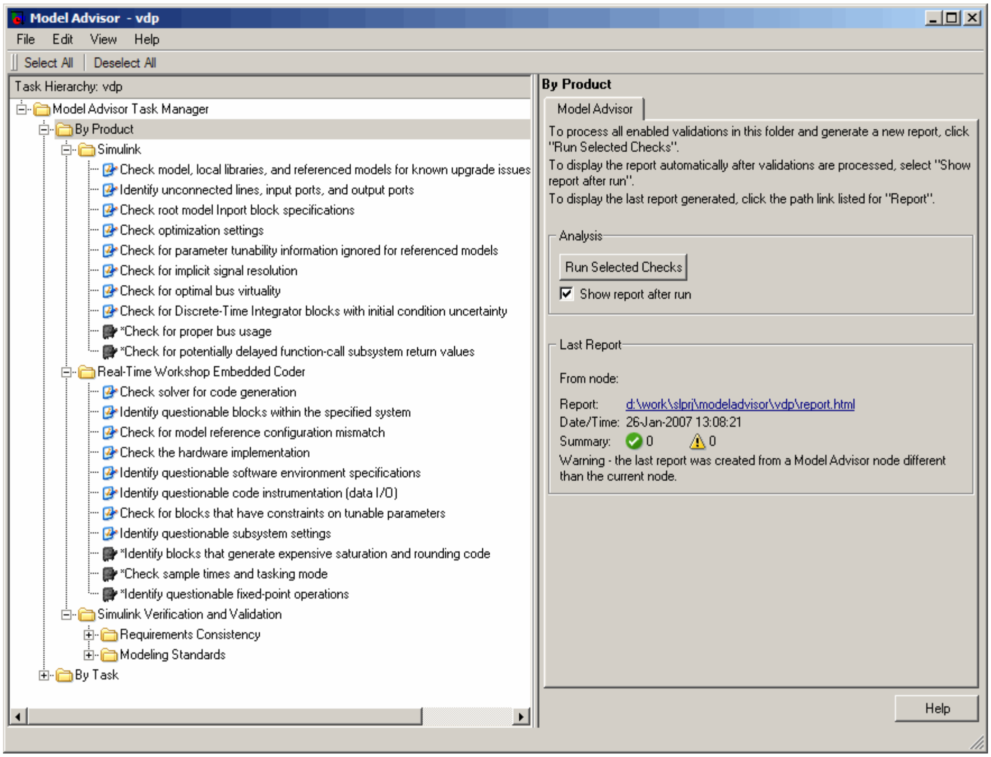
Navigating Model Advisor Checks

The following procedure demonstrates how to use Model Advisor to perform checks on your model and view the check results.

1 Open the `vdp` demo model, for example, by entering `vdp` on the MATLAB command line.

2 Open Model Advisor, for example, by selecting **Model Advisor** from Model Editor's **Tools** menu. The Model Advisor window is launched and displays checks for the `vdp` demo model, as shown in "Model Advisor Window" on page 3-122. You can examine which checks are selected and which are not. By default, most of the checks are selected.

3 Select **By Product** in the left pane. This switches the right pane to a **By Product** view, as shown below.



- Model Advisor Task Manager
  - By Product
    - Simulink
      - Check model, local libraries, and referenced models for known upgrade issues
      - Identify unconnected lines, input ports, and output ports
      - Check root model Inport block specifications
      - Check optimization settings
      - Check for parameter tunability information ignored for referenced models
      - Check for implicit signal resolution
      - Check for optimal bus virtuality
      - Check for Discrete-Time Integrator blocks with initial condition uncertainty
      - *Check for proper bus usage
      - *Check for potentially delayed function-call subsystem return values
    - Real-Time Workshop Embedded Coder
      - Check solver for code generation
      - Identify questionable blocks within the specified system
      - Check for model reference configuration mismatch
      - Check the hardware implementation
      - Identify questionable software environment specifications
      - Identify questionable code instrumentation (data I/O)
      - Check for blocks that have constraints on tunable parameters
      - Identify questionable subsystem settings
      - *Identify blocks that generate expensive saturation and rounding code
      - *Check sample times and tasking mode
      - *Identify questionable fixed-point operations
    - Simulink Verification and Validation
      - Requirements Consistency
      - Modeling Standards
  - By Task

To process all enabled validations in this folder and generate a new report, click "Run Selected Checks".
To display the report automatically after validations are processed, select "Show report after run".
To display the last report generated, click the path link listed for "Report".

Analysis

☑ Show report after run

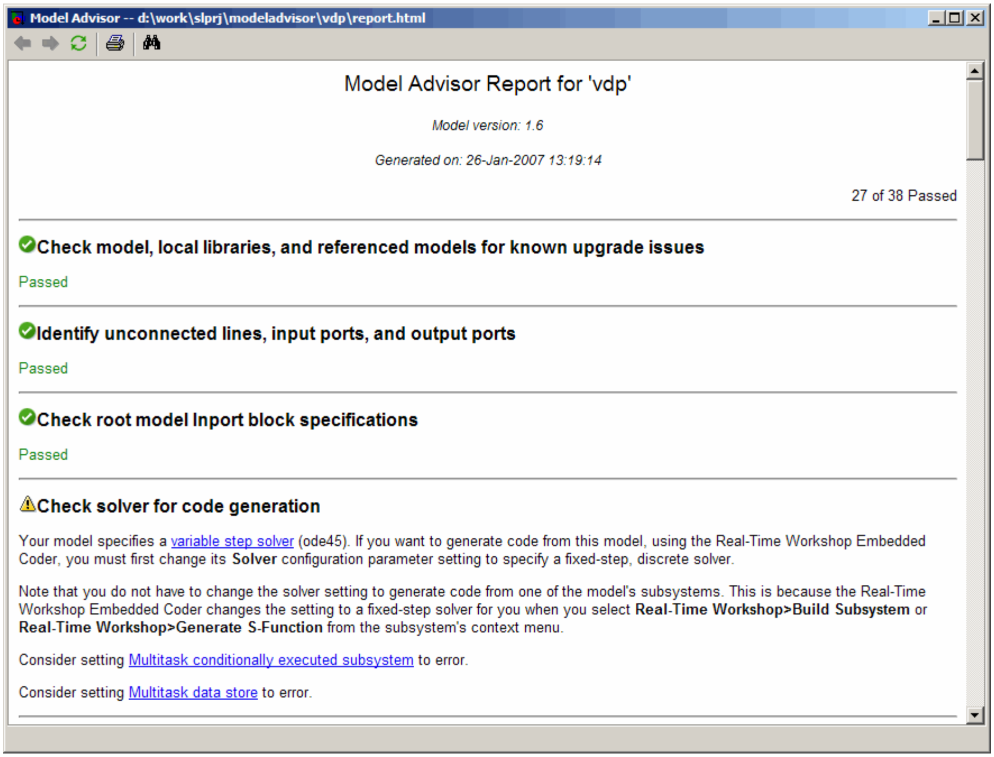
Last Report

From node:

Report: d:\work\slprj\modeladvisor\vdp\report.html
Date/Time: 26-Jan-2007 13:08:21
Summary: ✓ 0 ⚠ 0
Warning - the last report was created from a Model Advisor node different than the current node.

Select the option **Show report after run**. This will cause an HTML report of check results to be generated and displayed after the checks run.

4 Run the selected checks by clicking the button **Run Selected Checks**. After the checks run, an HTML report of the check results is displayed in a browser window, as shown below.



Model Advisor Report for 'vdp'

Model version: 1.6

Generated on: 26-Jan-2007 13:19:14

27 of 38 Passed

✓Check model, local libraries, and referenced models for known upgrade issues

Passed

✓Identify unconnected lines, input ports, and output ports

Passed

✓Check root model Inport block specifications

Passed

⚠Check solver for code generation

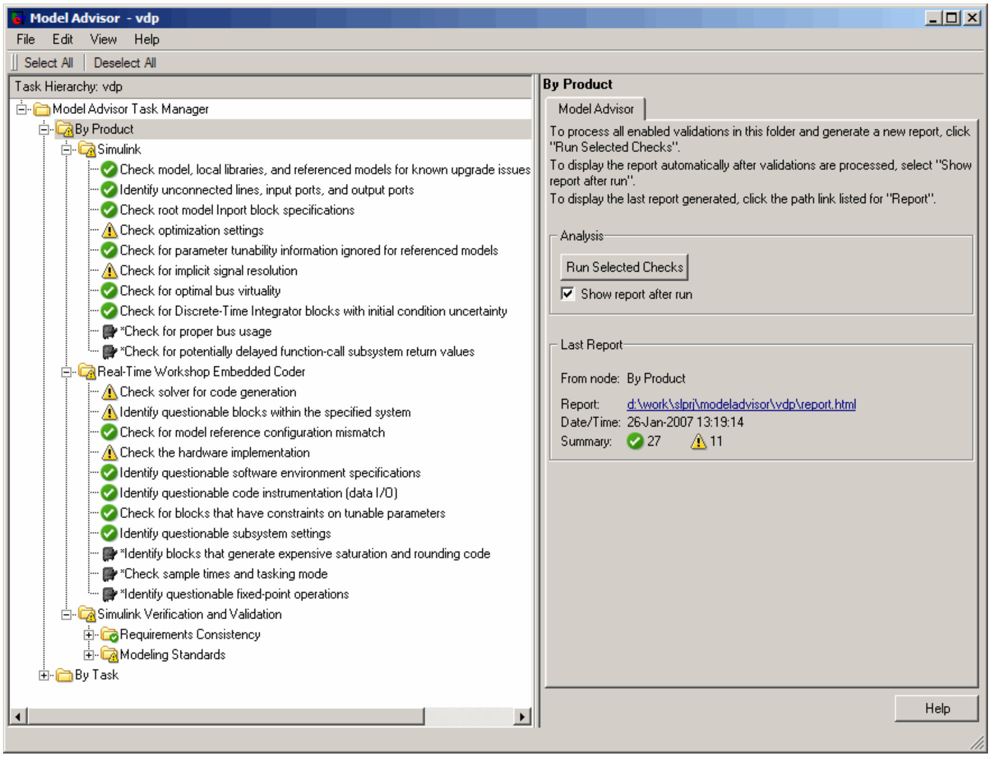
Your model specifies a variable step solver (ode45). If you want to generate code from this model, using the Real-Time Workshop Embedded Coder, you must first change its **Solver** configuration parameter setting to specify a fixed-step, discrete solver.

Note that you do not have to change the solver setting to generate code from one of the model's subsystems. This is because the Real-Time Workshop Embedded Coder changes the setting to a fixed-step solver for you when you select **Real-Time Workshop>Build Subsystem** or **Real-Time Workshop>Generate S-Function** from the subsystem's context menu.

Consider setting Multitask conditionally executed subsystem to error.

Consider setting Multitask data store to error.

5 Return to the Model Advisor window, which has been updated with the check results, as shown below.



Task Hierarchy: vdp

- Model Advisor Task Manager
  - By Product
    - Simulink
      - ✓ Check model, local libraries, and referenced models for known upgrade issues
      - ✓ Identify unconnected lines, input ports, and output ports
      - ✓ Check root model Inport block specifications
      - ⚠ Check optimization settings
      - ✓ Check for parameter tunability information ignored for referenced models
      - ⚠ Check for implicit signal resolution
      - ✓ Check for optimal bus virtuality
      - ✓ Check for Discrete-Time Integrator blocks with initial condition uncertainty
      - *Check for proper bus usage
      - *Check for potentially delayed function-call subsystem return values
    - Real-Time Workshop Embedded Coder
      - ⚠ Check solver for code generation
      - ⚠ Identify questionable blocks within the specified system
      - ✓ Check for model reference configuration mismatch
      - ⚠ Check the hardware implementation
      - ✓ Identify questionable software environment specifications
      - ✓ Identify questionable code instrumentation (data I/O)
      - ✓ Check for blocks that have constraints on tunable parameters
      - ✓ Identify questionable subsystem settings
      - *Identify blocks that generate expensive saturation and rounding code
      - *Check sample times and tasking mode
      - *Identify questionable fixed-point operations
    - Simulink Verification and Validation
      - Requirements Consistency
      - Modeling Standards
  - By Task

By Product

To process all enabled validations in this folder and generate a new report, click "Run Selected Checks".
To display the report automatically after validations are processed, select "Show report after run".
To display the last report generated, click the path link listed for "Report".

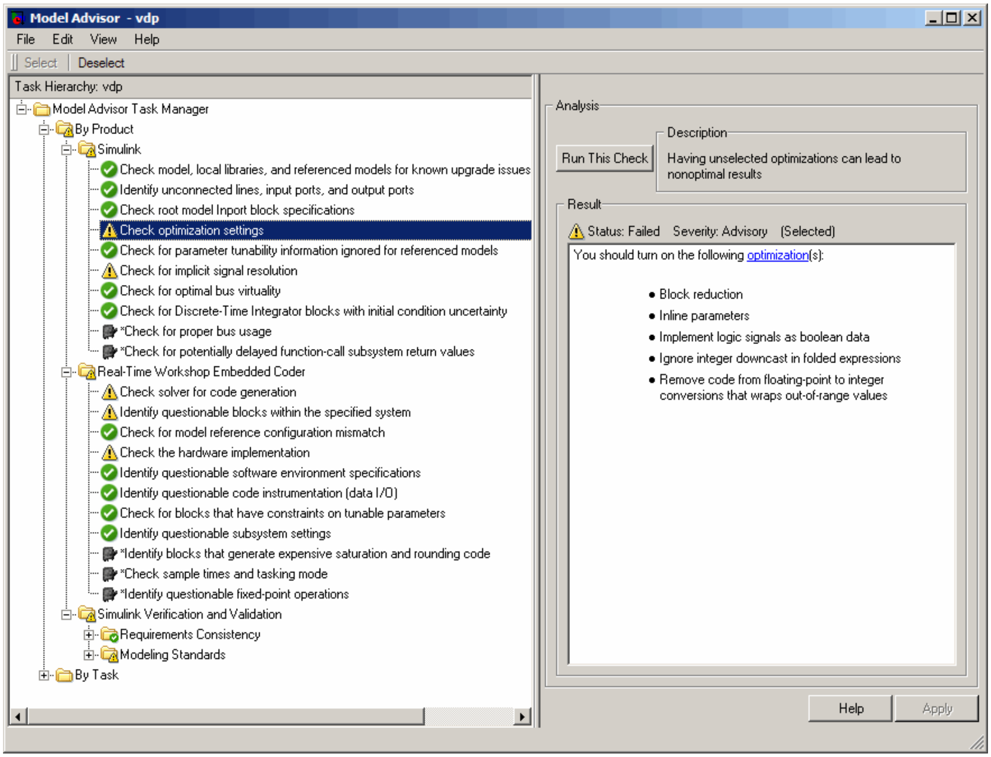
Analysis

☑ Show report after run

Last Report

From node: By Product
Report: d:\work\slprj\modeladvisor\vdp\report.html
Date/Time: 26-Jan-2007 13:19:14
Summary: ✓ 27 ⚠ 11

6 You can select an individual check to open a detailed view of the check in the right pane. For example, selecting the check **Check optimization settings** switches the right pane to the view shown below. You can use this view to examine and exercise a check individually.



Task Hierarchy: vdp

- Model Advisor Task Manager
  - By Product
    - Simulink
      - ✓ Check model, local libraries, and referenced models for known upgrade issues
      - ✓ Identify unconnected lines, input ports, and output ports
      - ✓ Check root model Inport block specifications
      - ⚠ Check optimization settings
      - ✓ Check for parameter tunability information ignored for referenced models
      - ⚠ Check for implicit signal resolution
      - ✓ Check for optimal bus virtuality
      - ✓ Check for Discrete-Time Integrator blocks with initial condition uncertainty
      - *Check for proper bus usage
      - *Check for potentially delayed function-call subsystem return values
    - Real-Time Workshop Embedded Coder
      - ⚠ Check solver for code generation
      - ⚠ Identify questionable blocks within the specified system
      - ✓ Check for model reference configuration mismatch
      - ⚠ Check the hardware implementation
      - ✓ Identify questionable software environment specifications
      - ✓ Identify questionable code instrumentation (data I/O)
      - ✓ Check for blocks that have constraints on tunable parameters
      - ✓ Identify questionable subsystem settings
      - *Identify blocks that generate expensive saturation and rounding code
      - *Check sample times and tasking mode
      - *Identify questionable fixed-point operations
    - Simulink Verification and Validation
      - Requirements Consistency
      - Modeling Standards
  - By Task

Analysis

Description

Having unselected optimizations can lead to nonoptimal results

Result

⚠ Status: Failed Severity: Advisory (Selected)

You should turn on the following optimization(s):

- Block reduction
- Inline parameters
- Implement logic signals as boolean data
- Ignore integer downcast in folded expressions
- Remove code from floating-point to integer conversions that wraps out-of-range values

Checking Code-Generation Targets

Before running Model Advisor on a model, select the target you plan to use in the **Real-Time Workshop** pane of the **Configuration Parameters** dialog box (see “Configuration Parameters Dialog Box” on page 11-66). Model Advisor works most effectively with ERT and ERT-based targets (targets based on the Real-Time Workshop Embedded Coder).

Model Advisor Demo

Select the following link `sldemo_mdladv` or enter `sldemo_mdladv` at the MATLAB command line to run a demo that illustrates usage of Model Advisor:

If Real-Time Workshop is installed on your system, select the following links to illustrate usage of Model Advisor

- rtwdemo_advisor1
- rtwdemo_advisor2
- rtwdemo_advisor3

You can also run these demos from the MATLAB command line. For example, the command

```
modeladvisor('rtwdemo_advisor1')
```

launches the rtwdemo_advisor1 model. Note that demo models rtwdemo_advisor2 and rtwdemo_advisor3 require Stateflow and Fixed-Point Toolbox.

Running Model Advisor Programmatically

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

Managing Model Versions

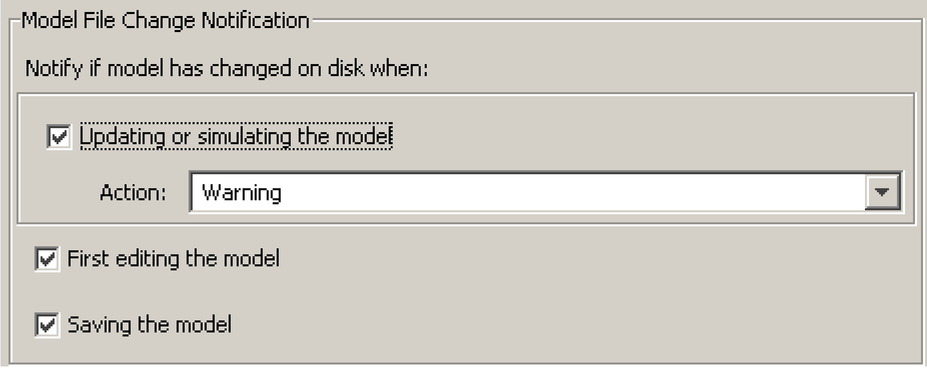
Simulink has features that help you to manage multiple versions of a model.

- Model File Change Notification helps you manage work with source control operations and multiple users. See "Model File Change Notification" on page 3-129.
- 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 3-141 for more information.
- The Simulink **Model Properties** 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 3-133).
- 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 "Source Control Interface" in the online MATLAB documentation for more information.

Model File Change Notification

You can use the Simulink Preferences to specify whether to notify if the model has changed on disk when updating, simulating, editing, or saving the model. This can occur, for example, with source control operations and multiple users.

To access the Simulink **Preferences** dialog box, select **File > Preferences** in Simulink or MATLAB, then select **Simulink** in the left pane.



Model File Change Notification

Notify if model has changed on disk when:

☑ Updating or simulating the model

Action: Warning

☑ First editing the model

☑ Saving the model

The Model File Change Notification options are in the right pane. You can use the three independent options as follows:

- If you select the **Updating or simulating the model** check box, you can choose what form of notification you want from the **Action** list:
  - `Warning` — in the MATLAB command window.
  - `Error` — in the MATLAB command window if simulating from the command line, or if simulating from a menu item, in the Simulation Diagnostics window.
  - `Reload model (if unmodified)` — if the model is modified, you see the prompt dialog. If unmodified, the model is reloaded.
  - `Show prompt dialog` — in the dialog, you can choose to close and reload, or ignore the changes.
- If you select the **First editing the model** check box, and the file has changed on disk, and the block diagram is unmodified in Simulink:
  - Any command-line operation that causes the block diagram to be modified (e.g., a call to `set_param`) will result in a warning:

    ```
    Warning: Block diagram 'mymodel' is being edited but file has
    changed on disk since it was loaded.  You should close and
    reload the block diagram.
    ```

  - Any graphical operation that modifies the block diagram (e.g., adding a block) causes a warning dialog to appear.

- If you select the **Saving the model** check box, and the file has changed on disk:

  - The `save_system` function displays an error, unless the `OverwriteIfChangedOnDisk` option is used.

  - Saving the model by using the menu (**File > Save**) or a keyboard shortcut causes a dialog to be shown. In the dialog, you can choose to overwrite, save with a new name, or cancel the operation.

Specifying the Current User

When you create or update a model, 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.

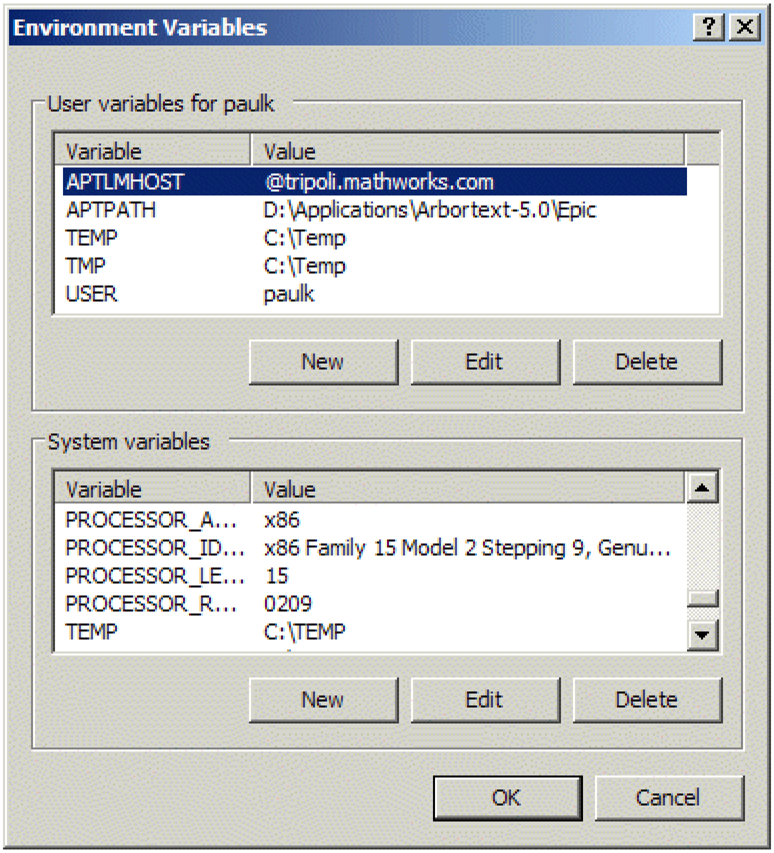
UNIX systems define the `USER` environment variable and set its value to the name you use to log on to your system. Thus, if you are using a UNIX system, you do not have to do anything to enable Simulink to identify you as the current user.

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



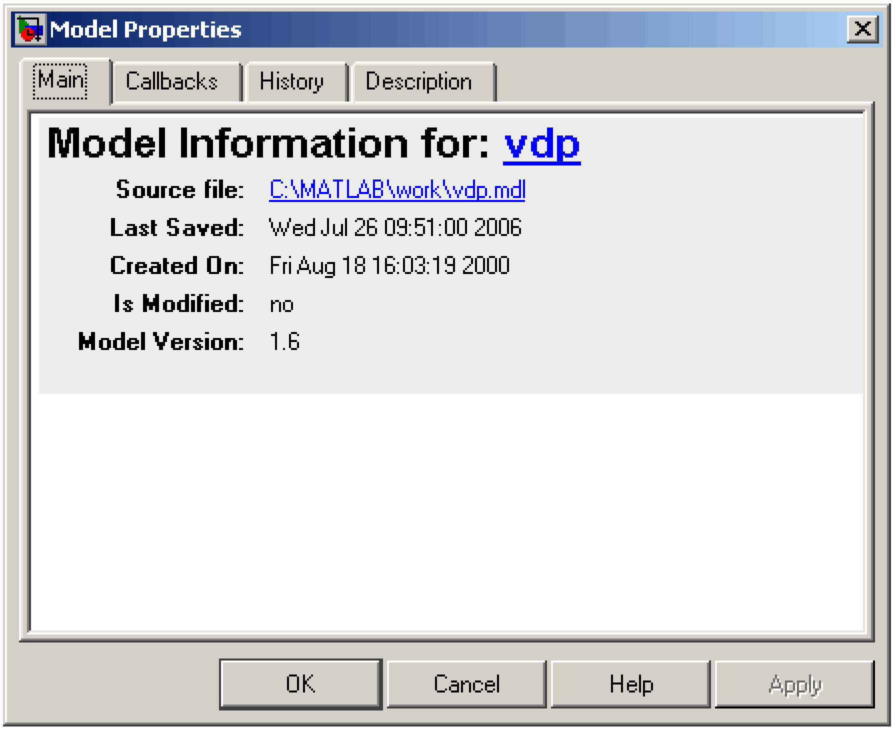
Variable	Value
APTLMHOST	@tripoli.mathworks.com
APTPATH	D:\Applications\Arbortext-5.0\Epic
TEMP	C:\Temp
TMP	C:\Temp
USER	paulk

Variable	Value
PROCESSOR_A...	x86
PROCESSOR_ID...	x86 Family 15 Model 2 Stepping 9, Genu...
PROCESSOR_LE...	15
PROCESSOR_R...	0209
TEMP	C:\TEMP

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:	C:\MATLAB\work\vdp.mdl
Last Saved:	Wed Jul 26 09:51:00 2006
Created On:	Fri Aug 18 16:03:19 2000
Is Modified:	no
Model Version:	1.6

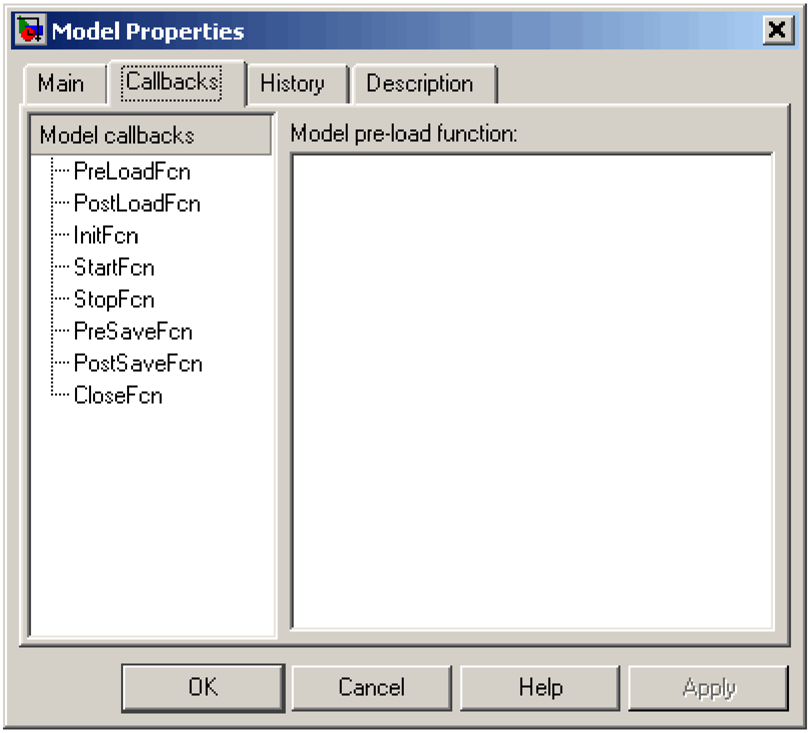
The dialog box includes the following panes.

Main Pane

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

Callbacks Pane

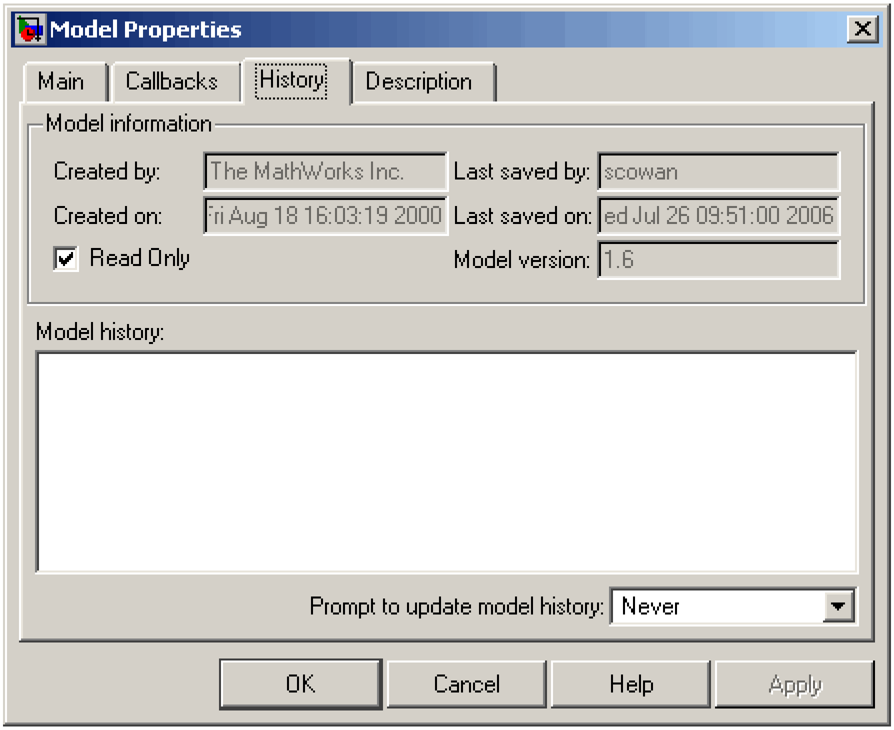
The **Callbacks** pane lets you specify functions to be invoked by Simulink at specific points in the simulation of the model.



In the left pane, select the callback. In the right pane, enter the name of the function you want to be invoked for the selected callback. See "Creating Model Callback Functions" on page 3-100 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: scowan

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

Last saved on: ed Jul 26 09:51:00 2006

☑ Read Only

Model version: 1.6

Prompt to update model history: Never

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

Model Information Controls

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

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

- **Created by**

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

- **Created on**

  Date and time this model was created.

- **Last saved by**

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

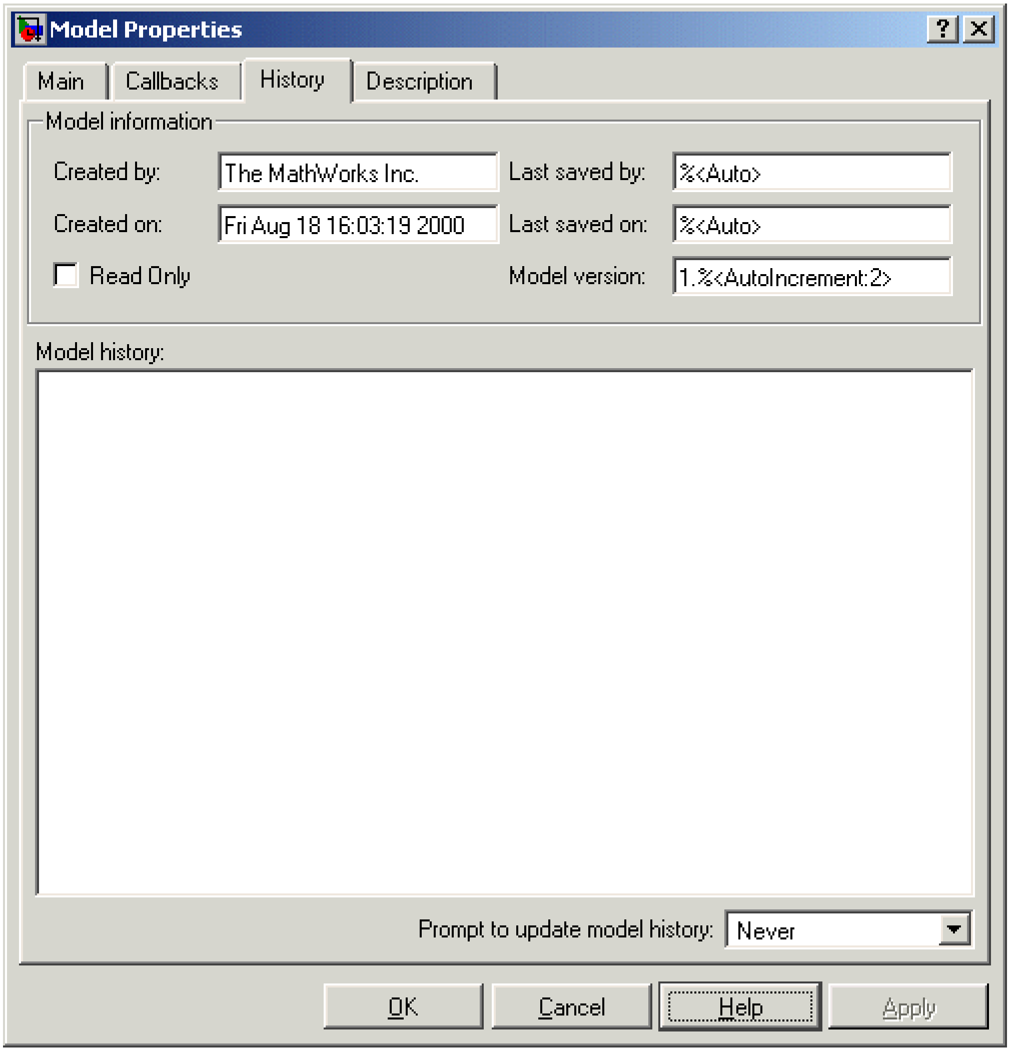
- **Last saved on**

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

- **Model version**

  Version number for this model.

Read Only Check Box Deselected. When **Read Only** is deselected, the dialog box shows the format strings or values for the following fields. You can edit all but the **Created on** field, as described.



Created by: The MathWorks Inc.

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

□ Read Only

Last saved by: %<Auto>

Last saved on: %<Auto>

Model version: 1.%<AutoIncrement:2>

Prompt to update model history: Never

- **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. Do not edit this field.

- **Last saved by**

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

- **Last saved on**

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

- **Model version**

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

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

  as

  ```
  1.2
  ```

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

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

  becomes

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

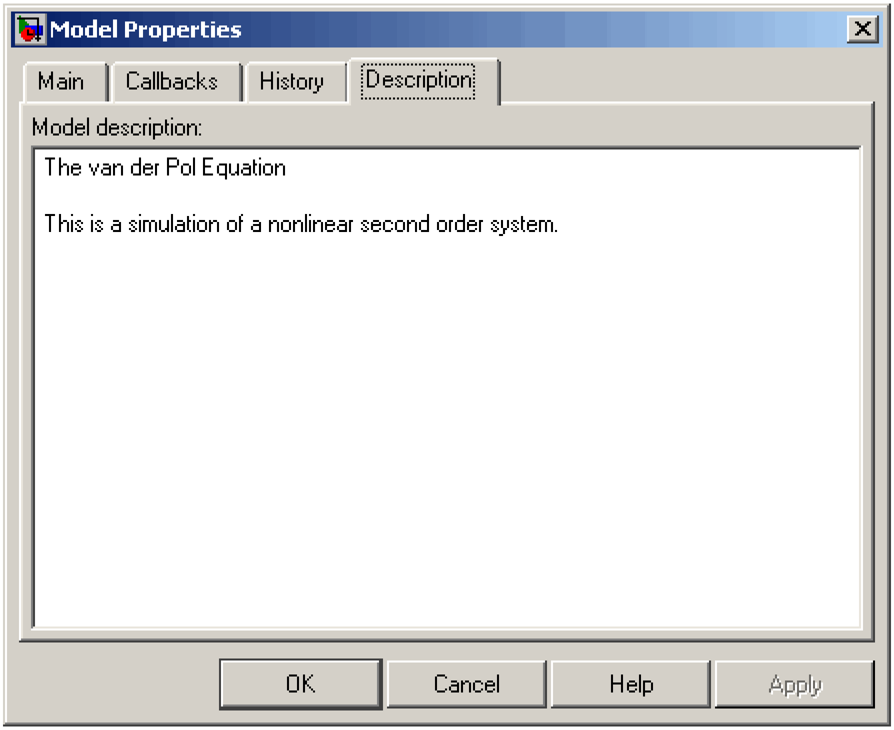
  and Simulink reports the model version number as `1.3`.

Model History Controls

The model history controls group contains a scrollable text field and an option list. The text field displays the history for the model in a scrollable text field. To change the model history, edit the contents of this field. The option list allows you to enable or disable the Simulink 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 3-140 for more information. To disable the change history feature, select `Never` from the **Prompt to update model history** list.

Model Description Controls

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



Model description:

The van der Pol Equation

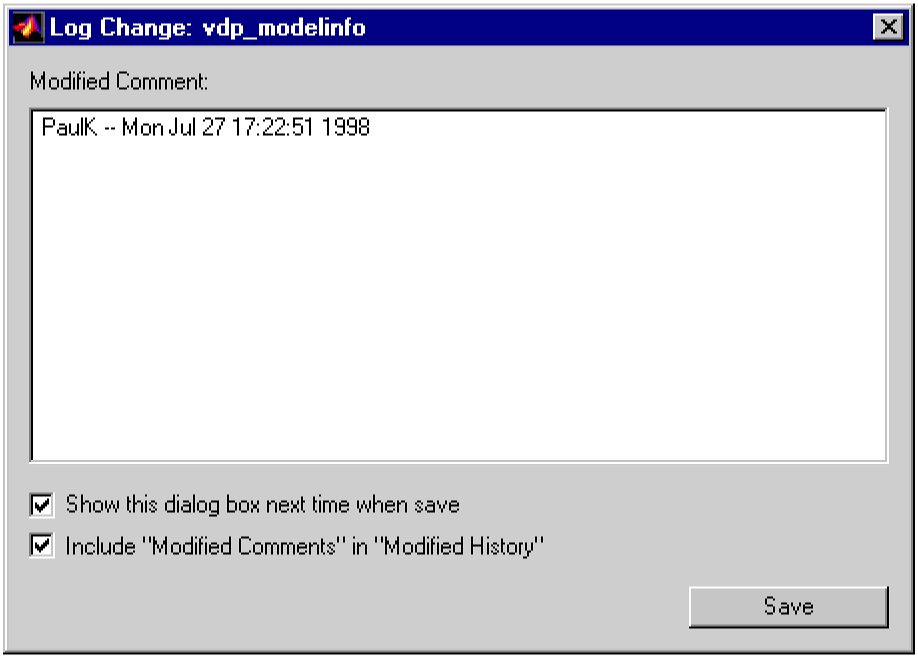
This is a simulation of a nonlinear second order system.

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.

Property	Description
LastModifiedBy	User name of the person who last modified this model.
ModifiedBy	Person who last modified this model.
ModifiedByFormat	Format of the `ModifiedBy` parameter. Value can be any string. The string can include the tag `%<Auto>`. 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.

Model Discretizer

Model Discretizer selectively replaces continuous Simulink blocks with discrete equivalents. Discretization is a critical step in digital controller design and for hardware in-the-loop simulations.

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.

See the following topics for information on model discretizing:

- "Requirements" on page 3-143
- "Discretizing a Model from the Model Discretizer GUI" on page 3-143
- "Viewing the Discretized Model" on page 3-152
- "Discretizing Blocks from the Simulink Model" on page 3-155
- "Discretizing a Model from the MATLAB Command Window" on page 3-166

Requirements

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

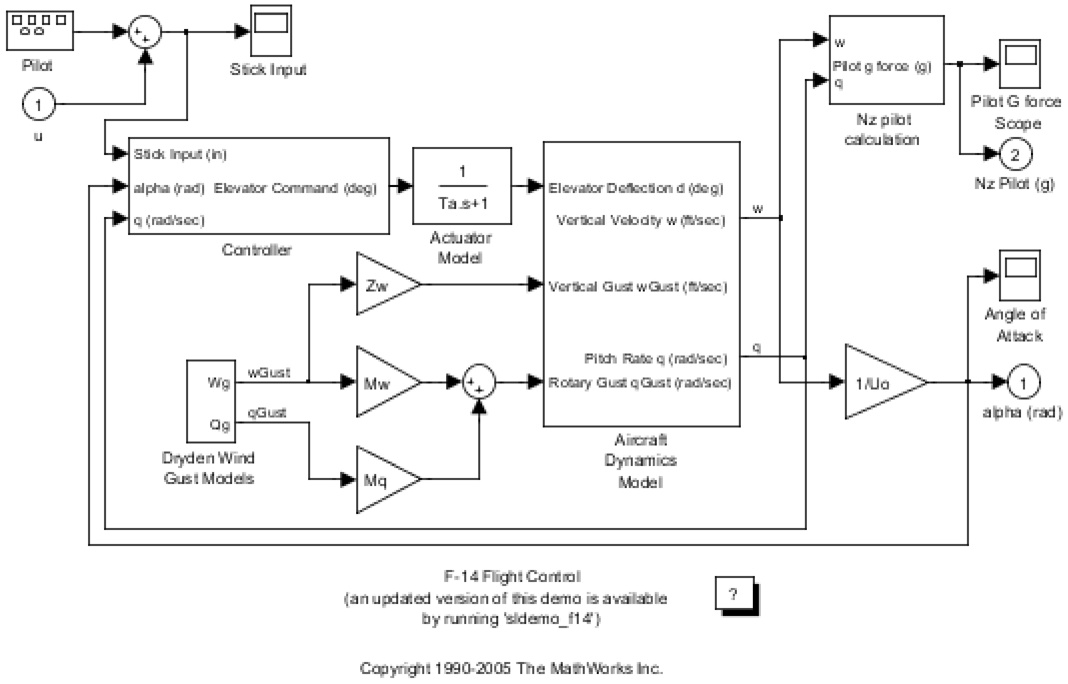
Discretizing a Model from the Model Discretizer GUI

To discretize a model:

- "Start Model Discretizer" on page 3-144
- "Specify the Transform Method" on page 3-146

- “Specify the Sample Time” on page 3-146
- “Specify the Discretization Method” on page 3-147
- “Discretize the Blocks” on page 3-150

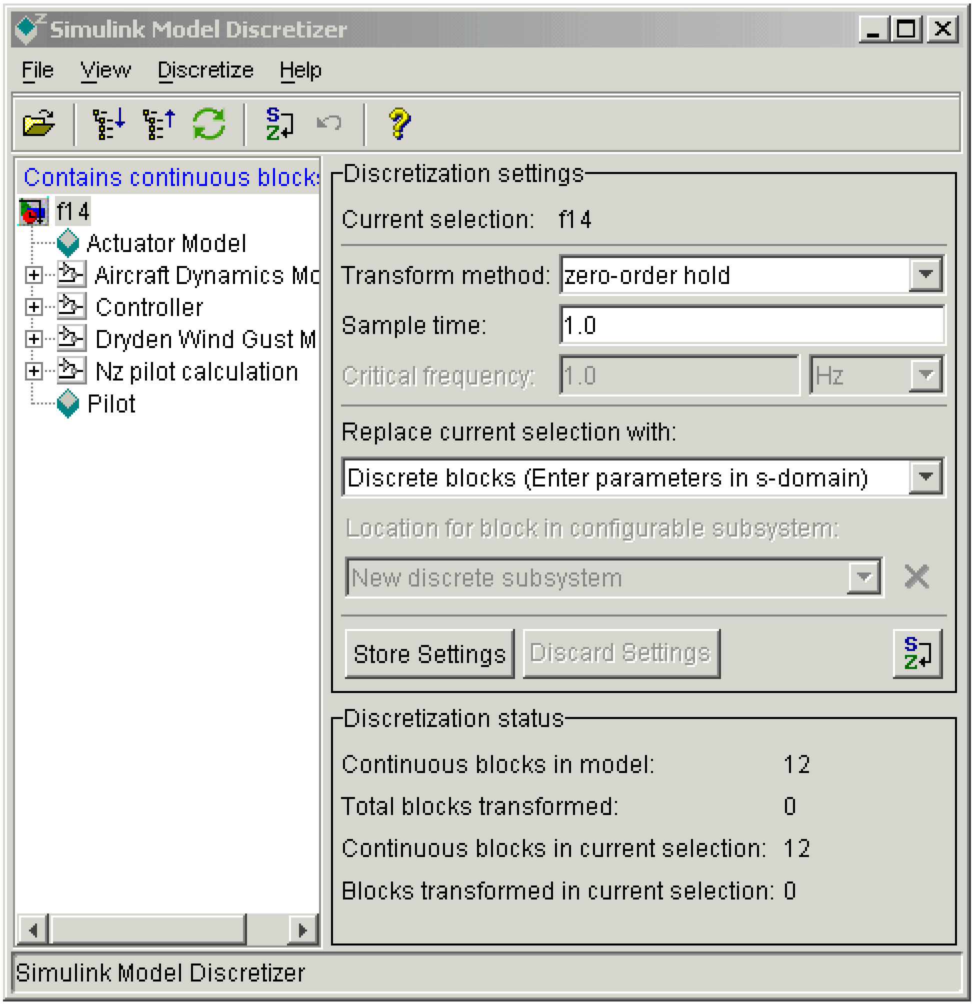
The f14 model, shown below, demonstrates the steps in discretizing a model.



Start Model Discretizer

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

The **Simulink Model Discretizer** appears.



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

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 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 Model Discretizer, select **Load model** from the **File** menu.

Specify the Transform Method

The transform method specifies the type of algorithms used in the discretization. For more information on the different transform methods, see "Linear, Time-Invariant Models" in the Control System Toolbox documentation.

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

- `zero-order hold`

  Zero-order hold on the inputs.

- `first-order hold`

  Linear interpolation of inputs.

- `tustin`

  Bilinear (Tustin) approximation.

- `tustin with prewarping`

  Tustin approximation with frequency prewarping.

- `matched pole-zero`

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

Specify the Sample Time

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

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

You can enter workspace variables when discretizing blocks in the s-domain. See “Discrete blocks (Enter parameters in s-domain)” on page 3-147.

Specify the Discretization Method

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

- “Discrete blocks (Enter parameters in s-domain)” on page 3-147

  Creates a discrete block whose parameters are retained from the corresponding continuous block.

- “Discrete blocks (Enter parameters in z-domain)” on page 3-148

  Creates a discrete block whose parameters are “hard-coded“ values placed directly into the block’s dialog.

- “Configurable subsystem (Enter parameters in s-domain)” on page 3-149

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

- “Configurable subsystem (Enter parameters in z-domain)” on page 3-150

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

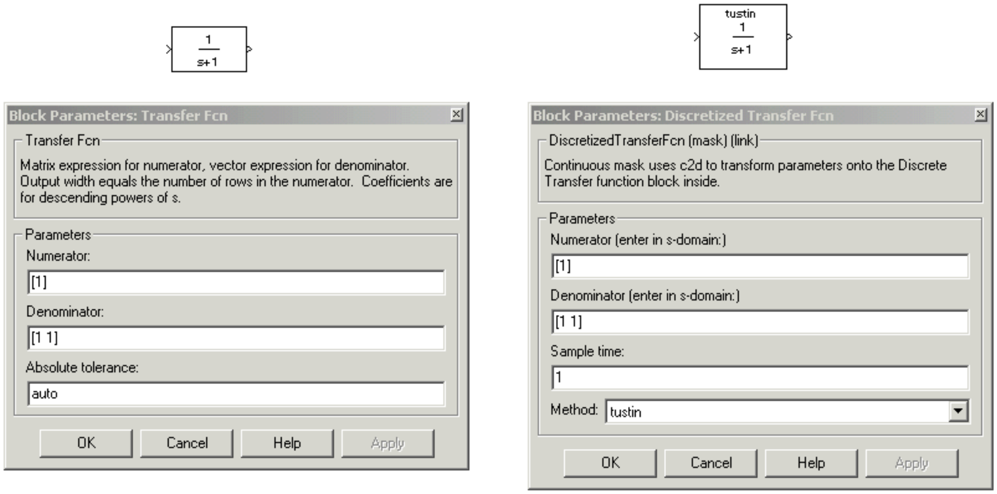
Discrete blocks (Enter parameters in s-domain). Creates a discrete block whose parameters are retained from the corresponding continuous block. The sample time and the discretization parameters are also on the block’s parameter dialog box.

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

These blocks have the unique capability of reverting to continuous behavior if the sample time is changed to zero. Entering the sample time as a workspace variable (`'Ts'`, for example) allows for easy changeover from continuous to discrete and back again. See “Specify the Sample Time” on page 3-146.

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

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



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

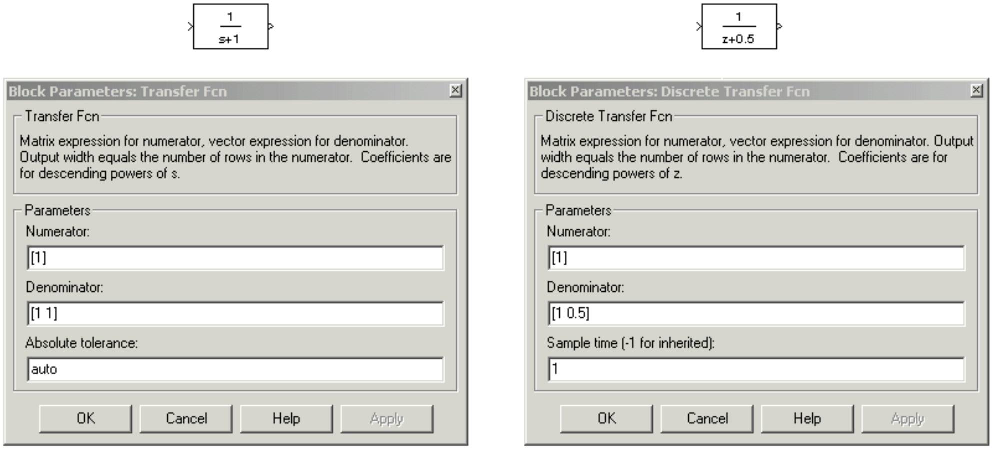
Method: tustin

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

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

```
help c2d
```

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



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 `f14_disc_lib`.

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

You can open a configurable subsystem library by right-clicking on the subsystem in the 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 Model Discretizer by selecting **Load model** from the **File** menu.

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

There are two methods for discretizing blocks.

Select Blocks and Discretize.

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

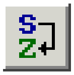
To choose multiple blocks, press and hold the **Ctrl** button on the keyboard while selecting the blocks.

Note You must select blocks from the Model Discretizer tree view. Clicking blocks in the 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**.

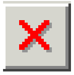
This adds the current block with its discretization settings to the group of preset blocks.

3 Repeat steps 1 and 2, as necessary.

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

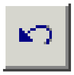
Deleting a Discretization Candidate from a Configurable Subsystem

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.

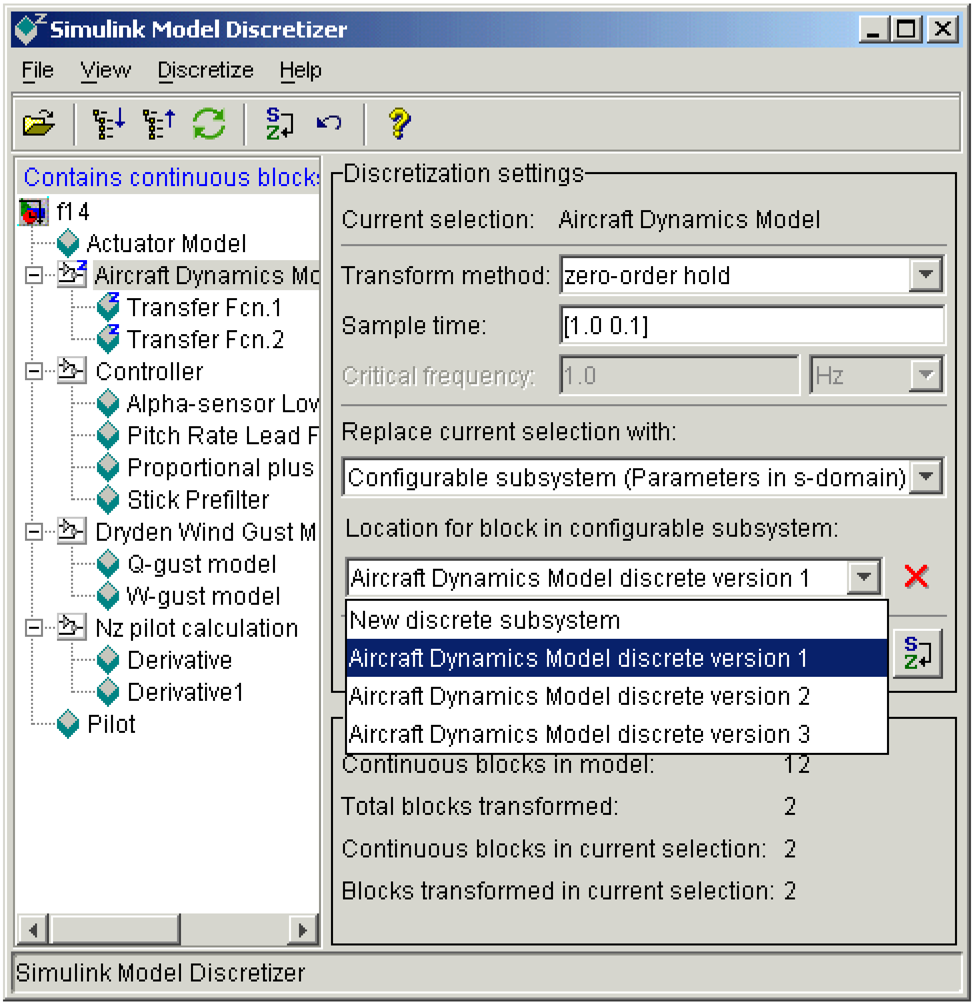
Viewing the Discretized Model

Model Discretizer displays the model in a hierarchical tree view.

Viewing Discretized Blocks

The block's icon in the tree view becomes highlighted with a "**z**" when the block has been discretized.

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



Discretization settings

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

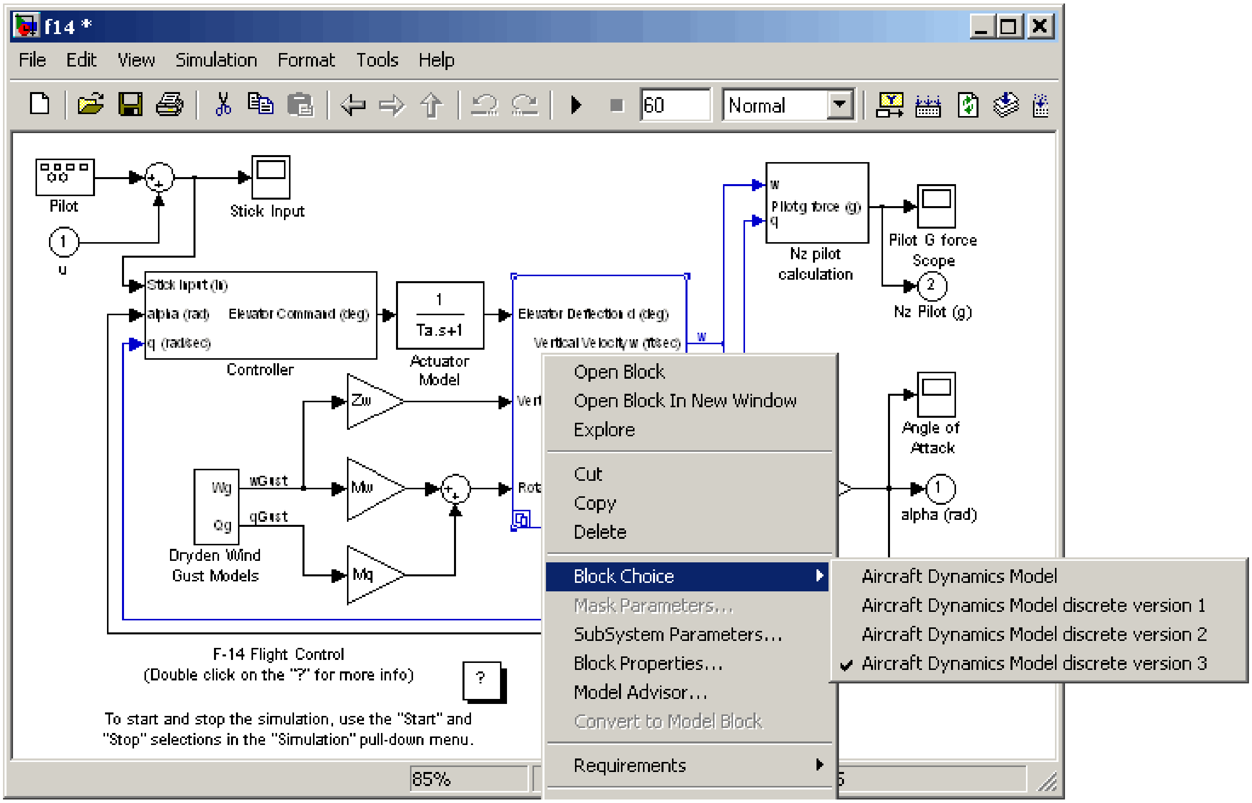
Continuous blocks in current selection: 2

Blocks transformed in current selection: 2

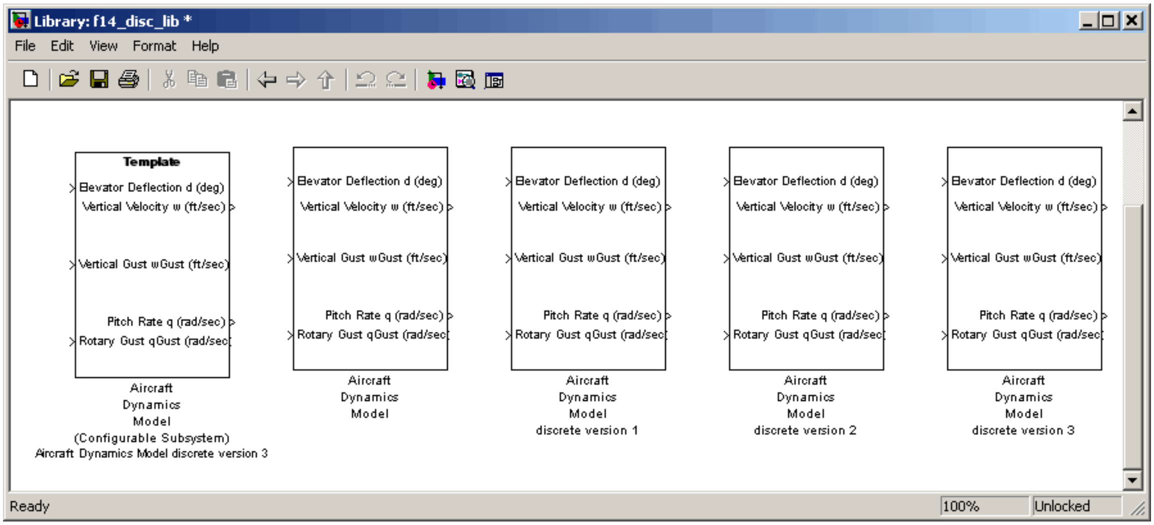
Simulink Model Discretizer

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

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
100%
Unlocked

Refreshing Model Discretizer View of the Model

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

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

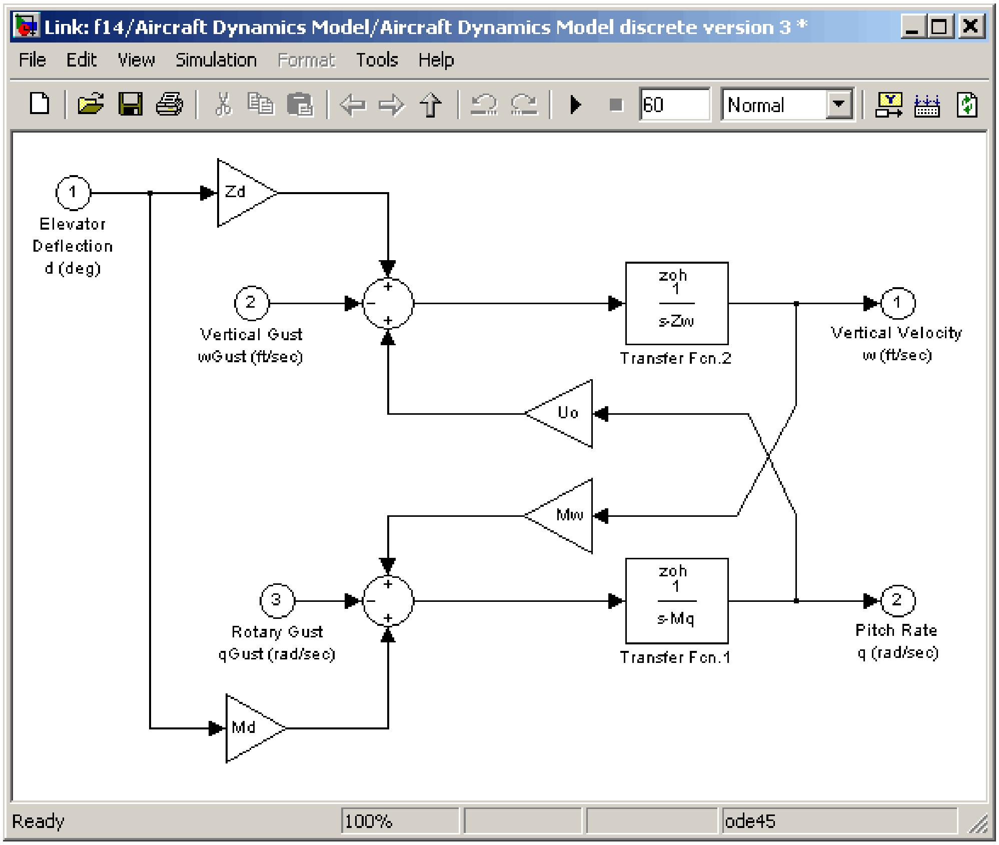
Discretizing Blocks from the Simulink Model

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.

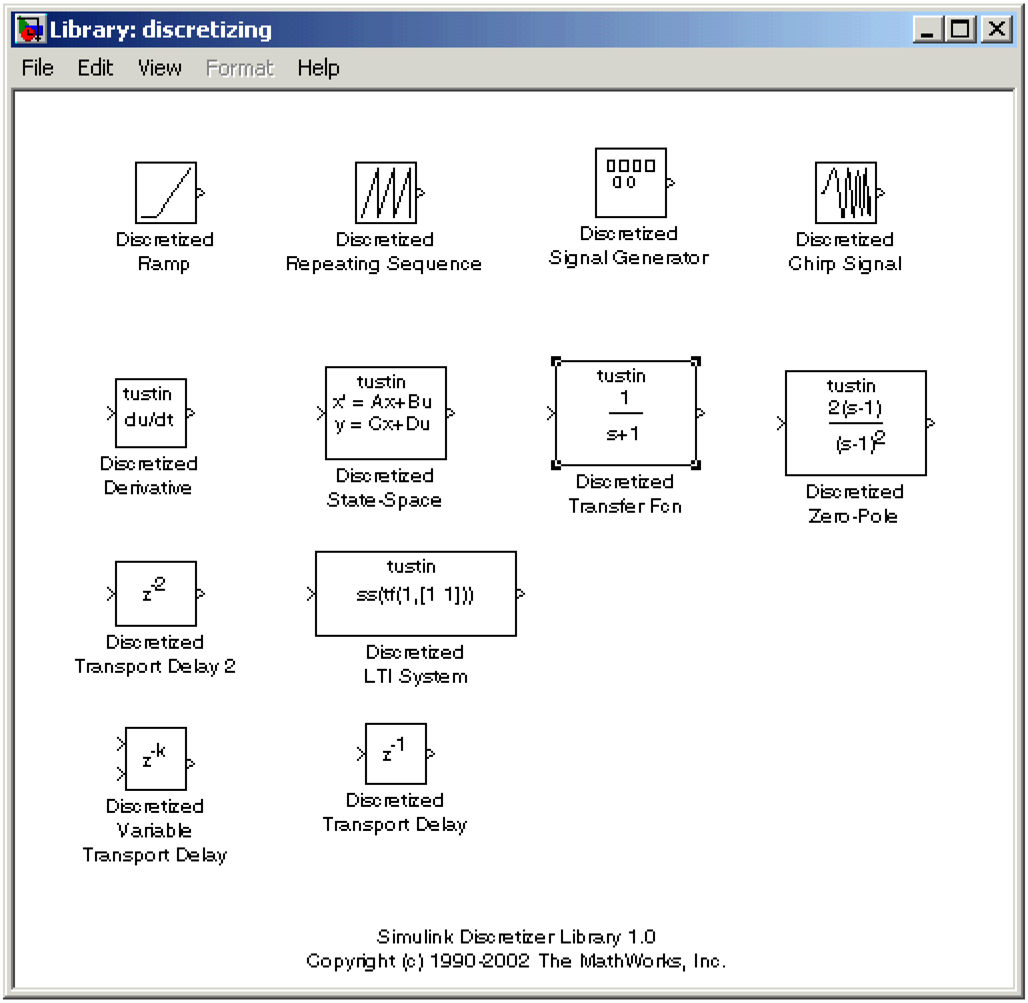
2 Open the Aircraft Dynamics Model subsystem in the `f14` model.


Link: f14/Aircraft Dynamics Model/Aircraft Dynamics Model discrete version 3 *
File
Edit
View
Simulation
Format
Tools
Help
60
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
+
-
+
zoh
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
zoh
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.

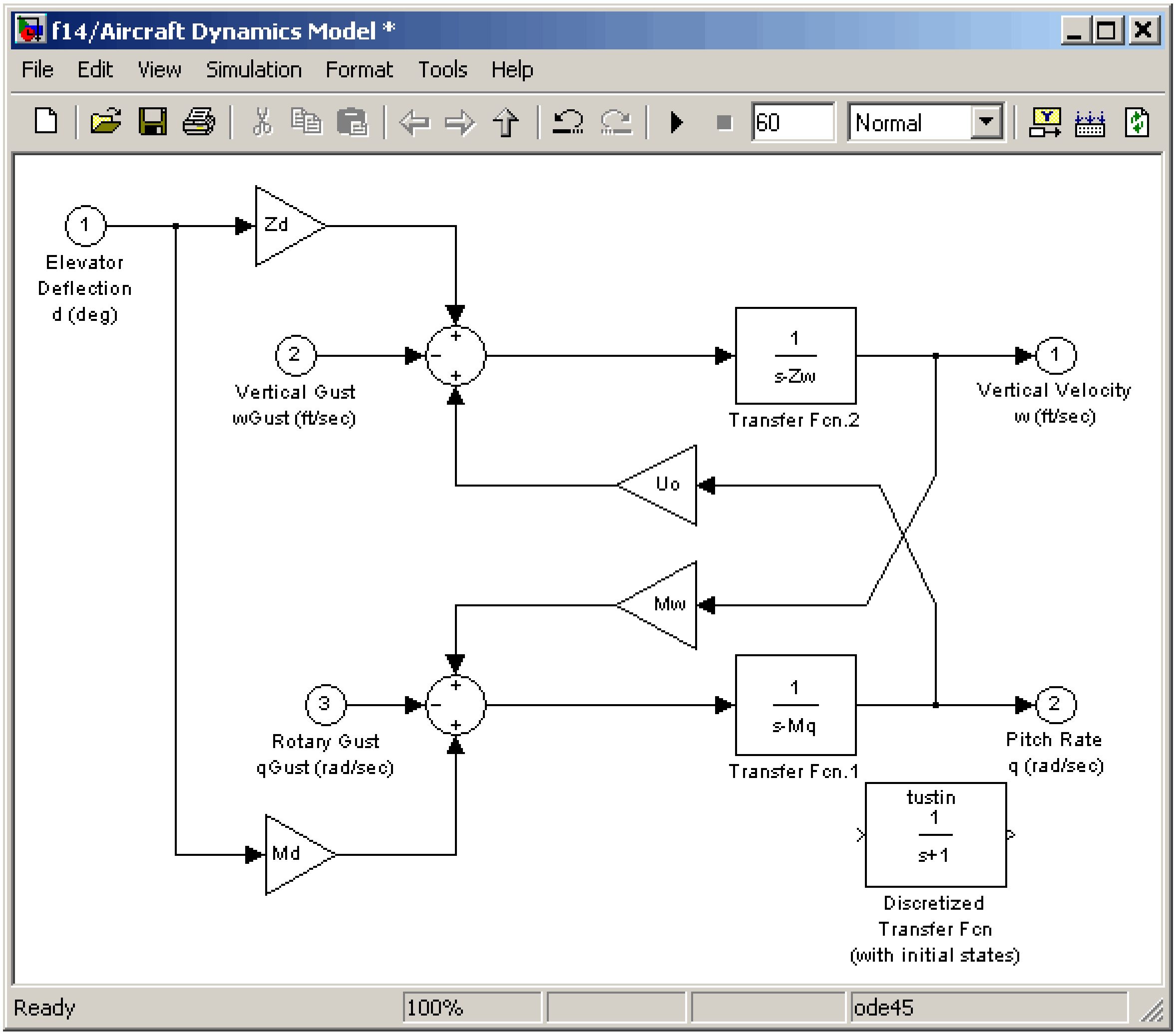


This library contains s-domain discretized blocks.

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.

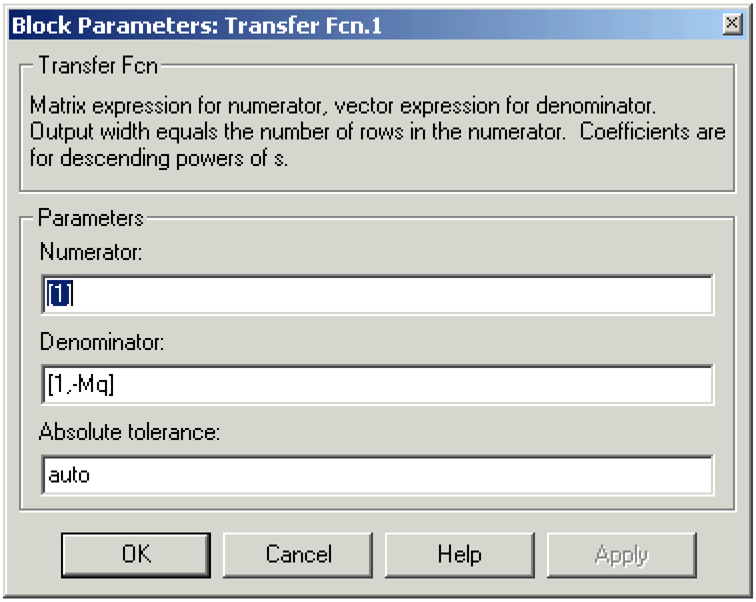
b Drag it into the **f14/Aircraft Dynamics Model** window.


f14/Aircraft Dynamics Model *
File
Edit
View
Simulation
Format
Tools
Help
60
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 (with initial states)
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]

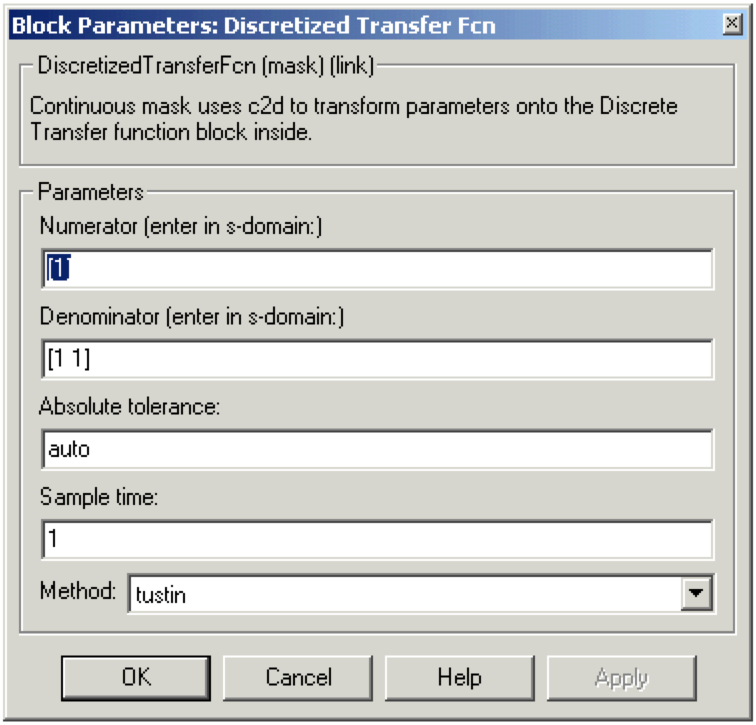
Absolute tolerance:

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:

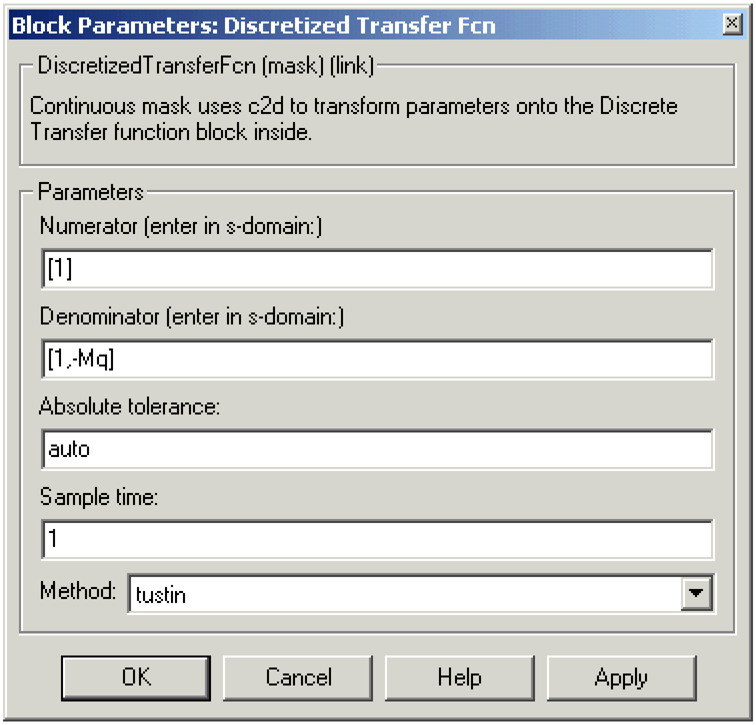
auto

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:

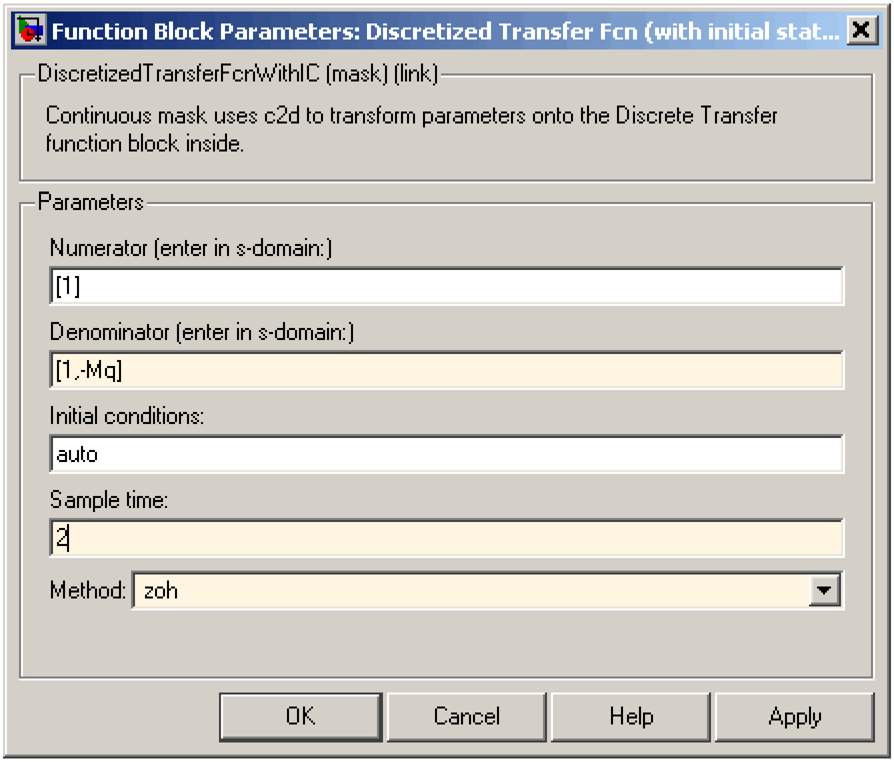
1

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.



Function Block Parameters: Discretized Transfer Fcn (with initial stat...

DiscretizedTransferFcnWithIC (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]

Initial conditions:

auto

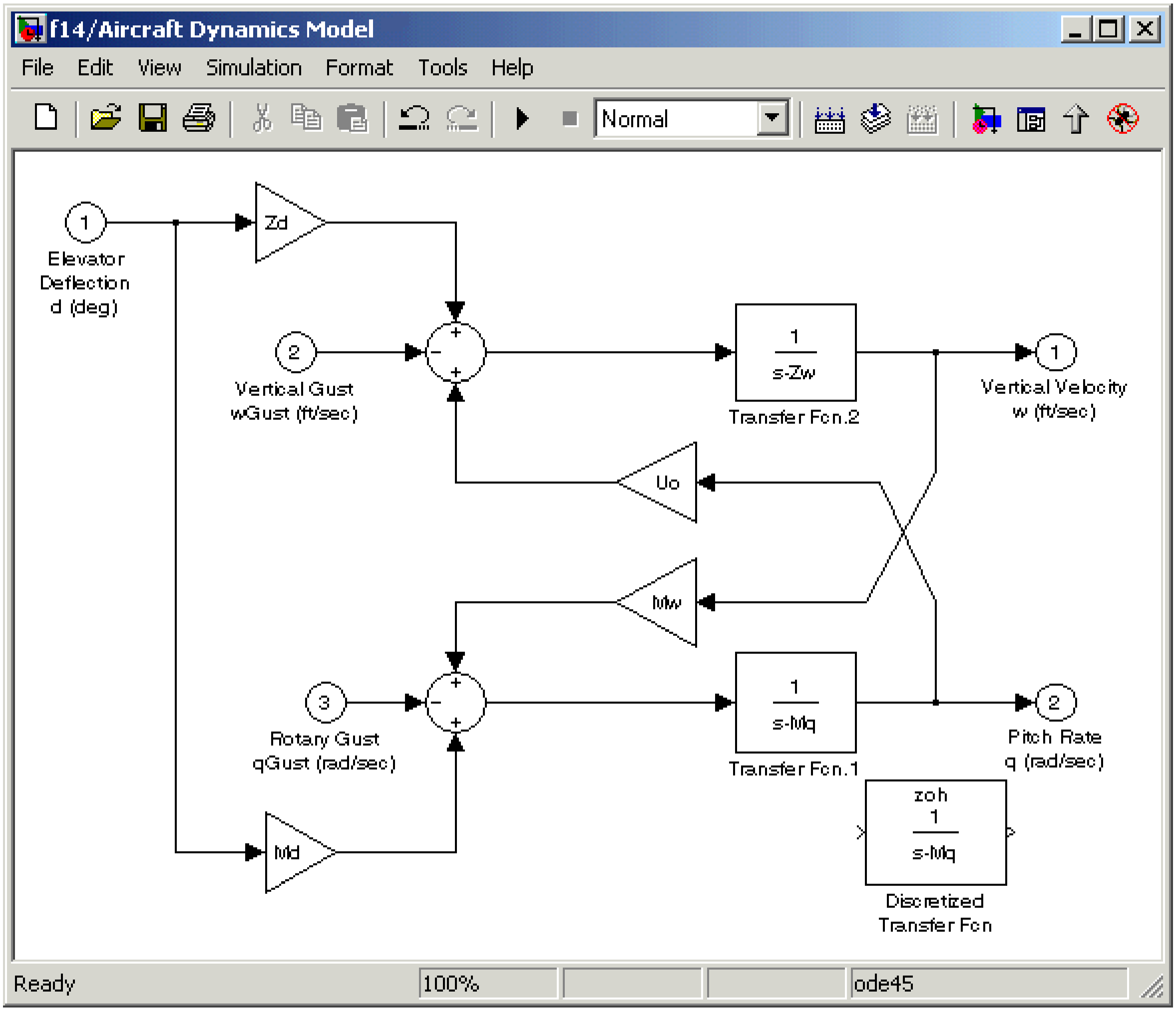
Sample time:

2

Method: zoh

9 Click **OK**.

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

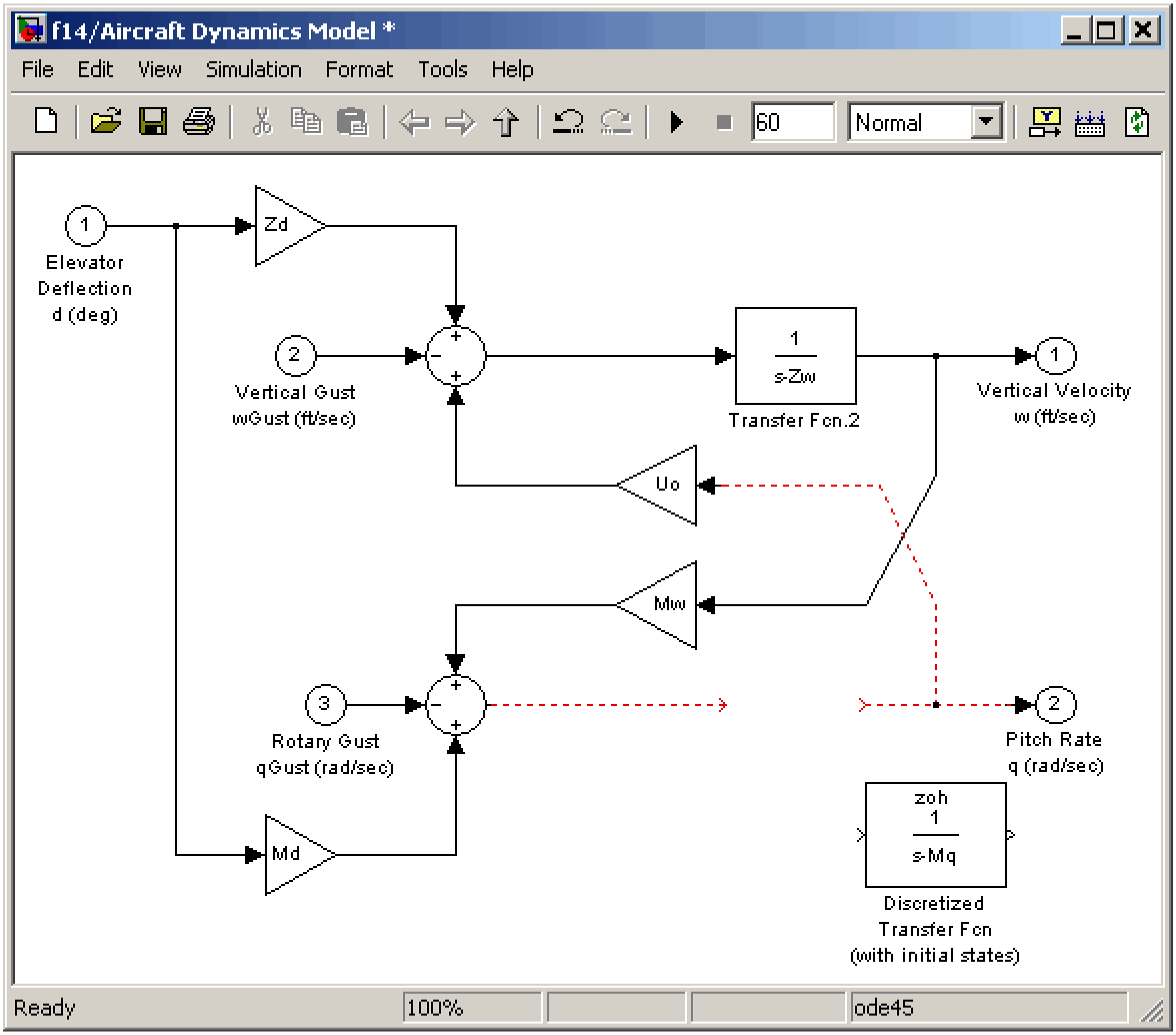


10 Delete the original Transfer Fcn.1 block.

a Click the Transfer Fcn.1 block.

b Press the **Delete** key.

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

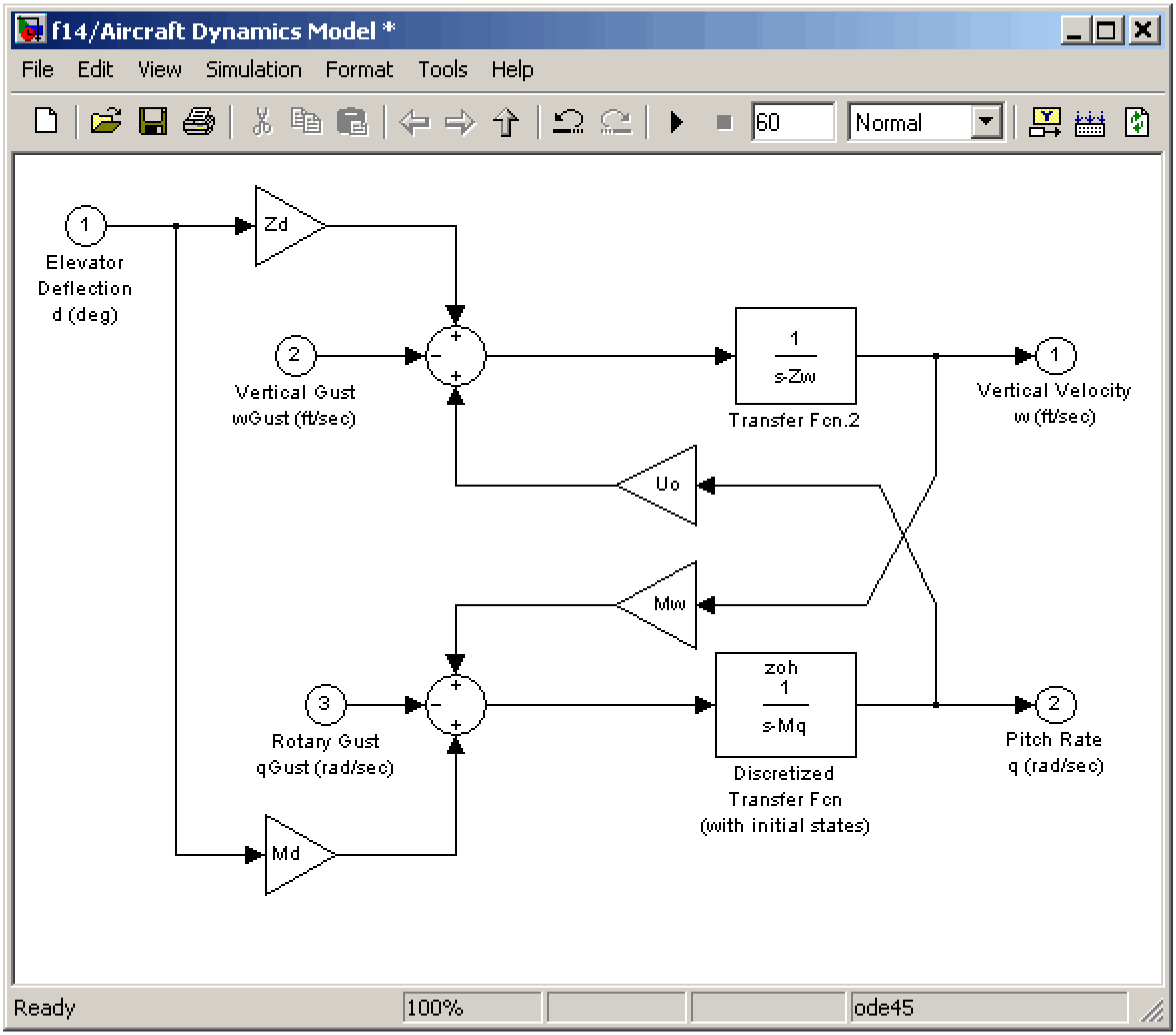

f14/Aircraft Dynamics Model *
File
Edit
View
Simulation
Format
Tools
Help
60
Normal
1
Elevator Deflection d (deg)
Zd
2
Vertical Gust wGust (ft/sec)
+
-
+
1
s-Zw
Transfer Fcn.2
1
Vertical Velocity w (ft/sec)
Uo
Mw
3
Rotary Gust qGust (rad/sec)
+
-
+
2
Pitch Rate q (rad/sec)
zoh
1
s-Mq
Discretized Transfer Fcn (with initial states)
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.



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 “Model Construction” in *Simulink Reference*.

4

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.

For more information about Blocks, see:

- "Block Data Tips" on page 4-2
- "Virtual Blocks" on page 4-2

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	Virtual if input bus is virtual.
Demux	Always virtual.
Enable	Virtual unless connected directly to an Outport block.
From	Always virtual.
Goto	Always virtual.

Block Name	Condition Under Which Block Is Virtual
Goto Tag Visibility	Always virtual.
Ground	Always virtual.
Inport	Virtual *unless* the block resides in a conditionally executed or atomic 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. For information, see:

- “Copying and Moving Blocks from One Window to Another” on page 4-4
- “Moving Blocks in a Model” on page 4-5
- “Copying Blocks in a Model” on page 4-7
- “Deleting Blocks” on page 4-7

Copying and Moving Blocks from One Window to Another

As you build your model, you often copy blocks from Simulink block libraries or other libraries or models into your model window. To do this:

1. Open the appropriate block library or model window.
2. Drag the block to copy into the target model window. To drag a block, position the cursor over the block, then press and hold down the mouse button. Move the cursor into the target window, then release the mouse button.

You can also drag blocks from the Simulink Library Browser into a model window. See “Browsing Block Libraries” on page 4-52 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 4-25.

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

Moving Blocks in a Model

To move a single block from one place to another in a model window, drag the block to a new location. Simulink automatically repositions lines connected to the moved block.

To move more than one block, including connecting lines:

1 Select the blocks and lines. If you need information about how to select more than one block, see “Selecting Multiple Objects” on page 3-5.

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

To move a block, disconnecting lines:

1 Select the block.

2 Press the **Shift** key, then drag the block to its new location and release the mouse button.

You can also move a block by selecting the block and pressing the arrow keys.

Moving blocks from one window to another is similar to copying blocks, except that you hold down the **Shift** key while you select the blocks.

You can use the **Undo** command from the **Edit** menu to remove an added block.

Aligning Blocks

Simulink uses an invisible five-pixel grid to simplify the alignment of blocks. When you move a block to a new location, the block snaps to the nearest line on the grid.

To facilitate aligning blocks at larger intervals, Simulink allows you to display a larger grid in a model window. To display the grid, enter the following command at the MATLAB command prompt.

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

The default width of the grid is 20 pixels. To change the grid spacing, enter

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

For example, to change the grid spacing to 25 pixels, enter

```
set_param('<model name>','gridspacing',25)
```

Note The new spacing must be a multiple of five pixels to ensure that the displayed grid aligns with the invisible snap grid.

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

Positioning Blocks Programmatically

You can position (and resize) a block programmatically, using its Position parameter. For example, the following command

```
set_param(gcb, 'Position', [5 5 20 20]);
```

moves the currently selected block to a location 5 points down and 5 points to the right of the top left corner of the block diagram and sets the block's height and width to 15 points, respectively.

Note The maximum size of a block diagram's height and width is `32767` points. Simulink displays an error message if you try to moving or resize a block to a position that exceeds the diagram's boundaries.

Copying Blocks in a Model

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

Deleting Blocks

To delete one or more blocks, select the blocks to be deleted and press the **Delete** or **Backspace** key. You can also choose **Clear** or **Cut** from the **Edit** menu. The **Cut** command writes the blocks into the clipboard, which enables you to paste them into a model. Using the **Delete** or **Backspace** key or the **Clear** command does not enable you to paste the block later.

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

Working with Block Parameters

All Simulink blocks have attributes that you can specify. Some user-specifiable attributes are common to all Simulink blocks, for example, a block's name and foreground color. Other attributes are specific to a block, for example, the gain of a Gain block. Simulink associates a variable, called a block parameter, with each user-specifiable attribute of a block. You specify the attribute by setting its associated parameter to a corresponding value. For example, to set the foreground color of a block to red, you set the value of its foreground color parameter to the string `'red'`. The Simulink parameter reference lists the names, usages, and valid settings for Simulink block parameters (see "Common Block Parameters" and "Block-Specific Parameters").

Mathematical Versus Configuration Parameters

Block parameters fall into two broad categories. A *mathematical parameter* is a parameter used to compute the value of a block's output, for example, the Gain parameter of a Gain block. All other parameters are *configuration parameters*, for example, a Gain block's Name parameter. In general, you can change the values of mathematical but not configuration parameters during simulation (see "Changing the Values of Block Parameters During Simulation" on page 4-12).

Setting Block Parameters

You can use the Simulink `set_param` command to set the value of any Simulink block parameter. In addition, you can set many block parameters via Simulink dialog boxes and menus. These include:

- **Format** menu

  The Model Editor's **Format** menu allows you to specify attributes of the currently selected block that are visible on the model's block diagram, such as the block's name and color (see "Changing a Block's Appearance" on page 4-23 for more information).

- **Block Properties** dialog box

  Specifies various attributes that are common to all blocks (see "Block Properties Dialog Box" on page 4-16 for more information).

- **Block Parameter** dialog box

  Every block has a dialog box that allows you to specify values for attributes that are specific to that type of block. See "Displaying a Block's Parameter Dialog Box" on page 4-9 for information on displaying a block's parameter dialog box. For information on the parameter dialog of a specific block, see "Blocks — Alphabetical List" in the online Simulink reference.

- Model Explorer

  The Model Explorer allows you to quickly find one or more blocks and set their properties, thus facilitating global changes to a model, for example, changing the gain of all of a model's Gain blocks. See "The Model Explorer" on page 10-2 for more information.

Displaying a Block's Parameter Dialog Box

To display a block's parameter dialog box, double-click the block in the model or library window. You can also display a block's parameter dialog box by selecting the block in the model's block diagram and choosing **BLOCK Parameters** from the model window's **Edit** menu or from the block's context (right-click) menu, where **BLOCK** is the name of the block you selected, e.g., **Constant Parameters**.

Note Double-clicking a block to display its parameter dialog box works for all blocks with parameter dialog boxes except for Subsystem blocks. You must use the Model Editor's **Edit** menu or the block's context menu to display a Subsystem block's parameter dialog box.

Specifying Numeric Parameter Values

Many block parameters, including mathematical parameters, accept MATLAB expression strings as values. When Simulink compiles a model, for example, at the start of a simulation or when you update the model, Simulink sets the compiled values of the parameters to the result of evaluating the expressions.

- “Using Workspace Variables in Parameter Expressions” on page 4-10
- “Resolving Variable References in Block Parameter Expressions” on page 4-10
- “Using Parameter Objects to Specify Parameter Values” on page 4-11
- “Determining Parameter Data Types” on page 4-11

Using Workspace Variables in Parameter Expressions

Block parameter expressions can include variables defined in the model’s mask and model workspaces and in the MATLAB workspace. Using a workspace variable facilitates updating a model that sets multiple block parameters to the same value, i.e., it allows you to update multiple parameters by setting the value of a single workspace variable. Using a workspace variable also allows you to change the value of a parameter during simulation without having to open a block’s parameter dialog box (see “Changing the Values of Block Parameters During Simulation” on page 4-12).

Note If you plan to generate code from a model, you can use workspace variables to specify the name, data type, scope, volatility, tunability, and other attributes of variables used to represent the parameter in the generated code. For more information, see “Parameter Storage, Interfacing, and Tuning” in the Real-Time Workshop documentation.

Resolving Variable References in Block Parameter Expressions

When evaluating a block parameter expression that contains a variable, Simulink by default searches the workspace hierarchy (see “Working with Model Workspaces” on page 3-107 and “Mask Workspace” on page 13-4) of the model that contains the expression for a definition of the variable. The search begins with the workspace of the nearest masked subsystem, if any, that contains the block whose parameter expression is being evaluated and continues, if necessary, up the workspace hierarchy to the model's workspace and finally to the MATLAB workspace itself. If the workspace hierarchy contains a definition of the variable, Simulink uses the first definition that it finds. If the variable is not defined in any workspace, Simulink halts compilation of the model and displays an error message.

Note You can prevent block parameter variable resolution from passing through particular subsystems in your model. See the **Permit hierarchical resolution** option of the Subsystem block for more information.

Using Parameter Objects to Specify Parameter Values

You can use `Simulink.Parameter` objects in parameter expressions to specify parameter values. For example, `K` and `2*K` are both valid parameter expressions where `K` is a workspace variable that references a `Simulink.Parameter` object. In both cases, Simulink uses the parameter object's `Value` property as the value of `K`. Using parameter objects to specify parameters can facilitate tuning parameters in some applications (see "Using a Parameter Object to Specify a Parameter As Noninlined" on page 4-15 and "Parameterizing Model References" on page 3-69 for more information).

Note Do not use expressions of the form `p.Value` where `p` is a parameter object in parameter expressions. Such expressions cause evaluation errors when Simulink compiles the model.

Determining Parameter Data Types

When Simulink compiles a model, each of the model's blocks determines a data type for storing the values of its parameters whose values are specified by MATLAB parameter expressions. Most blocks use internal rules to determine the data type assigned to a specific parameter. Exceptions include the Gain block, whose parameter dialog box allows you to specify the data type assigned to the compiled value of its Gain parameter. You can configure your model to check whether the data type assigned to a parameter can accommodate the parameter value specified by the model (see "Data Validity Diagnostics" on page 11-111).

Changing the Values of Block Parameters During Simulation

Simulink lets you change the values of many block parameters during simulation. Such parameters are called *tunable parameters*. In general, only parameters that represent mathematical variables, such as the Gain parameter of the Gain block, are tunable. Parameters that specify the appearance or structure of a block, e.g., the number of inputs of a Sum block, or when it is evaluated, e.g., a block's sample time or priority, are not tunable. You can tell whether a particular parameter is tunable by examining its edit control in the block's dialog box or Model Explorer during simulation. If the control is disabled, the parameter is nontunable.

Note You cannot tune inline parameters. See "Inlining Parameters" on page 4-14 for more information.

Tuning a Block Parameter

You can use a block's dialog box or the Model Explorer to modify the tunable parameters of any block, except a source block (see "Changing Source Block Parameters During Simulation" on page 4-12). To use the block's parameter dialog box, open the block's parameter dialog box, change the value displayed in the dialog box, and click the dialog box's **OK** or **Apply** button.

You can also tune a parameter at the MATLAB command line, using either the `set_param` command or by assigning a new value to the MATLAB workspace variable that specifies the parameter's value. In either case, you must update the model's block diagram for the change to take effect (see "Updating a Block Diagram" on page 2-14).

Changing Source Block Parameters During Simulation

Opening the dialog box of a source block with tunable parameters (see "Source Blocks with Tunable Parameters" on page 4-13) causes a running simulation to pause. While the simulation is paused, you can edit the parameter values displayed on the dialog box. However, you must close the dialog box to have the changes take effect and allow the simulation to continue. Similarly, starting a simulation causes any open dialog boxes associated with source blocks with tunable parameters to close.

Note If you enable the **Inline parameters** option, Simulink does not pause the simulation when you open a source block's dialog box because all of the parameter fields are disabled and can be viewed but cannot be changed.

The Model Explorer disables the parameter fields that it displays in the list view and the dialog pane for a source block with tunable parameters while a simulation is running. As a result, you cannot use the Model Explorer to change the block's parameters. However, while the simulation is running, the Model Explorer displays a **Modify** button in the dialog view for the block. Clicking the **Modify** button opens the block's dialog box. Note that this causes the simulation to pause. You can then change the block's parameters. You must close the dialog box to have the changes take effect and allow the simulation to continue. Your changes appear in the Model Explorer after you close the dialog box.

Source Blocks with Tunable Parameters. Source blocks with tunable parameters include the following blocks.

- Simulink source blocks, including
  - Band-Limited White Noise
  - Chirp Signal
  - Constant
  - Pulse Generator
  - Ramp
  - Random Number
  - Repeating Sequence
  - Signal Generator
  - Sine Wave
  - Step
  - Uniform Random Number
- User-developed masked subsystem blocks that have one or more tunable parameters and one or more output ports, but no input ports.

- S-Function and M-file (level 2) S-Function blocks that have one or more tunable parameters and one or more output ports but no input ports.

Inlining Parameters

The **Inline parameters** optimization (see "Inline parameters" on page 11-90) controls how mathematical block parameters appear in code generated from the model. When this optimization is off (the default), a model's mathematical block parameters appear as variables in the generated code. As a result, you can tune the parameters both during simulation and when executing the code. When this option is on, the parameters appear in the generated code as inlined numeric constants. This reduces the generated code's memory and processing requirements. However, because the inline parameters appear as constants in the generated code, you cannot tune them during code execution. Furthermore, to ensure that simulation faithfully models the generated code, Simulink prevents you from changing the values of block parameters during simulation when the **Inline parameters** option is on.

Specifying Some Parameters as Noninline

Suppose that you want to take advantage of the **Inline parameters** optimization while retaining the ability to tune some of your model's parameters. You can do this by declaring some parameters as *noninline*, using either the "Model Parameter Configuration Dialog Box" on page 11-100 or a `Simulink.Parameter` object. In either case, you must use a workspace variable to specify the value of the parameter.

Note The documentation for the Real-Time Workshop refers to workspace variables used to specify the value of noninline parameters as *tunable workspace parameters*. In this context, the term *parameter* refers to a workspace variable used to specify a parameter as opposed to the parameter itself.

Note When compiling a model with the inline parameters option on, Simulink checks to ensure that the data types of the workspace variables used to specify the model's noninline parameters are compatible with code generation. If not, Simulink halts the compilation and displays an error. See "Tunable Workspace Parameter Data Type Considerations" for more information.

Using a Parameter Object to Specify a Parameter As Noninlined. If you use a parameter object to specify a parameter's value (see "Using Parameter Objects to Specify Parameter Values" on page 4-11), you can also use the object to specify the parameter as noninlined. To do this, set the parameter object's `RTWInfo.StorageClass` property to any value but `'Auto'` (the default).

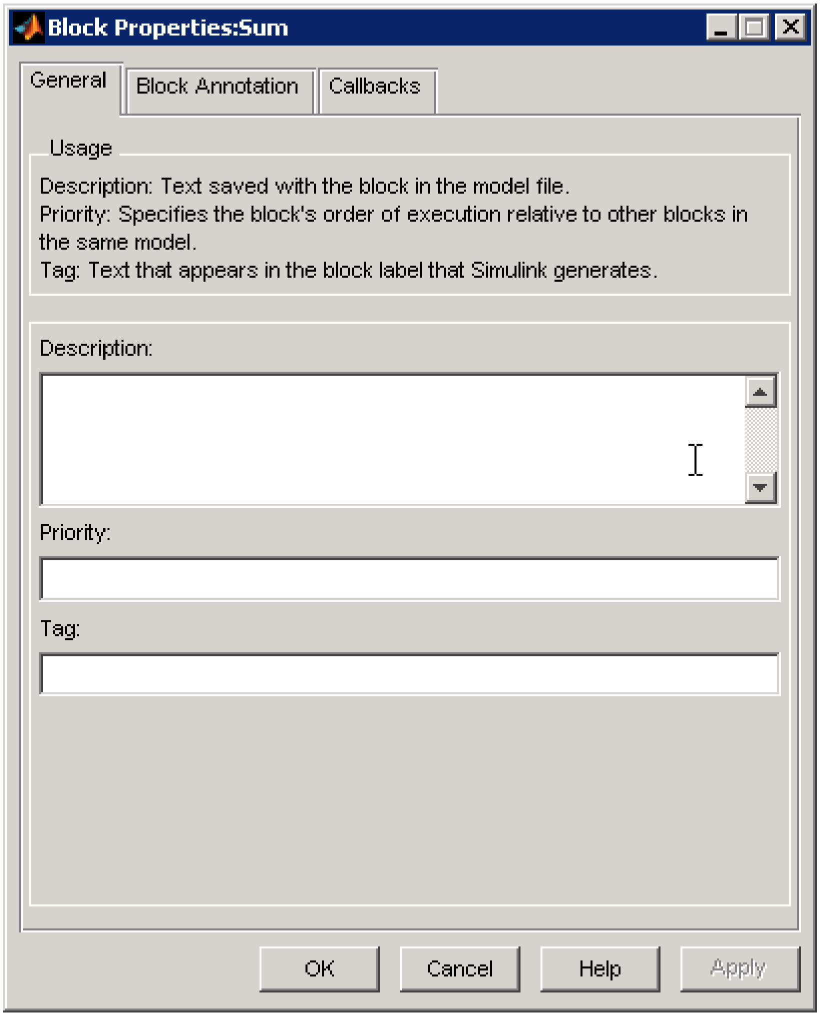
```
K=Simulink.Parameter;
K.RTWInfo.StorageClass = 'SimulinkGlobal';
```

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

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

Block Properties Dialog Box

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



Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

Description:

Priority:

Tag:

The dialog box contains the following tabbed panes.

General Pane

This pane allows you to set the following properties.

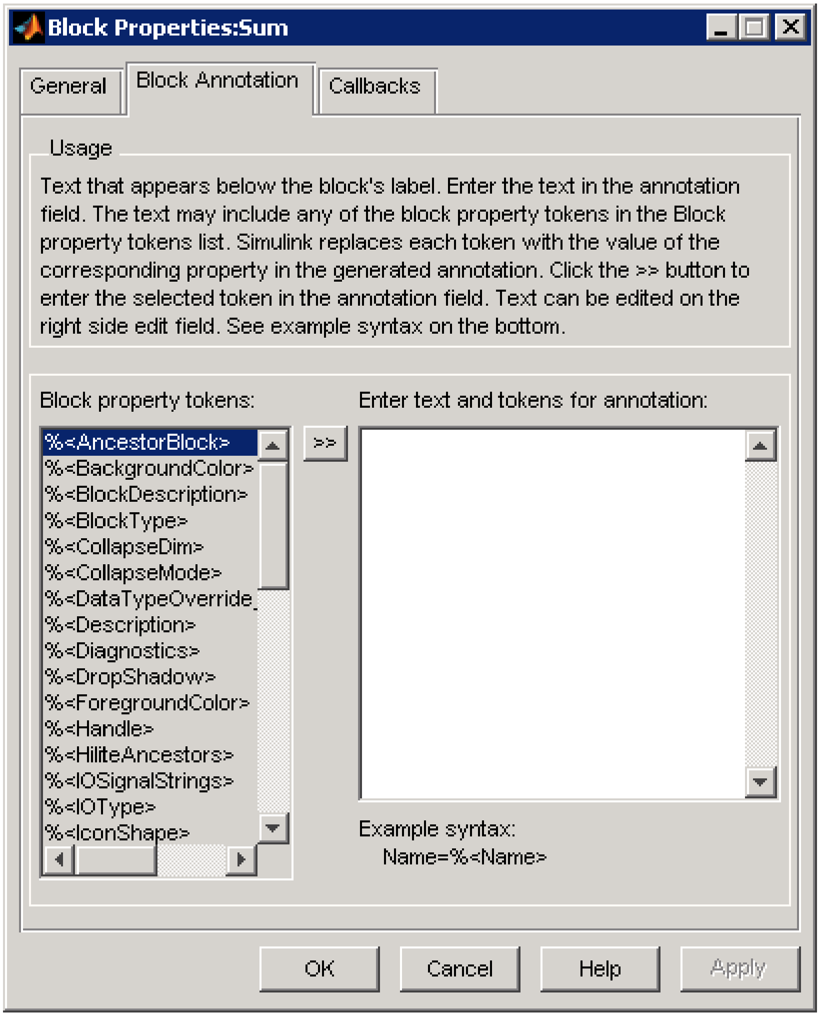
Description. Brief description of the block's purpose.

Priority. Execution priority of this block relative to other blocks in the model. See "Assigning Block Priorities" on page 4-33 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.



Text that appears below the block's label. Enter the text in the annotation field. The text may include any of the block property tokens in the Block property tokens list. Simulink replaces each token with the value of the corresponding property in the generated annotation. Click the >> button to enter the selected token in the annotation field. Text can be edited on the right side edit field. See example syntax on the bottom.

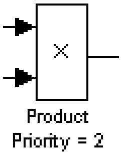
- %<AncestorBlock>
- %<BackgroundColor>
- %<BlockDescription>
- %<BlockType>
- %<CollapseDim>
- %<CollapseMode>
- %<DataTypeOverride_
- %<Description>
- %<Diagnostics>
- %<DropShadow>
- %<ForegroundColor>
- %<Handle>
- %<HiliteAncestors>
- %<IOSignalStrings>
- %<IOType>
- %<IconShape>

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.,

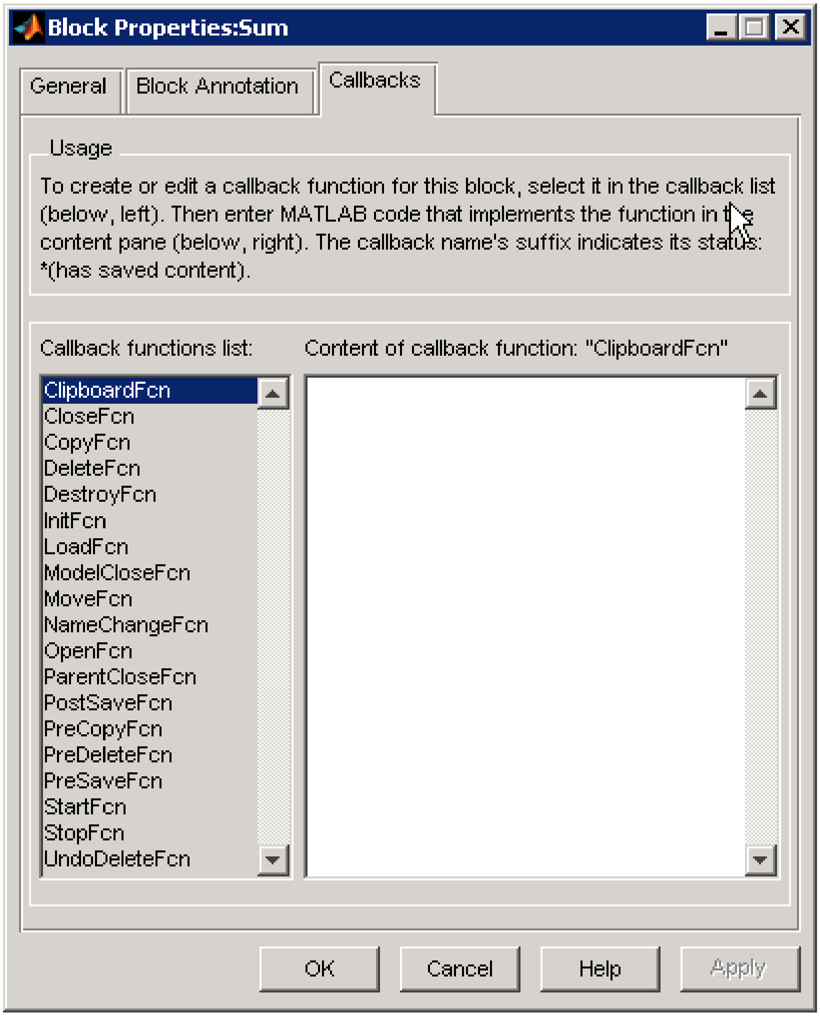


The block property tag list on the left side of the pane lists all the tags that are valid for the currently selected block. To include one of the listed tags in the annotation, select the tag and then click the button between the tag list and the annotation field.

You can also create block annotations programmatically. See "Creating Block Annotations Programmatically" on page 4-21.

Callbacks Pane

The **Callbacks Pane** allows you to specify implementations for a block's callbacks (see "Using Callback Functions" on page 3-100).



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:

ClipboardFcn
CloseFcn
CopyFcn
DeleteFcn
DestroyFcn
InitFcn
LoadFcn
ModelCloseFcn
MoveFcn
NameChangeFcn
OpenFcn
ParentCloseFcn
PostSaveFcn
PreCopyFcn
PreDeleteFcn
PreSaveFcn
StartFcn
StopFcn
UndoDeleteFcn

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 **Apply** to save the change. Simulink appends an asterisk to the name of the saved callback to indicate that it has been implemented.

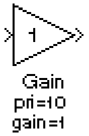
Creating Block Annotations Programmatically

You can use a block's `AttributesFormatString` parameter to display selected 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). "Model and Block Parameters" in *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. See “Block State Storage and Interfacing” in *Real-Time Workshop User’s Guide* 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.

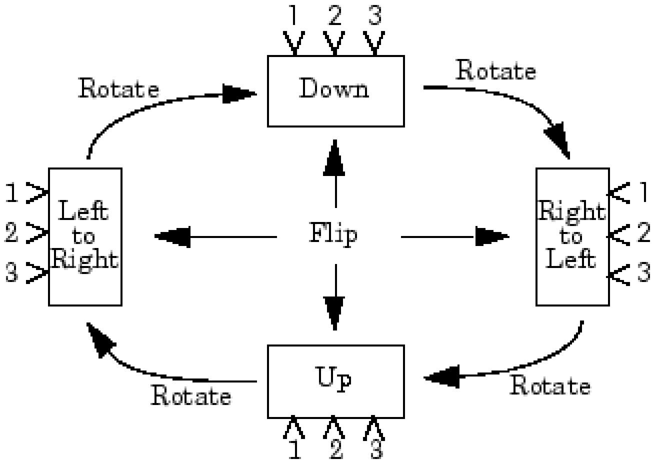
- "Changing the Orientation of a Block" on page 4-23
- "Resizing a Block" on page 4-24
- "Displaying Parameters Beneath a Block" on page 4-25
- "Using Drop Shadows" on page 4-25
- "Manipulating Block Names" on page 4-25
- "Specifying a Block's Color" on page 4-27

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 selecting 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 following figure 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.

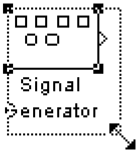


Resizing a Block

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 following figure below shows a Signal Generator block being resized. The lower-right handle was selected and dragged to the cursor position. When the mouse button is released, the block takes its new size.

This figure shows a block being resized:



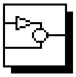
Displaying Parameters Beneath a Block

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 4-16)
- By setting the value of the block's `AttributesFormatString` property to the format string, using `set_param`

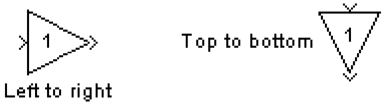
Using Drop Shadows

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



Manipulating Block Names

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



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

Changing Block Names

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

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

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

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

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

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

Changing the Location of a Block Name

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

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

For more information about block orientation, see “Changing the Orientation of a Block” on page 4-23.

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:

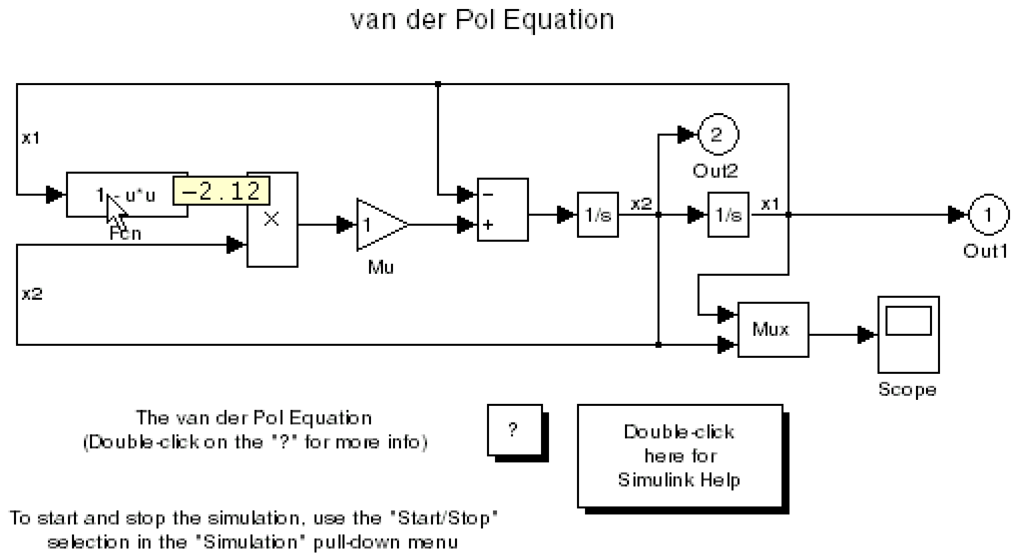
- The **Hide Name** menu item hides a visible block name. When you select **Hide Name**, it changes to **Show Name** when that block is selected.
- The **Show Name** menu item shows a hidden block name.

Specifying a Block's Color

See "Specifying Block Diagram Colors" on page 3-7 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.



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 4-28) 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 4-30).

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.

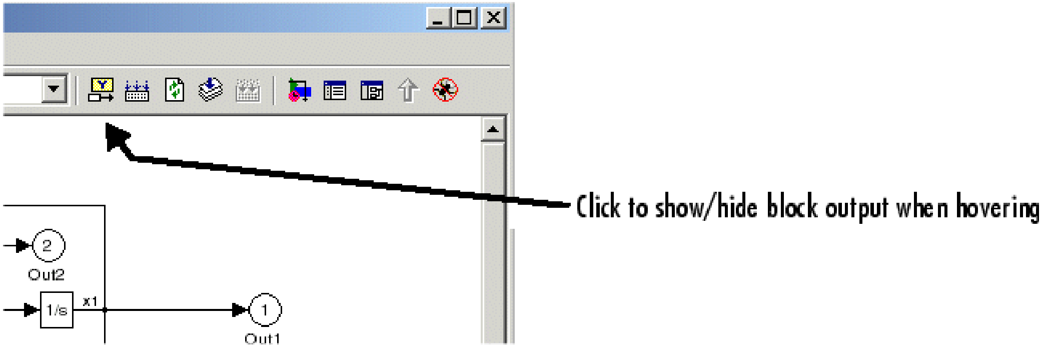
- **Show when hovering**

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

- **Toggle when selected**

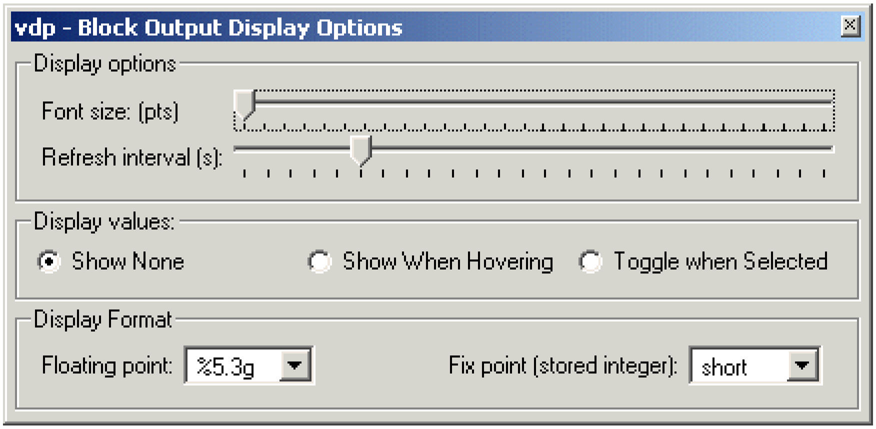
  Selecting a block displays its outputs. Reselecting the block turns the display off.

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 toolbar.



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 (see "Block Methods" on page 1-12) 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. For more information, see

- "How Simulink Determines the Sorted Order" on page 4-31
- "Displaying the Sorted Order" on page 4-33
- "Assigning Block Priorities" on page 4-33

How Simulink Determines the Sorted Order

Simulink uses the following basic rules to sort the blocks:

- Each block must appear in the sorted order ahead any of the blocks whose direct-feedthrough ports (see "About Direct-Feedthrough Ports" on page 4-32) it drives.

  This rule ensures that the direct-feedthrough inputs to blocks will be valid when block methods that require current inputs are invoked.

- Blocks that do not have direct feedthrough inputs can appear anywhere in the sorted order as long as they precede any blocks whose direct-feedthrough inputs they drive.

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

The result of applying these rules is a sorted order in which blocks without direct feedthrough ports appear at the 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 the block needs the value of the direct-feedthrough input to compute its output.

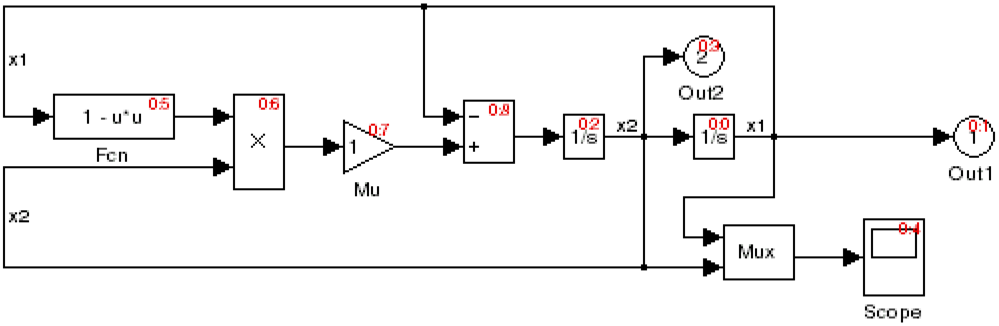
However, an algebraic loop can represent a set of simultaneous algebraic equations (hence the name) where the block's input and output are the unknowns. Further, these equations can have valid solutions at each time step. Accordingly, Simulink assumes that loops involving direct-feedthrough ports do, in fact, represent a solvable set of algebraic equations and attempts to solve them each time the block's output is required during a simulation. For more information, see "Algebraic Loops" on page 1-26.

About Direct-Feedthrough Ports

In order to ensure that the sorted order reflects data dependencies among blocks, Simulink categorizes a block's input ports according to the dependency of the block's outputs on its 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).

Displaying the Sorted Order

To display the sorted order, select **Sorted Order** from the Simulink **Format > Block Displays** 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 whose execution context (see "Conditional Execution Behavior" on page 3-57) 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 4-2). 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 4-16).

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 "Diagnostics Pane" on page 11-102).

Working with Block Libraries

Libraries are collections of blocks that can be copied into models. Blocks copied from a library remain linked to their originals such that changes in the originals automatically propagate to the copies in a model. Libraries ensure that your models automatically include the most recent versions of blocks developed by yourself or others.

See the following topics for more information on working with block libraries:

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.

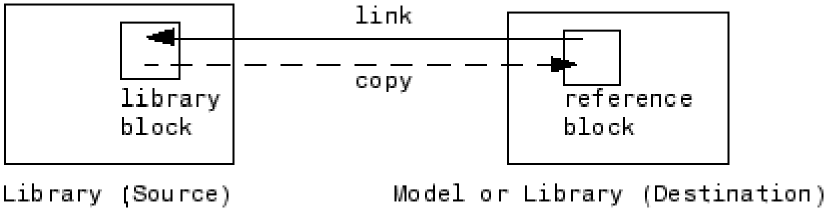
Library block - A block in a library.

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 2-2 for information on displaying and using this library.

Creating a Library

Simulink allows you to create your own library of masked or subsystem blocks. 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.

Once you have created a library, you can drag blocks from models or other libraries into it. If you want to be able to create links in models to a block in the library, you must provide a mask (see Chapter 13, “Creating Block Masks”) for the block. You can also provide a mask for a subsystem in a library but you do not need to do so in order to create links to it in models.

The library must be named (saved) before you can copy blocks from it. See “Adding Libraries to the Library Browser” on page 4-55 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 4-4) or by dragging the block from the Library Browser (see “Browsing Block Libraries” on page 4-52) into the model window.

When you drag a library block into a model or another library, Simulink creates a copy of the library block, called the reference block, in the model or the other library. 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 make nonstructural changes to the contents of the reference subsystem (see “Modifying a Linked Subsystem” on page 4-39).

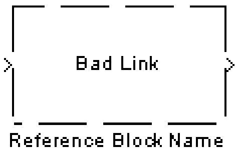
The library and reference blocks are linked *by name*; that is, the reference block is linked to the specific block and library whose names are in effect at the time the copy is made.

Fixing Unresolved Library Links

If Simulink is unable to find either the library block or the source library on your MATLAB path when it attempts to update the reference block, the link becomes *unresolved*. Simulink issues an error message and displays these blocks using red dashed lines. The error message is

```
Failed to find block "source-block-name"
in library "source-library-name"
referenced by block
"reference-block-path".
```

The unresolved reference block appears like this (colored red).



To fix a bad link, you must do one of the following:

- Delete the unlinked reference block and copy the library block back into your model.
- Add the directory that contains the required library to the MATLAB path and select **Update Diagram** from the **Edit** menu.
- Double-click the unlinked reference block to open its dialog box (see Bad Link). On the dialog box that appears, correct the pathname in the **Source block** field and click **OK**.

Disabling Library Links

Simulink allows you to disable linked blocks in a model. Disabling a link means that Simulink will not get the latest version of the link from the library before simulation. Simulink simulates disabled links as regular blocks 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 make any change to the local copy of an active library link that does not alter the structure of the local copy. Examples of nonstructural changes include changes to parameter values that do not affect the structure of the subsystem. Examples of structural modifications include adding or deleting a block or line or changing the number of ports on a block.

Note Simulink displays "parameterized link" on the parameter dialog box of a masked subsystem whose parameters differ in value from those of the library version to which it is linked. This allows you to determine whether the local copy differs from the library version simply by opening the local copy's dialog box.

When you save the model containing the modified subsystem, Simulink saves the changes to the local copy of the subsystem together with the path to the library copy in the model's model `(.mdl)` file. When you reopen the system, Simulink copies the library subsystem into the loaded model and applies the saved changes.

Note If you attempt to use the Simulink GUI to make a structural change to the local copy of an active library link, for example, by editing the subsystem's diagram, Simulink offers to disable the link to the subsystem. If you accept, Simulink makes the change. Otherwise, it does not allow you to make the structural change. Simulink does not prevent you from using `set_param` to attempt to make a structural change to an active link. However, the results of the change are undefined.

Self-Modifying Linked Subsystems

Simulink allows linked subsystems to change their own structural contents without disabling the link. This allows you to create masked subsystems that modify their structural contents based on mask parameter dialog box values. See "Creating Self-Modifying Masks for Library Blocks" on page 13-46 for more information.

Propagating Link Modifications

If you restore a disabled 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 version of the subsystem specified by the restored link with the changes made in the model's version of that subsystem. If you choose to discard the changes, Simulink replaces the version of the subsystem in the model with the version in the library. In either case, the end result is that the versions of the subsystem in the library and the model are exactly the same.

If you restore a disabled link to a block with nonstructural changes, Simulink enables the link without prompting you to propagate or discard the changes. To see the nonstructural parameter differences between the model's version of a library block and the library block itself, choose **View changes** from the **Link options** menu.

If you want to propagate or discard nonstructural changes, select the modified copy of the library block in the model, choose **Link options** from the model window's **Edit** or context menu, then choose **Propagate/Discard changes**. A dialog box appears that asks whether you want to propagate or discard the changes. If you elect to propagate the changes, Simulink applies the changes made to the model's copy of the library block to the library block itself. If you elect to discard the changes, Simulink removes the changes from the model's copy of the block. In either case, the library and model versions of the block become the same.

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 use the `find_system` command
- When you query the `LinkStatus` parameter of a block, using the `get_param` command (see "Library Link Status" on page 4-49)

Note Querying the `StaticLinkStatus` parameter of a block does not update any out-of-date reference blocks.

Making Backward-Compatible Changes to Libraries

Simulink provides the following features to facilitate making changes to library blocks without invalidating models that use the library blocks.

Forwarding Tables

Library forwarding tables enable Simulink to update models to reflect changes in the names or locations of the library blocks that they reference. For example, suppose that you rename a block in a library. You can use a forwarding table for that library to enable Simulink to update models that reference the block under its old name to reference it under its new name.

Simulink allows you to associate a forwarding table with any library. The forwarding table for a library specifies the old locations and new locations of blocks that have moved within the library or to another library. You associate a forwarding table with a library by setting its ForwardingTable parameter to a cell array of two-element cell arrays, each of which specifies the old and new path of a block that has moved. For example, the following command creates a forwarding table and assigns it to a library named Lib1.

```
set_param('Lib1', 'ForwardingTable', {{'Lib1/A', 'Lib2/A'}
{'Lib1/B', 'Lib1/C'}});
```

The forwarding table specifies that block A has moved from Lib1 to Lib2. and that block B is now named C. Suppose that you opens a model that contains links to Lib1/A and Lib1/B. Simulink updates the link to Lib1/A to refer to Lib2/A and the link to Lib1/B to refer to Lib1/C. The changes become permanent when you subsequently save the model.

Creating Aliases for Mask Parameters

Simulink lets you create aliases, i.e., alternate names, for a mask's parameters. A model can then refer to the mask parameter by either its name or its alias. This allows you to change the name of a mask parameter in a library block without having to recreate links to the block in existing models (see “Example: Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes” on page 4-42).

To create aliases for a masked block’s mask parameters, use the `set_param` command to set the block’s MaskVarAliases parameter to a cell array that specifies the names of the aliases in the same order as the mask names appear in the block’s MaskVariables parameter.

Example: Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes. The following example illustrates the use of mask parameter aliases to create backward-compatible parameter name changes.

1. Create a library named `mymdl`.

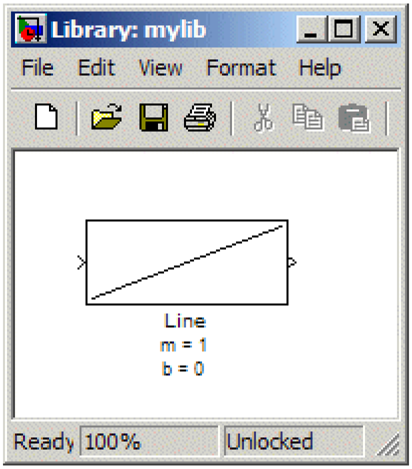
2. Create the masked subsystem described in “Masked Subsystem Example” on page 13-6 in `mymdl`.

3. Name the masked subsystem Line.

4. Set the masked subsystem’s annotation property (see “Block Annotation Pane” on page 4-18) to display the value of its `m` and `b` parameters, i.e., to

   ```
   m = %<m>
   b = %<b>
   ```

The library appears as follows:



5 Save `mylib`.

6 Create a model name `mymdl`.

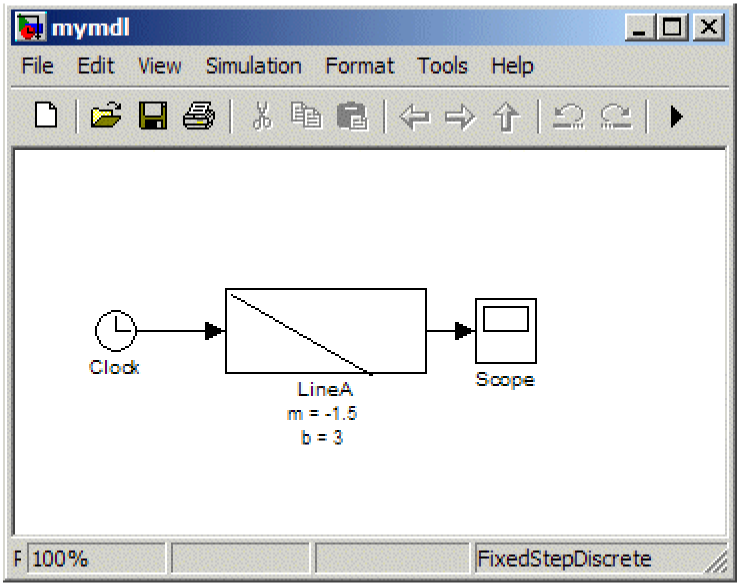
7 Create an instance of the Line block `in mymdl`.

8 Rename the instance LineA.

9 Change the value of LineA's m parameter to `-1.5`.

10 Change the value of LineA's b parameter to `3`.

11 Set LineA's annotation property to display the values of its `m` and `b` parameters.



12 Configure `mymdl` to use a fixed-step, discrete solver with a `0.1` sec. step size.

13 Save `mymdl`.

14 Simulate `mymdl`.

Note that the model simulates without error.

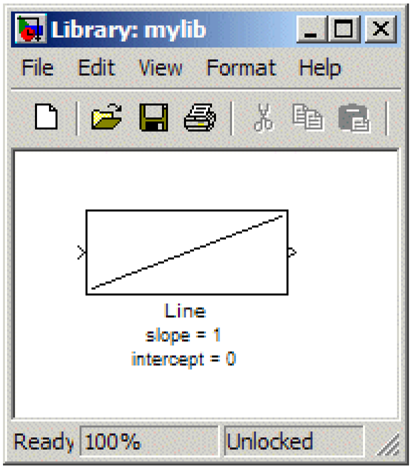
15 Close `mymdl`.

16 Unlock `mylib`.

17 Rename the `m` parameter of the Line block in `mylib` to `slope`.

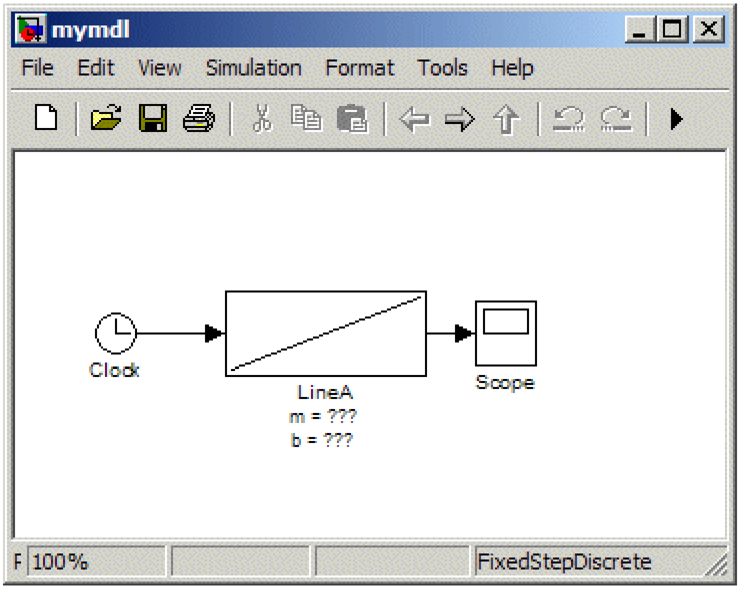
18 Rename Line's `b` parameter to `intercept`.

19 Change Line’s mask icon and annotation properties to reflect the parameter name changes.



20 Save `mylib`.

21 Reopen mymdl.



Note that LineA's icon has reverted to the appearance of its library master (i.e., mylib/Line) and that its annotation displays question marks for the values of m and b. These changes reflect the parameter name changes in the library block . In particular, Simulink cannot find any parameters named m and b in the library block and hence does not know what to do with the instance values for those parameters. As a result, LineA reverts to the default values for the slope and intercept parameters, thereby inadvertently changing the behavior of the model. The following steps show how to use parameter aliases to avoid this inadvertent change of behavior.

22 Close mymdl.

23 Unlock mylib.

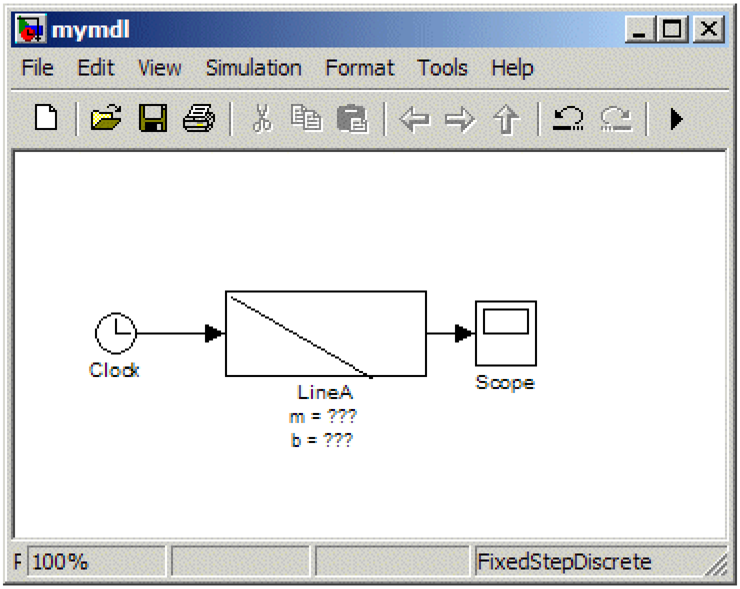
24 Select the Line block in mylib,

25 Execute the following command at the MATLAB command line.

```
set_param(gcb, 'MaskVarAliases',{'m', 'b'})
```

This specifies that `m` and `b` are aliases for the Line block's `slope` and `intercept` parameters.

26 Reopen `mymdl`.



Note that LineA's appearance now reflects the value of the slope parameter under its original name, i.e., m. This is because when Simulink opened the model it found the m is an alias for slope and assigned the value of m stored in the model file to LineA's slope parameter.

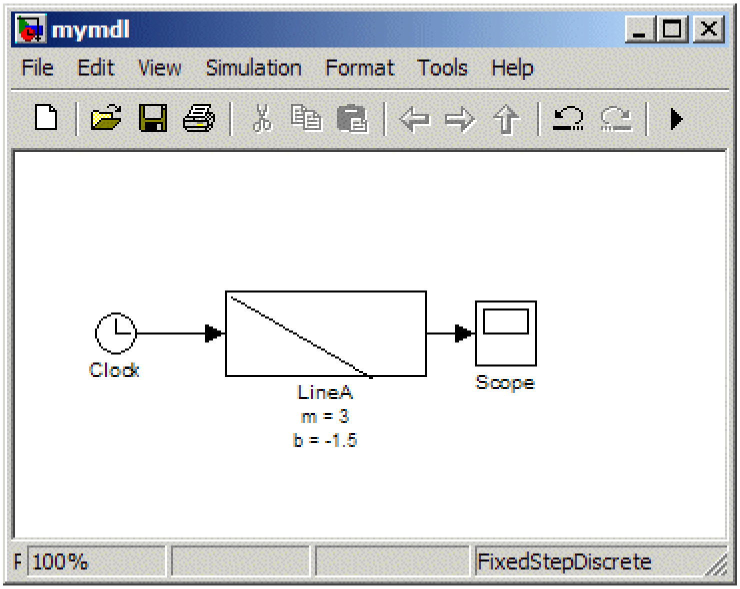
27 Change LineA's block annotation property to reflect LineA's parameter name changes, i.e., replace

```
m = %<m>
b = %<b>
```

with

```
m = %<slope>
b = %<intercept>
```

LineA now appears as follows.



Note that LineA's annotation shows that, thanks to parameter aliasing, Simulink has correctly applied the parameter values stored for LineA in `mymdl`'s model file to the block's renamed parameters.

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 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 and a `StaticLinkStatus` parameter that indicate whether the block is a reference block. The parameters can have these values.

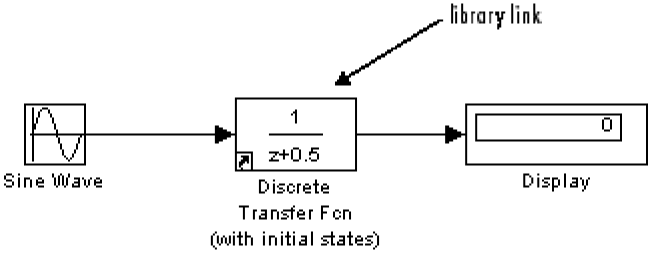
Status	Description
`none`	Block is not a reference block.
`resolved`	Link is resolved.

Status	Description
`unresolved`	Link is unresolved.
`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.

Note Using `get_param` to query a block's `LinkStatus` also resolves any out-of-date block references. It is, therefore, useful when you need to programmatically update library links in a model. Conversely, querying the `StaticLinkStatus` property does not resolve any out-of-date references. You should query the `StaticLinkStatus` property when the call to `get_param` is used in the callback of a child block querying the link status of its parent.

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

Locking Libraries

To lock a block library, save and close the library or set its Lock parameter to `'on'` at the MATLAB command line, using the `set_param` command. Locking a library prevents a user from inadvertently modifying a library, for example, by moving a block in the library or adding or deleting a block from the library. If you attempt to modify a locked library, Simulink displays a dialog box that allows you to unlock the library and make the change. You must then relock the library from the MATLAB command line to prevent further changes.

Getting Information About Library Blocks

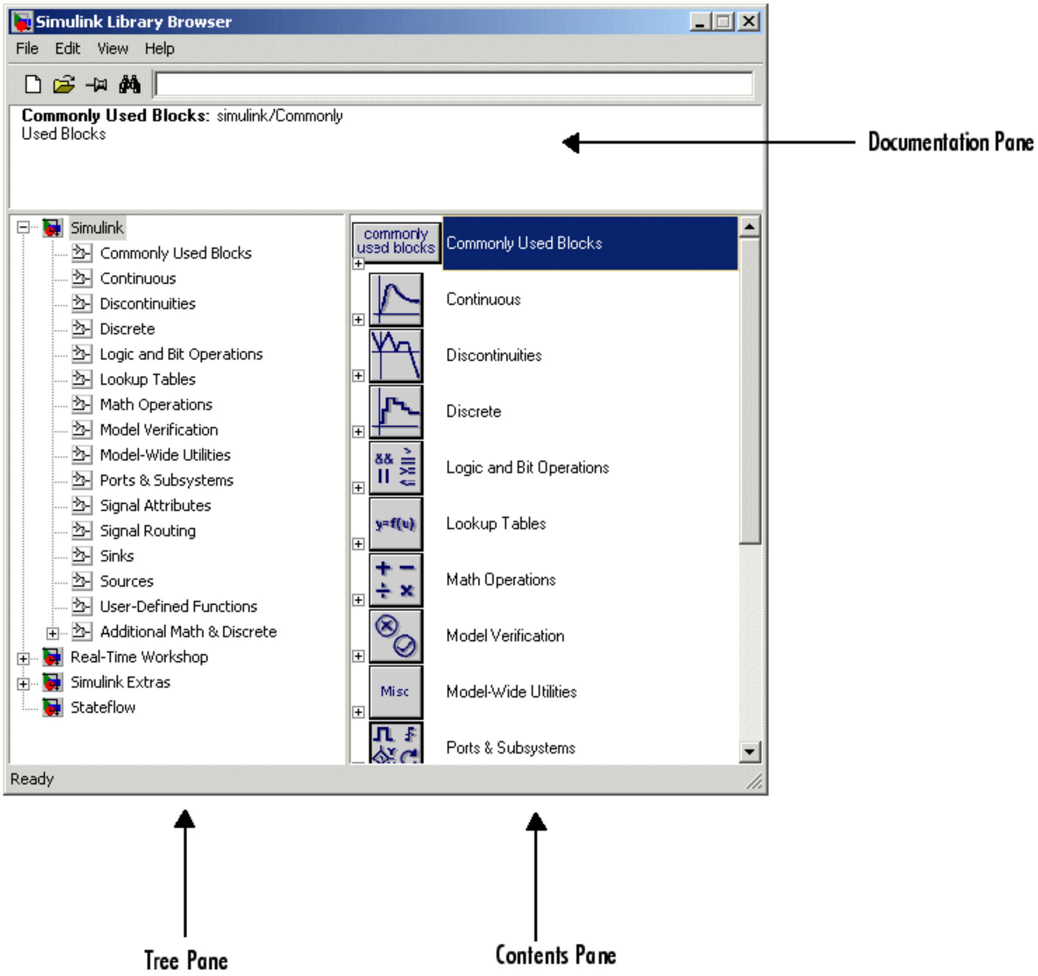
Use the `libinfo` command to get information about reference blocks in a system.

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

To cause your own library to appear in the Library Browser:

1. Create a directory in the MATLAB path for the library that you want to appear in the Library Browser.

   Each library that you want to appear in the Library Browser must be stored in its own directory on the MATLAB path. In other words, two libraries cannot exist in the same directory.

2. Create or copy the library into its directory.

   The directory for each library to be displayed in the Library Browser must contain a file named `slblocks.m` that describes the library to be displayed. The easiest way for you to create such a file is to use an existing slblocks.m file as a template and edit it to describe your library. The next two steps direct you to perform this task.

3. Create a copy of the *`matlabroot`*`/toolbox/simulink/blocks/slblocks.m` file in the library's directory.

   The file that you have copied is the `slblocks.m` file for the standard Simulink libraries.

4. Edit the file as necessary to specify the name, open function, mask, and structure of your library.

   The comments in the `slblocks.m` file explain how to specify the information about your library that the Library Browser needs.

Sample slblocks.m file. The following slblocks.m file describes a custom block library named "My Library."

```
function blkStruct = slblocks


%SLBLOCKS Defines a block library.
```

```
% Library's name. The name appears in the Library Browser's
% contents pane.

blkStruct.Name = ['My' sprintf('\n') 'Library'];

% The function that will be called when the user double-clicks on
% the library's name. ;

blkStruct.OpenFcn = 'mylib';

% The argument to be set as the Mask Display for the subsystem. You
% may comment this line out if no specific mask is desired.
% Example: blkStruct.MaskDisplay =
'plot([0:2*pi],sin([0:2*pi]));';
% No display for now.

% blkStruct.MaskDisplay = '';

% End of blocks
```

To find additional examples of slblocks.m files on your system, enter

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

at the MATLAB command prompt.

Accessing Block Data During Simulation

Simulink provides an application programming interface, called the block run-time interface, that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to access block run-time data from the MATLAB command line, the Simulink Debugger, and from Level-2 M-file S-functions (see "Writing S-Functions in M" in the online Simulink documentation).

Note You can use this interface even when the model is paused or is running or paused in the debugger.

See the following topics for more information:

- "About Block Run-Time Objects" on page 4-57
- "Accessing a Run-Time Object" on page 4-57
- "Listening for Method Execution Events" on page 4-58
- "Synchronizing Run-Time Objects and Simulink Execution" on page 4-59

About Block Run-Time Objects

The block run-time interface consists of a set of Simulink data object classes (see "Working with Data Objects" on page 7-12) whose instances provide data about the blocks in a running model. In particular, the interface associates an instance of `Simulink.RunTimeBlock`, called the block's run-time object, with each nonvirtual block in the running model. A run-time object's methods and properties provide access to run-time data about the block's I/O ports, parameters, sample times, and states.

Accessing a Run-Time Object

Every nonvirtual block in a running model has a RuntimeObject parameter whose value, while the simulation is running, is a handle for the blocks' run-time object. This allows you to use `get_param` to obtain a block's run-time object. For example, the following statement

```
rto = get_param(gcb,'RuntimeObject');
```

returns the run-time object of the currently selected block.

Note Virtual blocks (see "Virtual Blocks" on page 4-2) do not have run-time objects. Blocks eliminated during model compilation as an optimization also do not have run-time objects (see "Block reduction" on page 11-88). A run-time object exists only while the model containing the block is running or paused. If the model is stopped, `get_param` returns an empty handle. When you stop or pause a model, all existing handles for run-time objects become empty.

Listening for Method Execution Events

One application for the block run-time API is to collect diagnostic data at key points during simulation, such as the value of block states before or after blocks compute their outputs or derivatives. The block run-time API provides an event-listener mechanism that facilitates such applications. For more information, see the *Simulink Reference* for the `add_exec_event_listener` command. For an example of using method execution events, enter

```
sldemo_msfcn_lms
```

at the MATLAB command line. This Simulink model contains the S-function `adapt_lms.m`, which performs a system identification to determine the coefficients of an FIR filter. The S-function's `PostPropagationSetup` method initializes the block run-time object's Dwork vector such that the second vector stores the filter coefficients calculated at each time step.

In the Simulink model, double-clicking on the annotation below the S-function block executes its `OpenFcn`. This function first opens a figure for plotting the FIR filter coefficients. It then executes the function `add_adapt_coef_plot.m` to add a `PostOutputs` method execution event to the S-function's block run-time object using the following lines of code.

```
% Get the full path to the S-function block
blk = 'sldemo_msfcn_lms/LMS Adaptive';

% Attach the event-listener function to the S-function
h   = add_exec_event_listener(blk, ...
        'PostOutputs', @plot_adapt_coefs);
```

The function plot_adapt_coefs.m is registered as an event listener that is executed after every call to the S-function's Outputs method. The function accesses the block run-time object's Dwork vector and plots the filter coefficients calculated in the Outputs method. The calling syntax used in plot_adapt_coefs.m follows the standard needed for any listener. The first input argument is the S-function's block run-time object, and the second argument is a structure of event data, as shown below.

```
function plot_adapt_coefs(block, eventData)

% The figure's handle is stored in the block's UserData
hFig  = get_param(block.BlockHandle,'UserData');
tAxis = findobj(hFig, 'Type','axes');

tAxis = tAxis(2);
tLines = findobj(tAxis, 'Type','Line');

% The filter coefficients are stored in the block run-time
%   object's second Dwork vector.
est = block.Dwork(2).Data;

set(tLines(3),'YData',est);
```

Synchronizing Run-Time Objects and Simulink Execution

Run-time objects can be used at the MATLAB command line to obtain the value of a block's output by entering the following commands.

```
rto = get_param(gcb,'RuntimeObject')
rto.OutputPort(1).Data
```

However, the displayed data may not be the block's true output if the run-time object is not sychronized with the Simulink execution. Simulink only ensures the run-time object and Simulink execution are synchronized when the run-time object is used either within a Level-2 M-file S-function or in an event listener callback. When called at the MATLAB command line, the run-time object can return incorrect output data if other blocks in the model are allowed to share memory.

To ensure the `Data` field contains the correct block output, turn off the **Signal storage reuse** option (see “Signal storage reuse”) on the **Optimization** pane in the **Configuration Parameters** dialog box.

5

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 blocks.

Composite Signals

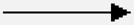
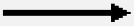
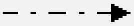
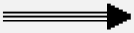
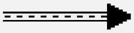
Simulink provides two capabilities, muxes and buses, that you can use to group multiple signals into a composite signal, route the composite signal from block to block, and extract constituent signals from the composite where needed. Composite signals have no functional effect, but can simplify the appearance of a model when many parallel signals exist. See Chapter 6, "Using Composite Signals" for details.

Creating Signals

You can create signals by creating source blocks in your model. For example, you can create a signal that varies sinusoidally with time by dragging an instance of the Sine block from the Simulink Sources library into the model. See "Sources" for information on blocks that you can use to create signals in a model. You can also use the Signal & Scope Manager to create signals in your model without using blocks. See "Signal & Scope Manager" on page 5-20 for more information.

Signal Line Styles

Simulink uses a variety of line styles to display different types of signals in the model window. Assorted line styles help you to differentiate the signal types in Simulink diagrams. The signal types and their line styles are as follows:

Signal Type	Line Style	Description
Scalar and Nonscalar		Simulink uses a thin, solid line to represent a diagram's scalar and nonscalar signals.
Nonscalar		When the **Wide nonscalar lines** option is enabled, Simulink uses a thick, solid line to represent a diagram's nonscalar signals. See also "Using Muxes" on page 6-3.
Control		Simulink uses a thin, dash-dot line to represent a diagram's control signals.
Virtual		Simulink uses a triple line with a solid core to represent a diagram's virtual signal buses. See "Using Buses" on page 6-5.
Nonvirtual		Simulink uses a triple line with a dotted core to represent a diagram's nonvirtual signal buses. See "Using Buses" on page 6-5.

Other than using the **Wide nonscalar lines** option to display nonscalar signals as thick, solid lines, you cannot customize or control the line style with which Simulink displays signals. See "Wide nonscalar lines" on page 5-59 for more information about this option.

Note As you construct a block diagram, Simulink uses a thin, solid line to represent all signal types. The lines are then redrawn using the specified line styles only after you update or start simulation of the block diagram.

Signal Labels

A signal label is text that appears next to the line that represents a signal that has a name. The signal label displays the signal's name. In addition, if the signal is a virtual signal (see "Virtual Signals" on page 5-10) and its **Show propagated signals** property is on (see "Show propagated signals" on page 5-43), 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 5-42). You can change the signal’s name by editing its label on the block diagram. To edit the label, left-click the label. Simulink replaces the label with an edit field. Edit the name in the edit field, then click outside the label to apply the change.

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 Simulink block 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-, two-, or multidimensional signals. A one-dimensional (1-D) signal consists of a stream of one-dimensional arrays output at a frequency of one array (vector) per simulation time step. A two-dimensional (2-D) signal consists of a stream of two-dimensional arrays output at a frequency of one 2-D array (matrix) per block sample time. A multidimensional signal consists of a stream of multidimensional (2 or more dimensions) arrays output at a frequency of one array per block sample time (see “Multidimensional Arrays” in the MATLAB programming documentation for information on multidimensional arrays). The Simulink user interface and documentation generally refer to 1-D signals as *vectors* and 2-D or multidimensional signals as *matrices*. A one-element array is frequently referred to as a *scalar*. A *row vector* is a 2-D array that has one row. A *column vector* is a 2-D array that has one column.

Only the following Simulink blocks support multidimensional signals. Simulink supports signals with up to 32 dimensions. Do not use signals with more than 32 dimensions.

- Abs
- Assignment
- Bitwise Operator
- Bus Assignment
- Bus Creator
- Bus Selector
- Compare to Constant
- Compare to Zero
- Complex to Magnitude-Angle
- Complex to Real-Imag
- Concatenate
- Constant
- Data Store Memory
- Data Store Read
- Data Store Write
- Data Type Conversion
- Embedded MATLAB Function
- Environment Controller
- From
- From Workspace
- Gain (only if the **Multiplication** parameter specifies `Element-wise(K*u)`)
- Goto
- Ground
- IC

- Inport
- Level-2 M-File S-Function
- Logical Operator
- Magnitude-Angle to Complex
- Manual Switch
- Math Function (no multidimensional signal support for the `transpose` and `hermitian` functions)
- Memory
- Merge
- MinMax
- Model
- Multiport Switch
- Outport
- Product, Product of Elements — only if the **Multiplication** parameter specifies `Element-wise`
- Probe
- Random Number
- Rate Transition
- Real-Imag to Complex
- Relational Operator
- Reshape
- Scope, Floating Scope
- Selector
- S-Function
- Signal Conversion
- Signal Specification
- Slider Gain

- Squeeze
- Subsystem, Atomic Subsystem, CodeReuse Subsystem
- Add, Subtract, Sum, Sum of Elements — along specified dimension
- Switch
- Terminator
- To Workspace
- Trigonometric Function
- Unary Minus
- Uniform Random Number
- Unit Delay
- Width

Simulink blocks vary in the dimensionality of the signals they can accept or output. Some blocks can accept or output signals of any dimensions. Some can accept or output only scalar or vector signals. To determine the signal dimensionality of a particular block, see the block's description in Blocks — Alphabetical List in the online Simulink reference. See "Determining Output Signal Dimensions" on page 5-15 for information on what determines the dimensions of output signals for blocks that can output nonscalar signals.

Note Simulink does not support dynamic signal dimensions during simulation. That is, the size of a signal must remain constant while the simulation executes. You can alter a signal's size only after terminating the simulation.

Complex Signals

The values of Simulink signals can be complex numbers. A signal whose values are complex numbers is called a complex signal. You can introduce a complex-valued signal into a model in the following ways:

- Load complex-valued signal data from the MATLAB workspace into the model via a root-level Inport block.

- Create a Constant block in your model and set its value to a complex number.
- Create real signals corresponding to the real and imaginary parts of a complex signal, then combine the parts into a complex signal, using the Real-Imag to Complex conversion block.

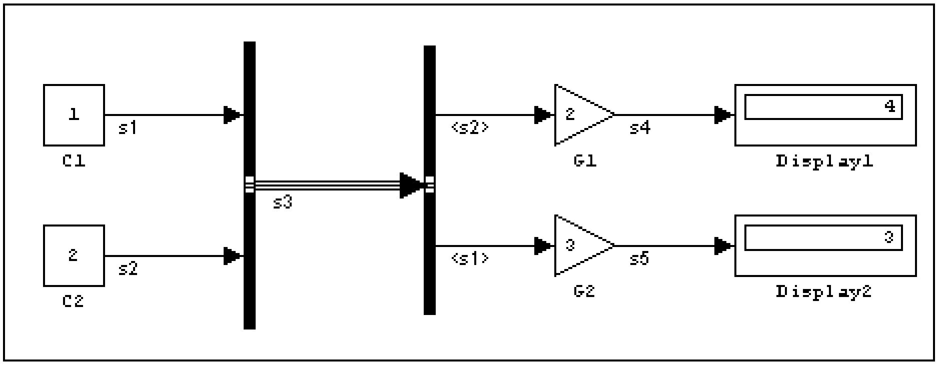
You can manipulate complex signals via blocks that accept them. If you are not sure whether a block accepts complex signals, see the documentation for the block in Blocks — Alphabetical List in the online Simulink reference.

Virtual Signals

A *virtual signal* is a signal that represents another signal graphically. Some blocks, such as Bus Creator, Inport, and Outport blocks, generate virtual signals either exclusively or optionally. Virtual signals are purely graphical entities; they have no mathematical or physical significance. Simulink ignores them when simulating a model, and they do not exist in generated code.

Whenever you update or run a model, Simulink determines the nonvirtual signal(s) represented by the model's virtual signal(s), using a procedure known as *signal propagation*. When running the model, Simulink uses the corresponding nonvirtual signal(s), determined via signal propagation, to drive the blocks to which the virtual signals are connected.

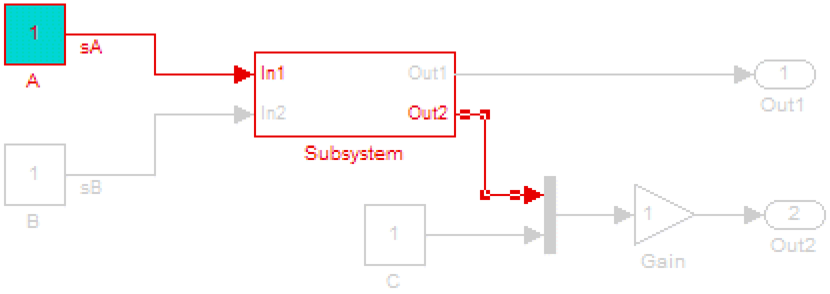
Consider, for example, the following model.



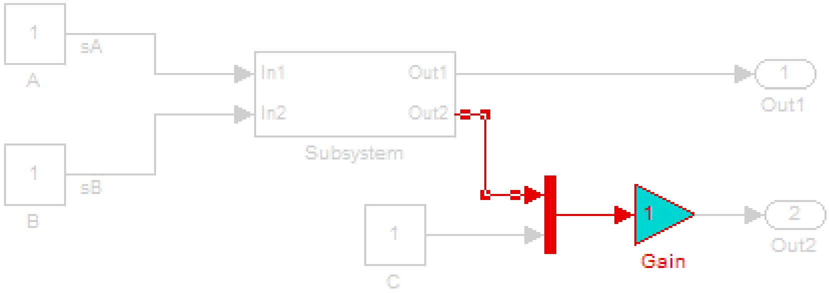
The signals driving Gain blocks G1 and G2 are virtual signals corresponding to signals s2 and s1, respectively. Simulink determines this automatically whenever you update or simulate the model.

Displaying Virtual Signal Sources and Destinations

To display the nonvirtual block whose output is the source of a signal, select the signal and then select **Highlight to Source** from the signal's context menu. Simulink highlights the path from the signal's source block to the signal itself.



To display the nonvirtual block that is the destination of a signal, select the signal and then select **Highlight to Destination** from the signal's context menu. Simulink highlights the path from the selected signal to the nonvirtual block that it feeds.

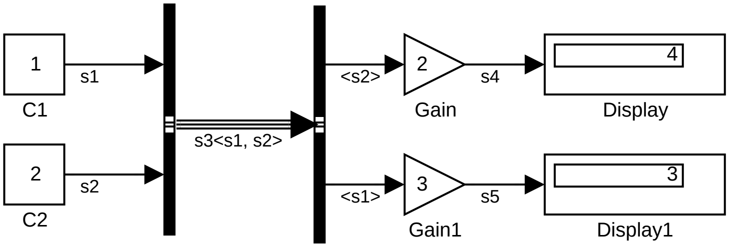


To remove the highlighting, select **Remove Highlighting** from the block diagram's context menu.

Note If the path from a source block or to a destination block includes an unresolved reference to a library block, Simulink highlights the path only from or to the library block, respectively. This is to avoid time-consuming library reference resolution while you are editing a model. To permit Simulink to display the complete path, first update the diagram (e.g., by pressing **Ctrl+D)**. This causes Simulink to resolve all library references and hence display the complete path to a destination block or from a source block.

Displaying the Nonvirtual Components of Virtual Signals

The **Show propagated signals** option (see "Signal Properties Dialog Box" on page 5-42) displays the nonvirtual signals represented by virtual signals in the labels of the virtual signals.

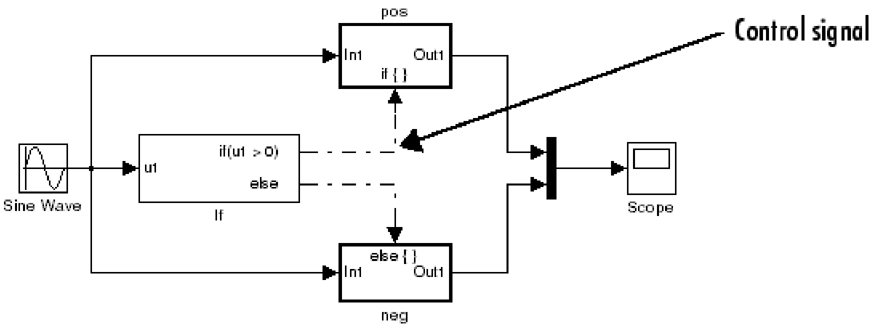


When you change the name of a nonvirtual signal, Simulink immediately updates the labels of all virtual signals that represent the nonvirtual signal and whose **Show propagated signals** is on, except if the path from the nonvirtual signal to the virtual signal includes an unresolved reference to a library block. In such cases, to avoid time-consuming library reference resolutions while you are editing a block diagram, Simulink defers updating the virtual signal's label until you update the model's block diagram either directly (e.g., by pressing **Ctrl+D**) or by simulating the model.

Note Virtual signals can represent virtual as well as nonvirtual signals. For example, you can use a Bus Creator block to combine multiple virtual and nonvirtual signals into a single virtual signal. If during signal propagation, Simulink determines that a component of a virtual signal is itself virtual, Simulink uses signal propagation to determine the nonvirtual components of the virtual component. This process continues until Simulink has determined all nonvirtual components of a virtual signal.

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.



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.
Signal bus	A composite signal made up of other signals, including other composite signals. You can use Bus Creator and Inport blocks to create signal buses. See “Using Buses” on page 6-5.
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.
Test point	A signal that must be accessible during simulation. See “Working with Test Points” on page 5-56 for more information.
Vector	One-dimensional signal array.
Virtual signal	Signal that represents another signal or set of signals.
Width	Size of a vector signal.

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.

For more information, see

- "Determining the Output Dimensions of Source Blocks" on page 5-15
- "Determining the Output Dimensions of Nonsource Blocks" on page 5-16
- "Signal and Parameter Dimension Rules" on page 5-16
- "Scalar Expansion of Inputs and Parameters" on page 5-17

Determining the Output Dimensions of Source Blocks

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

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

Constant Value	Interpret Vector Parameters as 1-D	Output
scalar	off	one-element array
scalar	on	one-element array
1-by-N matrix	off	1-by-N matrix
1-by-N matrix	on	N-element vector
N-by-1 matrix	off	N-by-1 matrix
N-by-1 matrix	on	N-element vector
M-by-N matrix	off	M-by-N matrix
M-by-N matrix	on	M-by-N matrix

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

Determining the Output Dimensions of Nonsource Blocks

If a block has inputs, the dimensions of its outputs are, after scalar expansion, the same as those of its inputs. (All inputs must have the same dimensions, as discussed in “Signal and Parameter Dimension Rules” on page 5-16).

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 5-18) 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 5-18) 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 11-120 for more information.

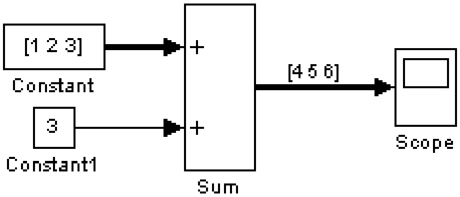
Scalar Expansion of Inputs and Parameters

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

Scalar Expansion of Inputs

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

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

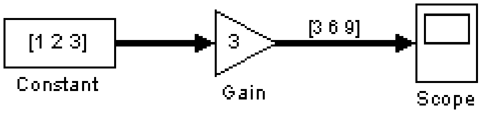


When a block's output is a function of a parameter and the parameter is nonscalar, Simulink expands a scalar input to match the dimensions of the parameter. For example, Simulink expands a scalar input to a Gain block to match the dimensions of a nonscalar gain parameter.

Scalar Expansion of Parameters

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

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

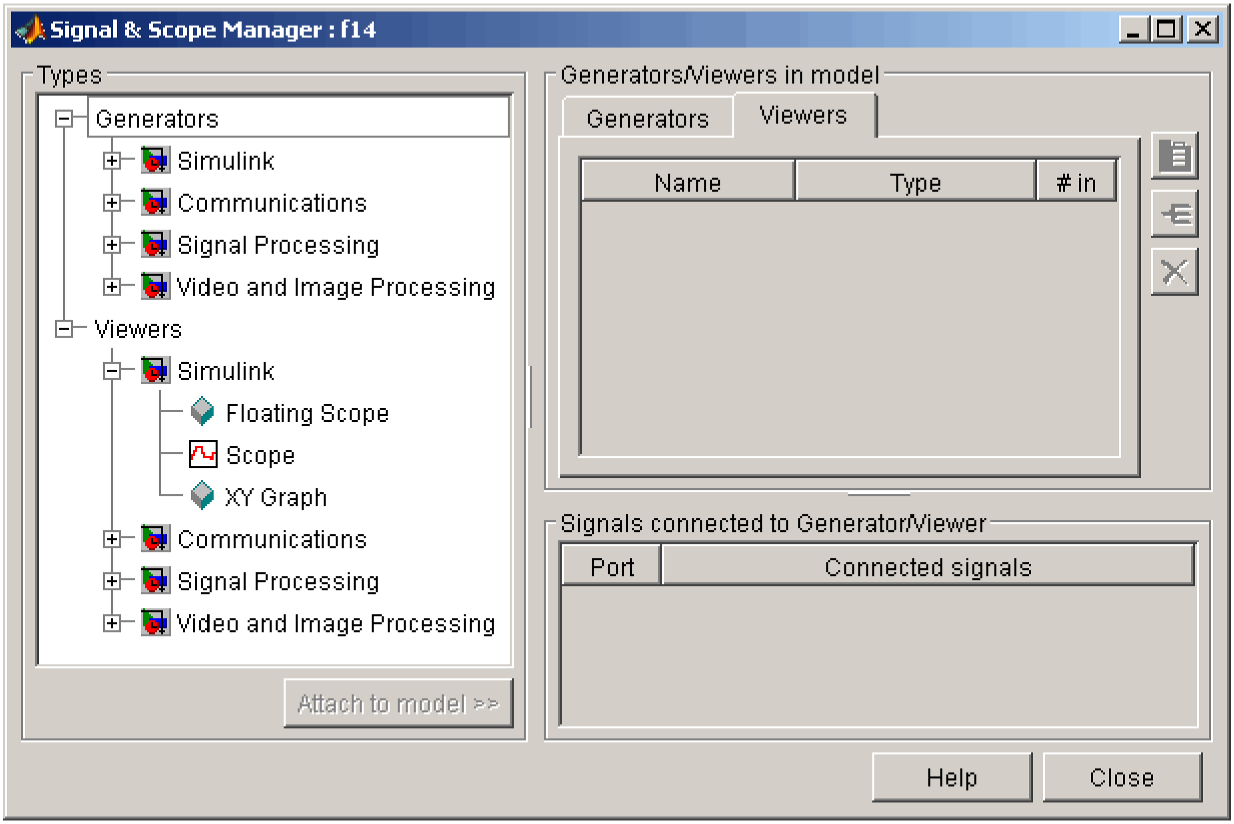


Signal & Scope Manager

The Signal & Scope Manager lets you globally manage signal generators and viewers.

Note The Signal & Scope Manager requires that you have Java enabled when you start MATLAB. This is the default.

To display the Signal & Scope Manager, from the model editor's **Tools** or context menu select **Signal & Scope Manager**. The Signal & Scope manager appears.



The **Types** pane shows the generators and viewers associated with the products installed on your system. Expand a products node list to show the generators and viewers available to you.

To...	See...
Connect a signal to a viewer or generator	"Connecting Viewers and Generators" on page 5-21
Edit generators and viewers associated with your model	"Generator and Viewer Objects" on page 5-22
Display signals connected to a generator or viewer	"Signals connected to Generator/Viewer" on page 5-25
Create custom viewers and generators	"Adding Custom Viewers and Generators to the Signal & Scope Manager" on page 5-27
Log data to a file for postprocessing	"Logging Signals" on page 5-34
Obtain data from a simulation while it is running	"Extracting Partial Data from a Running Simulation" on page 5-41

Connecting Viewers and Generators

Signals can be connected to new viewers or generators, and they can be added to viewers already in the model.

To connect a new viewer or generator to a signal in the currently selected model:

1. From the model editor's **Tools** or context menu, select **Signal & Scope Manager** to display the Signal & Scope manager.
2. Select a scope or generator from the **Types** pane.
3. Click **Attach to Model**.
4. Use the Signal Selector (see "The Signal Selector" on page 5-30) to connect the scope or generator.

To connect a new viewer by selecting a signal:

1. Right-click a signal to display the signal context menu.
2. Select **Create & Connect Viewer**.
3. Select the viewer.

To connect to an existing scope or generator:

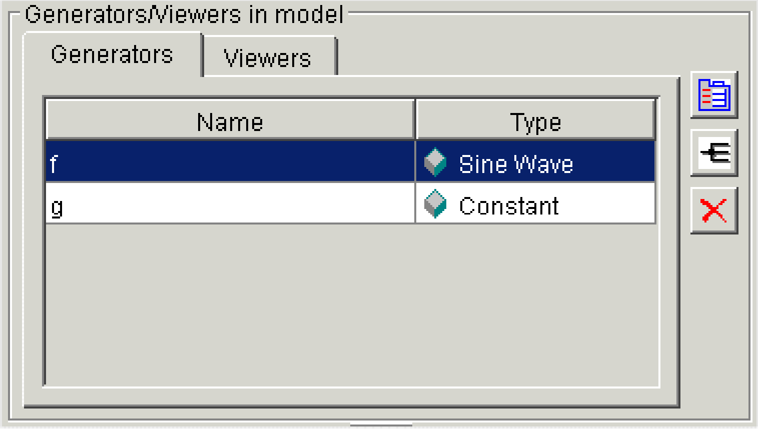
1. Right-click a signal to display the signal context menu.
2. Select **Connect to Existing Viewer**.
3. Select the viewer and axis from the list.

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.

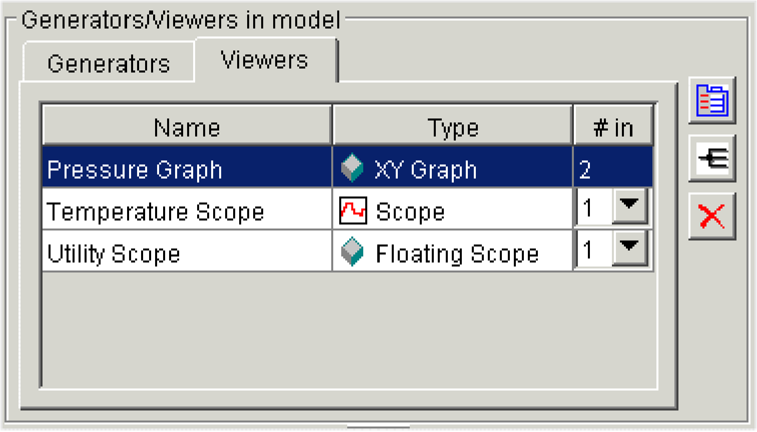


Name	Type
f	Sine Wave
g	Constant

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 *Simulink Reference* 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. You can connect multiple signals to the axes of a Simulink Scope viewer. 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.

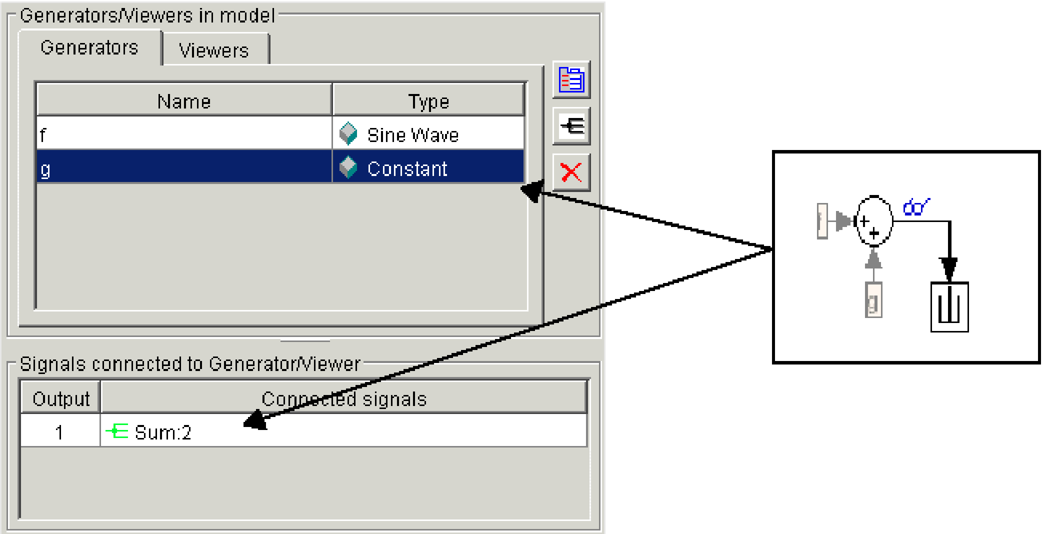
Edit Menu

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 Generators/Viewers control group of the Signal & Scope Manager.



Name	Type
f	Sine Wave
g	Constant

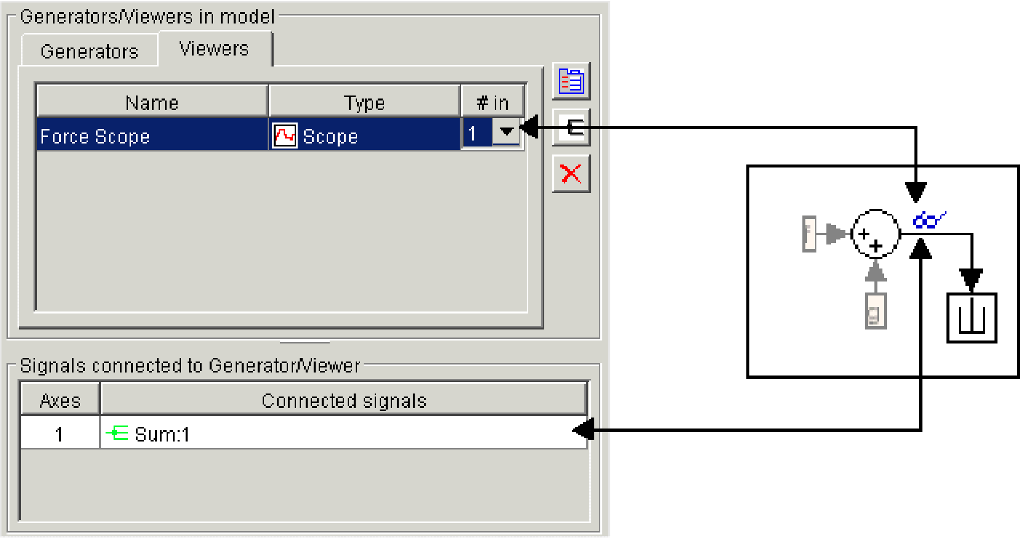
Signals connected to Generator/Viewer

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

Name	Type	# in
Force Scope	Scope	1

Signals connected to Generator/Viewer

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 **Highlight signal in model** from the menu. To open the Signal Selector, select **Edit Signal Connections** from the model.

Adding Custom Viewers and Generators to the Signal & Scope Manager

You can add custom signal viewers or generators so that they appear in the Signal & Scope Manager. The following procedure assumes that you have a custom viewer named `newviewer` that you want to add.

1. Open a new Simulink library.

   For example, open the Simulink browser and select **File > New > Library**.

2. Save the library.

   For example, save it as `newlib`.

3. In the MATLAB Command Window, set the library type for the library.

   For example, to set the library type of `newviewer` to `viewer`,

   ```
   set_param('newlib','LibraryType','SSMgrViewerLibrary')
   ```

   To set library type for generators, use the type `'SSMgrGenLibrary'`.

   For example,

   ```
   set_param('newlib','LibraryType','SSMgrGenLibrary')
   ```

4. Set the display name of the library.

   For example,

   ```
   set_param('newlib','SSMgrDisplayString','My Custom Library')
   ```

5. Add the viewer or generator to the library.

   Note If the viewer is a compound viewer, such as a subsystem with multiple blocks, make the top-level subsystem an atomic one.

6. Set the `iotype` of the viewer.

   For example,

```
set_param('newlib/newviewer','iotype','viewer')
```

7 Save the library `newlib`. In the Simulink window, select **File > Save**.

8 Using the MATLAB editor, create a file named `sl_customization.m`. In this file, enter a directive to incorporate the new library as a viewer library.

For example, to save `newlib` as a viewer library, add the following lines:

```
function sl_customization(cm)
cm.addSigScopeMgrViewerLibrary('newlib')
%end function
```

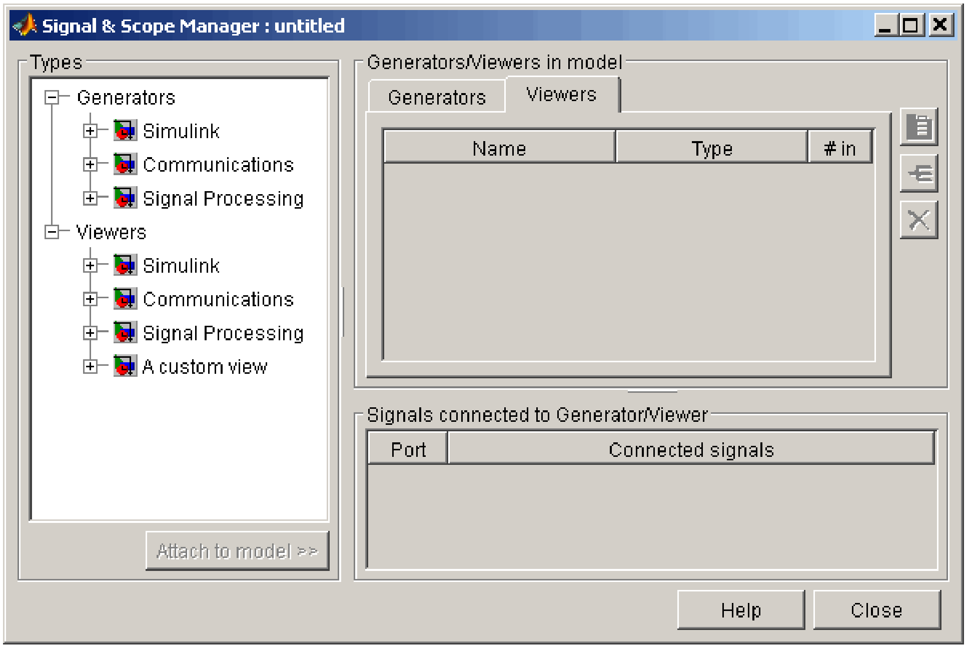
To add a library as a generator library, add a line like the following:

```
cm.addSigScopeMgrGeneratorLibrary('newlib')
```

9 Add a corresponding `cm.addSigScope` line for each viewer or generator library you want to add.

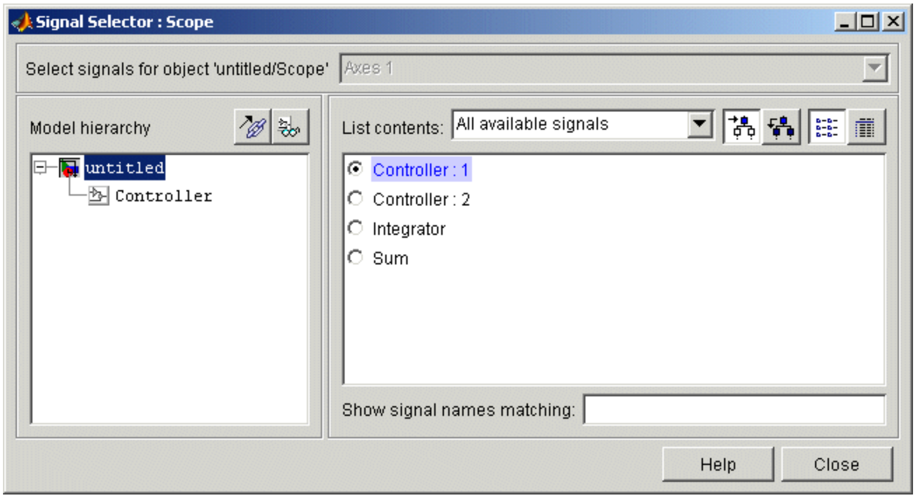
10 Save the `sl_customization.m` file on your MATLAB path. Edit this file to add new viewer or generator libraries.

11 To see the new custom libraries, restart MATLAB and start the Signal & Scope Manager.



The Signal Selector

The Signal Selector allows you to connect a generator or viewer object (see “Signal & Scope Manager” on page 5-20) or the Floating Scope to block inputs and outputs. It appears when you click the **Signal selection** button for a generator or viewer object in the Signal & Scope Manager or on the toolbar of the Floating Scope’s window.



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:

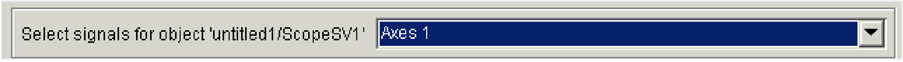
The Signal Selector that appears when you click the **Signal selection** button applies only to the currently selected generator or viewer object (or the Floating Scope). If you want to connect blocks to another generator or viewer object, you must select the object in the Signal & Scope Manager and launch another instance of the Signal Selector. The object used to launch a particular instance of the Signal Selector is called that instance’s owner.

The Signal Selector includes the following controls.

- “Port/Axis Selector” on page 5-31
- “Model Hierarchy” on page 5-31
- “Inputs/Signals List” on page 5-32

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.

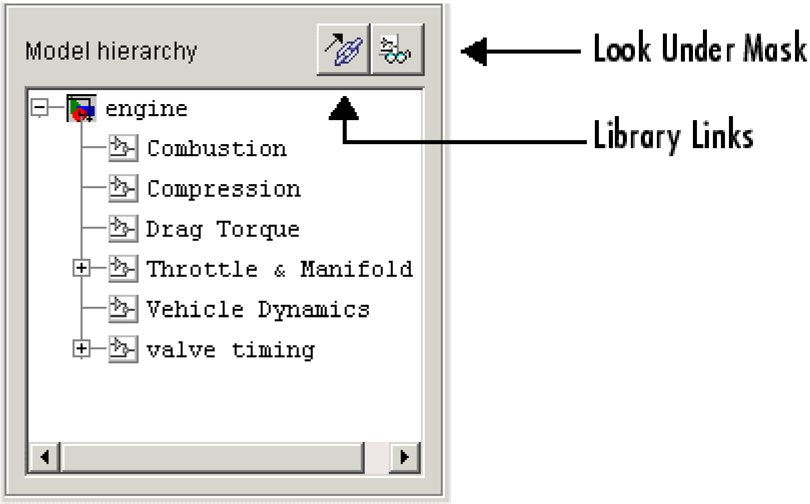


Select signals for object 'untitled1/ScopeSV1' Axes 1

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.

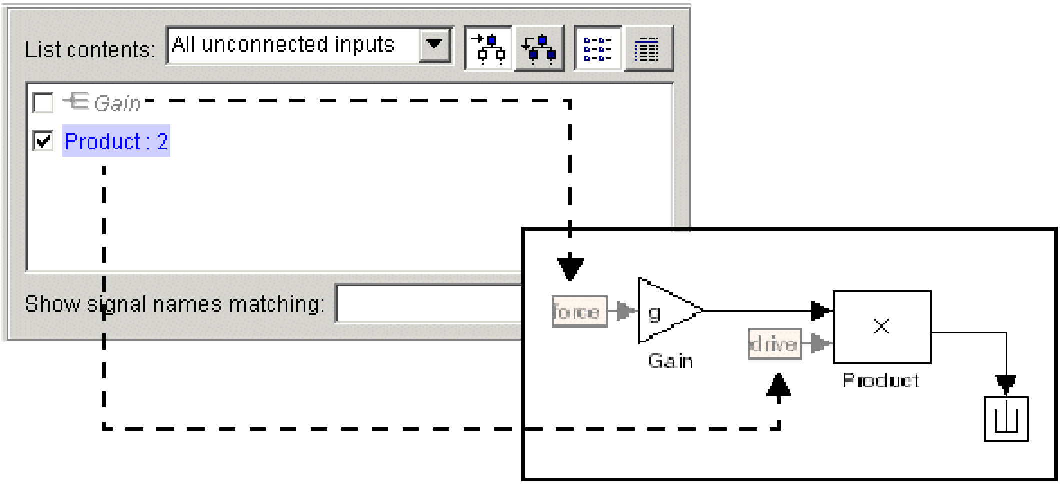


Selecting a subsystem causes the adjacent port list to display the ports available for connection in the selected subsystem. To display subsystems included as library links in your model, click the **Library Links** button at the top of the **Model hierarchy** control. To display subsystems contained by masked subsystems, click the **Look Under Masks** button at the top of the panel.

Inputs/Signals List

The contents of this panel displays input ports available for connection to the Signal Selector's owner if the owner is a signal generator or signals available for connection to the owner if the owner is a signal viewer.

If the Signal Selector's owner is a signal generator, the inputs/signals list by default lists each input port in the system selected in the model hierarchy tree that is either unconnected or connected to a signal generator.


List contents: All unconnected inputs
□ Gain
☑ Product : 2
Show signal names matching:
force
g
Gain
drive
×
Product

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. Selecting the check box next to a port's entry in the list connects the Signal Selector's owner to the port, replacing, if necessary, the signal generator previously connected to the port.

To display more information on each signal, click the **Detailed view** button at the top of the pane. The detailed view shows the path and data type of each signal and whether the signal is a test point. The controls at the top and bottom of the panel let you restrict the amount of information shown in the ports list.

- To show named signals only, select `Named signals only` from the **List contents** control at the top of the pane.
- To show only signals selected in the Signal Selector, select `Selected signals only` from the **List contents** control.
- To show test point signals only, select `Testpointed signals only` from the **List contents** control.
- To show only signals whose signals match a specified string of characters, enter the characters in the **Show signals matching** control at the bottom of the **Signals** pane and press the **Enter** key.
- To show the selected types of signals for all subsystems below the currently selected subsystem in the model hierarchy, click the **Current and Below** button at the top of the **Signals** pane.

To select or deselect a signal in the **Signals** pane, click its entry or use the arrow keys to move the selection highlight to the signal entry and press the **Enter** key. You can also move the selection highlight to a signal entry by typing the first few characters of its name (enough to uniquely identify it).

Note You can continue to select and deselect signals on the block diagram with the Signal Selector open. For example, shift-clicking a line in the block diagram adds the corresponding signal to the set of signals that you previously selected with the Signal Selector. If the simulation is running when you open and use the Signal Selector, Simulink updates the Signal Selector to reflect signal selection changes you have made on the block diagram. However, the changes do not appear until you select the Signal Selector window itself. You can also use the Signal Selector before running a model. If no simulation is running, selecting a signal in the model does not change the Signal Selector.

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, Scope block or viewer, or root-level Outport 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 *Simulink Reference* for the To Workspace and Outport blocks for information on using these blocks to log signal data.

For more information, see

- "Signal Logging Limitations" on page 5-34
- "Enabling Signal Logging" on page 5-35
- "Specifying a Logging Name" on page 5-36
- "Limiting the Data Logged for a Signal" on page 5-36
- "Logging Referenced Model Signals" on page 5-36
- "Viewing Logged Signal Data" on page 5-38
- "Accessing Logged Signal Data" on page 5-38
- "Example: Logging Signal Data in the F14 Model" on page 5-39
- "Extracting Partial Data from a Running Simulation" on page 5-41

Signal Logging Limitations

Simulink does not support signal logging for the following types of signals:

- Output of a Function-Call Generator block
- Signal connected to the input of a Merge block
- Outputs of Trigger and Enable blocks

Enabling Signal Logging

To enable signal logging for a signal, select the **Log signal data** option on the signal's **Signal Properties** dialog box (see "Signal Properties Dialog Box" on page 5-42).

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 "Designating a Signal as a Test Point" on page 5-56 for more information).

Globally Enabling and Disabling Signal Logging

You can globally enable or disable signal logging for a model by checking or unchecking the **Signal logging** option on the **Data Import/Export** pane of the **Configuration Parameters** dialog box (see "Signal logging" on page 11-83). Simulink logs signals only if this option is checked. If the option is not checked, Simulink ignores the signal logging settings for individual signals.

Enabling Signal Logging Programmatically

You can enable signal logging programmatically for selected blocks with the outport `DataLogging` property. You can set this property using the `set_param` command. For example:

1. At the MATLAB Command Window, open a model. Type

   ```
   vdp
   ```

2. Select a block in that model. For example, select the Mux block.

3. Get the port handles of the selected block.

   ```
   get_param(gcb,'PortHandles')
   ```

4. Enable signal logging for the desired outport port.

   ```
   set_param(ans.Outport(1),'DataLogging','on')
   ```

   The logged signal indicator (⇡) appears.

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 (mux or bus) that has another element of the same name, you must specify a unique log name for the signal. See Chapter 6, "Using Composite Signals" for details about muxes and buses.

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.

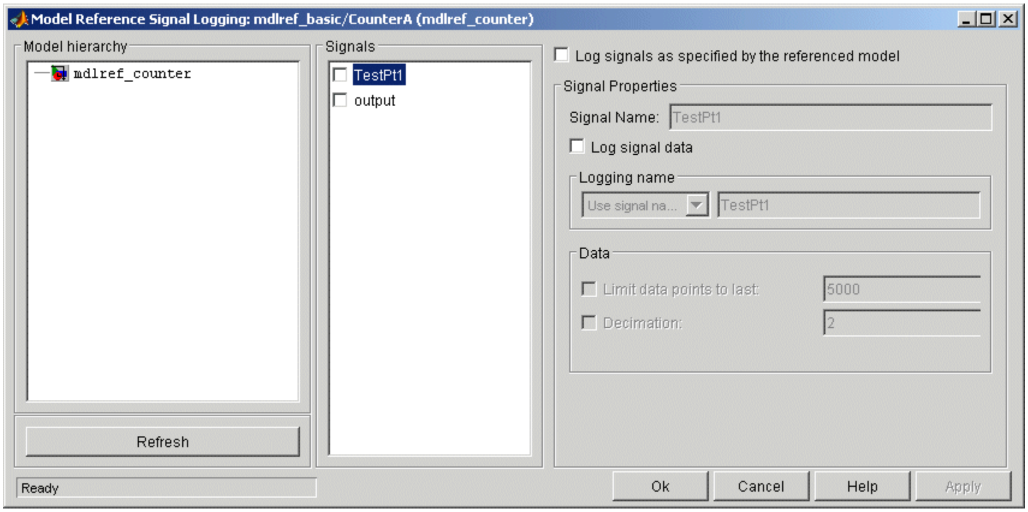
Limiting the Data Logged for a Signal

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 5-45 for more information.

Logging Referenced Model Signals

You can log any signal in a referenced model that has been test pointed (see "Designating a Signal as a Test Point" on page 5-56). To log test pointed 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.



Model Reference Signal Logging: mdlref_basic/CounterA (mdlref_counter)

Model hierarchy: mdlref_counter

Signals:
□ TestPt1
□ output

□ Log signals as specified by the referenced model

Signal Properties

Signal Name: TestPt1

□ Log signal data

Logging name: Use signal na... TestPt1

Data

□ Limit data points to last: 5000

□ Decimation: 2

Ready

The dialog box contains the following panes and controls.

Model Hierarchy

This pane displays the contents of the referenced model as a tree control with expandable nodes. The top-level node represents the referenced model. Expanding this node displays the subsystems that the referenced model contains and any models that it itself references. Expanding a subsystem node displays the subsystems that it contains and the models that it references.

Refresh Button

Refreshes the dialog box to reflect changes in the model hierarchy.

Signals

This pane displays the test points of the model or subsystem selected in the **Model Hierarchy** pane (see “Working with Test Points” on page 5-56). Check the check box next to a test point’s name to specify that it should be logged.

Log signals as specified by the referenced model

Checking this check box causes Simulink to log the signals that the referenced model specifies should be logged.

Signal Properties

This pane is enabled if **Log signals as specified by the referenced model** is not selected. In this case, the controls on this pane allow you to specify the signal logging properties of the signal selected in the **Signals** pane. The values that you specify override for this instance of the referenced model those specified by the model itself. The controls correspond to the controls of the same name on the **Signal Properties** dialog box. See “Signal Properties Dialog Box” on page 5-42 for information on how to use them.

Viewing Logged Signal Data

To view logged signal data, either check the **Inspect signals when simulation is stopped/paused** in the **Data Import/Export** pane of the **Configuration Parameters** dialog box or select **Tools > Inspect Logged Signals** from the model editor’s menu bar. The first method causes Simulink to display logged signals in the MATLAB **Time Series Tools** viewer (see “Time Series Tools” in the online MATLAB documentation) whenever a simulation ends or you pause a simulation. The second method causes Simulink to display the data immediately.

Note You must run the simulation first before selecting **Tools > Inspect Logged Signals**. Otherwise, selecting this command has no effect.

Accessing Logged Signal Data

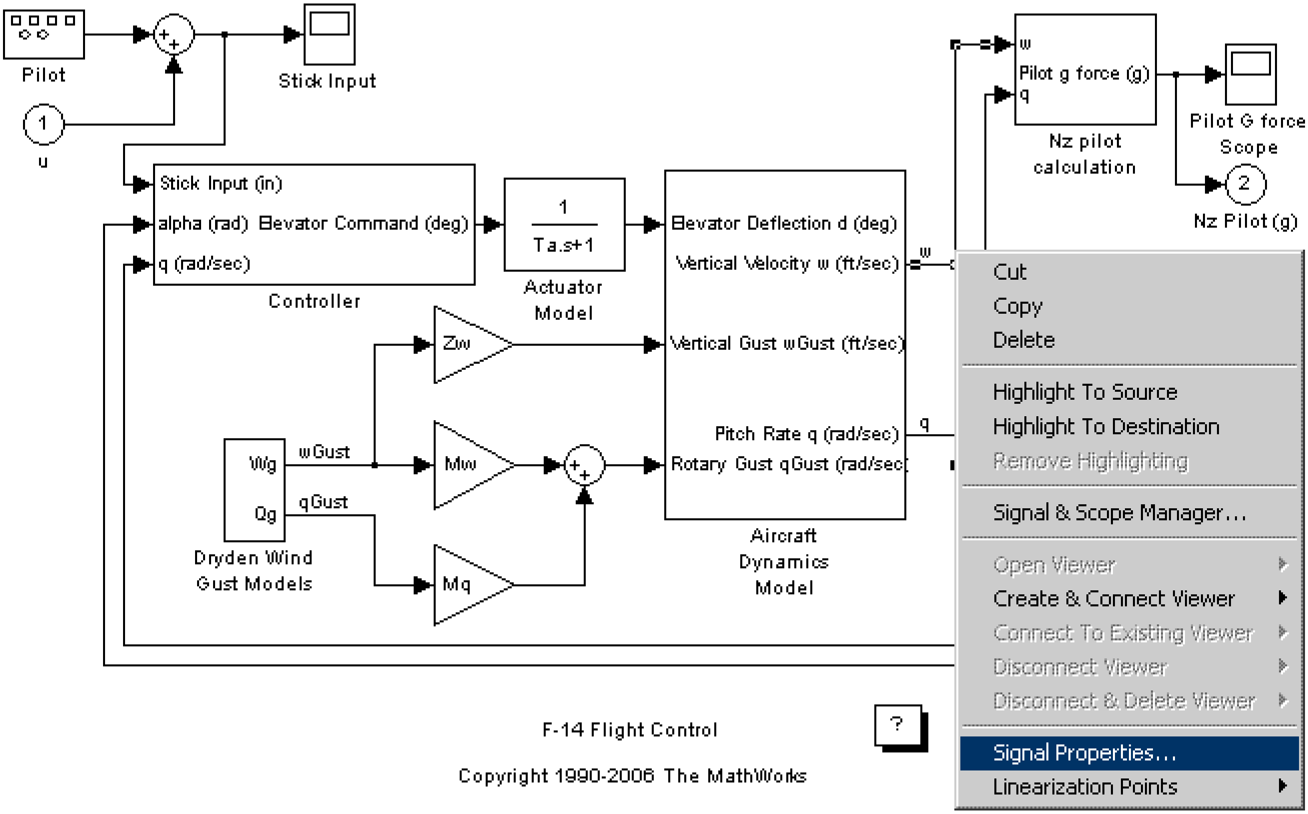
Simulink saves signal data that it logs during simulation in a Simulink data object of type `Simulink.ModelDataLogs` that resides in the MATLAB workspace. The name of the object’s handle is `logsout` by default. The **Data Import/Export** configuration pane (see “Data Import/Export Pane” on page 11-81) allows you to specify another name for this object. See `Simulink.ModelDataLogs` in the online Simulink reference for information on extracting signal data from this object.

Example: Logging Signal Data in the F14 Model

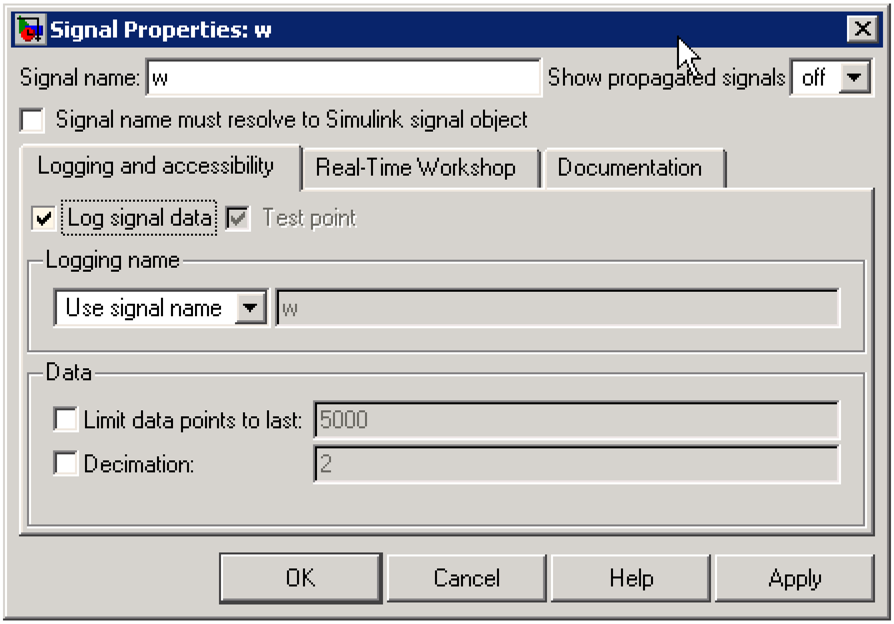
Enabling signal logging on a signal-by-signal basis allows you to store signal data without modifying the structure of the Simulink diagram. For example, use the following steps to log and access the signal data for the vertical velocity signal w in the F14 model.

1 Open the F14 model by typing f14 at the MATLAB command prompt.

2 Right-click on the signal labeled w and select the **Signal Properties** menu.



3 In the **Signal Properties** dialog box that opens, check the **Log signal data** option. Notice that the **Test point** option automatically becomes checked and the logging name initializes to the signal's name.



Signal Properties: w

Signal name: w   Show propagated signals: off

☐ Signal name must resolve to Simulink signal object

☑ Log signal data ☑ Test point

Logging name: Use signal name   w

Data

☐ Limit data points to last: 5000

☐ Decimation: 2

4 Click the **OK** button on the **Signal Properties** dialog box. The icon appears on the signal labeled `w`, indicating that this signal will be logged during simulation.

5 Ensure that the **Signal logging** option on the **Data Import/Export** pane of the **Configuration Parameters** dialog box is checked and that the logging name is set to the default variable `logsout`.

6 Run the F14 simulation. The logged signal data is stored in a `Simulink.ModelDataLogs` object named `logsout` in the MATLAB workspace. Typing `logsout` at the MATLAB command prompt displays the following

```
logsout =

Simulink.ModelDataLogs (f14):
  Name                   Elements  Simulink Class

  w                           1      Timeseries
```

7 Type `logsout.w` to view the information stored for the signal `w`.

```
logsout.w
          Name: 'w'
     BlockPath: 'f14/Aircraft Dynamics Model'
     PortIndex: 1
    SignalName: 'w'
    ParentName: 'w'
      TimeInfo: [1x1 Simulink.TimeInfo]
          Time: [1353x1 double]
          Data: [1353x1 double]
```

8 To inspect the signal using the MATLAB **Time Series Tools**, select **Inspect Logged Signals** from the `f14` model editor's **Tools** menu (see "Time Series Tools" in the online MATLAB documentation).

Extracting Partial Data from a Running Simulation

Before a simulation ends, you can extract and write the currently logged signal data (from `Simulink.ModelDataLogs`) with the `set_param` `WriteDataLogs` command. The currently logged signal is the partial data logged between when the simulation started and when you request an extraction of the signal data. If you use this command during the simulation, Simulink writes the current logging variable values to the MATLAB workspace. If you use this command at the end of the simulation, Simulink writes the values from the last simulation to the MATLAB workspace.

To use this command, type the following at the MATLAB Command Window.

```
set_param(bdroot,'SimulationCommand','WriteDataLogs')
```

Signal Properties Dialog Box

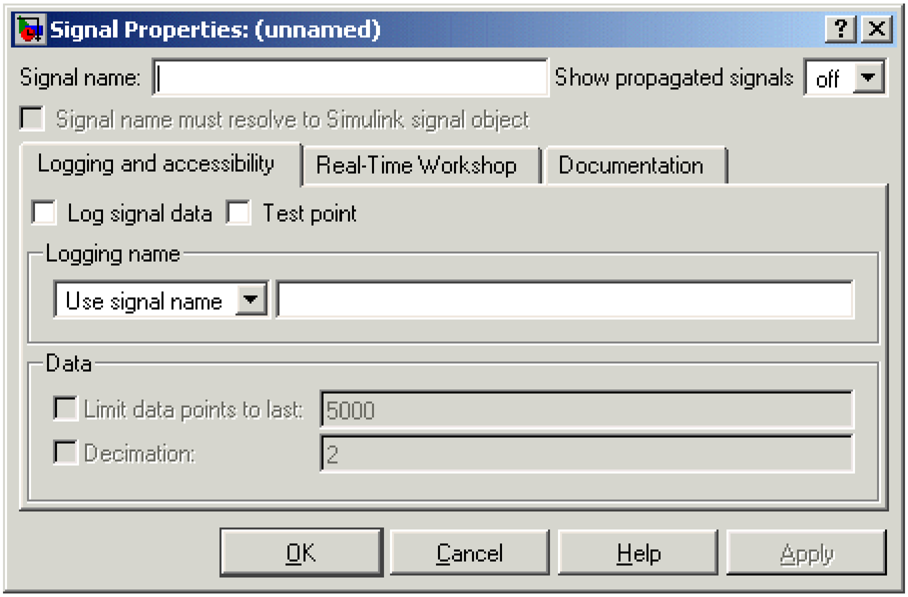
The **Signal Properties** dialog box lets you display and edit signal properties. To display the dialog box, either

- 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 a block that outputs or inputs the signal and select **Port Signal Properties** from the block's context menu, then select the port to which the signal is connected from the resulting menu

The **Signal Properties** 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 following controls.

- "Signal Properties Controls" on page 5-43

- "Logging and Accessibility Options" on page 5-44
- "Real-Time Workshop Options" on page 5-46
- "Documentation Options" on page 5-46

Signal Properties Controls

Signal name

Name of signal.

Signal name must resolve to a Simulink signal object.

Specifies that either the base MATLAB workspace or the model 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.

Note `Simulink.Signal` objects in the model workspace must have their storage class set to `Auto`. See "Working with Model Workspaces" on page 3-107 for more information.

Show propagated signals

Note This option appears only for signals that originate from a virtual block other than a Bus Selector block.

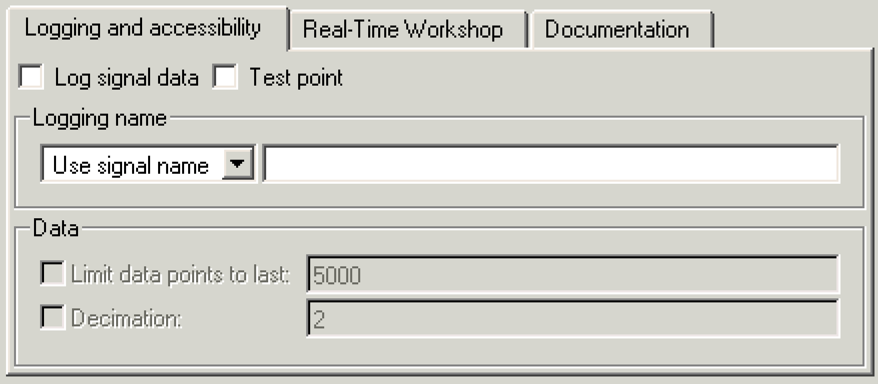
Show propagated signal names. You can select one of the following options:

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>`.

See "Displaying the Nonvirtual Components of Virtual Signals" on page 5-12 for more information.

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 5-34).

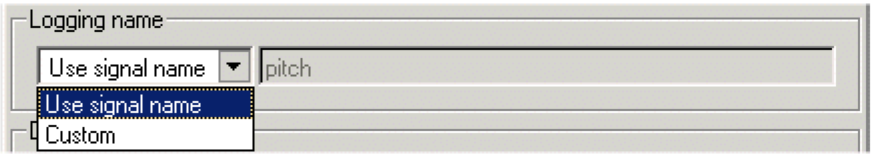
Test point

Select this option to designate this signal as a test point (see "Working with Test Points" on page 5-56).

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 name associated with logged signal data.



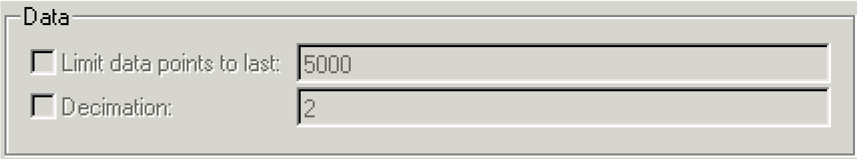
Logging name

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 as follows.

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 Nth 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 "Interfacing Signals to External Code"for an explanation of the listed options.

RTW storage type qualifier

Enter a storage type qualifier for this signal. See "Interfacing Signals to External Code" 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.

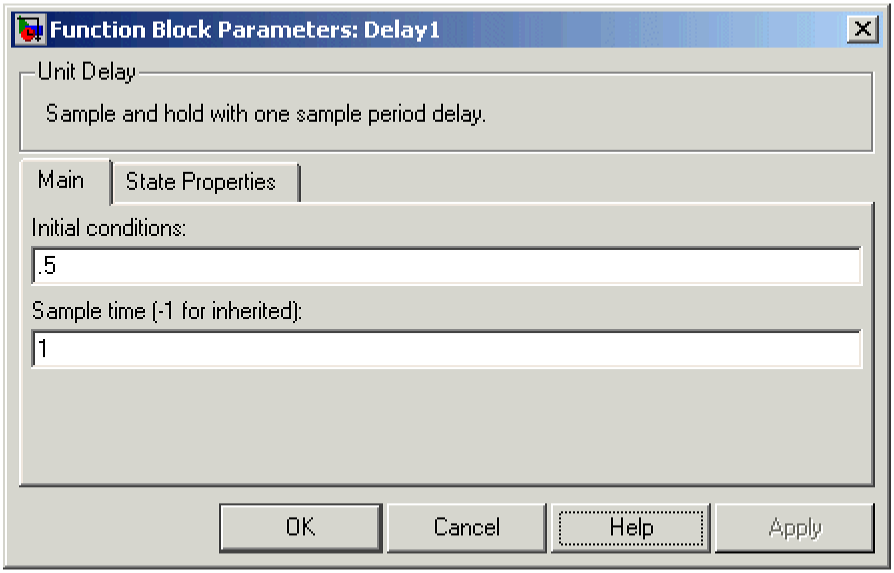
Initializing Signals and Discrete States

Simulink allows you to specify the initial values of signals and discrete states, i.e., the values of the signals and discrete states at time 0 of the simulation. You can use signal objects to specify the initial values of any signal or discrete state in a model. In addition, for some blocks, e.g., Outport, Data Store Memory, or Memory, you can use either a signal object or a block parameter or both to specify the initial value of a block state or output. In such cases, Simulink checks to ensure that the values specified by the signal object and the parameter are consistent. For more information, see

- "Using Block Parameters to Initialize Signals and Discrete States" on page 5-49
- "Using Signal Objects to Initialize Signals and Discrete States" on page 5-49
- "Using Signal Objects to Tune Initial Values" on page 5-50
- "Example: Using a Signal Object to Initialize a Subsystem Output" on page 5-52
- "Initialization Behavior Summary for Signal Objects" on page 5-53

Using Block Parameters to Initialize Signals and Discrete States

For blocks that have an initial value or initial condition parameter, you can use that parameter to initialize a signal. For example, the following Block Parameters dialog box initializes the signal for a Unit Delay block with an initial condition of 0.5.


Function Block Parameters: Delay1
Unit Delay
Sample and hold with one sample period delay.
Main
State Properties

Initial conditions: .5

Sample time (-1 for inherited): 1

Using Signal Objects to Initialize Signals and Discrete States

To use a signal object to specify an initial value:

1 Create the signal object in the MATLAB workspace, as explained in "Working with Data Objects" on page 7-12.

   The name of the signal object must be the same as the name of the signal or discrete state that the object is initializing.

Note Consider also setting the **Signal name must resolve to Simulink signal object** option in the Signal Properties dialog box. This setting ensures consistency between signal objects in the MATLAB workspace and the signals that appear in your model.

2 Set the signal object's storage class to a value other than `'auto'` or `'SimulinkGlobal'`.

3 Set the signal object's `Initial value` property to the initial value of the signal or state. For details on what you can specify, see the description of `Simulink.Signal` in the Simulink online reference.

   If you can also use a block parameter to set the initial value of the signal or state, you should set the parameter either to null (`[]`) or to the same value as the initial value of the signal object. If you set the parameter value to null, Simulink uses the value specified by the signal object to initialize the signal or state. If you set the parameter to any other value, Simulink compares the parameter value to the signal object value and displays an error if they differ.

Using Signal Objects to Tune Initial Values

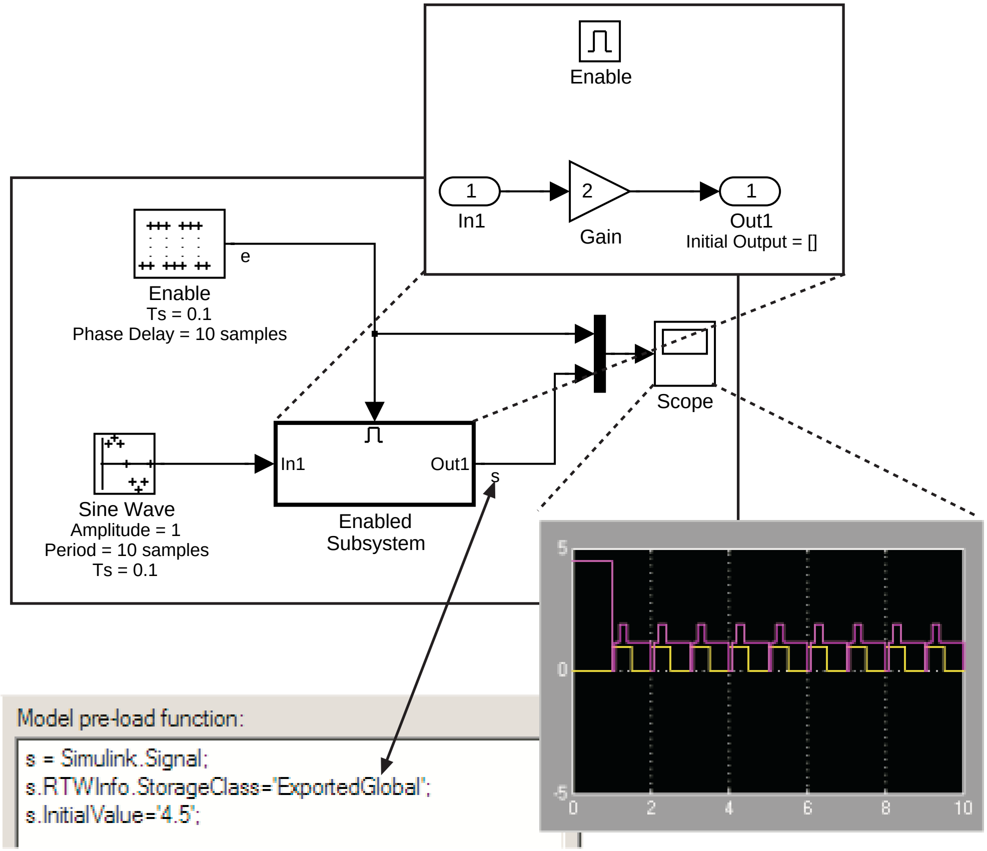
Simulink allows you to use signal objects as an alternative to parameter objects (see `Simulink.Parameter` class in the Simulink online reference) to tune the initial values of block outputs and states that can be specified via a tunable parameter. To use a signal object to tune an initial value, create a signal object with the same name as the signal or state and set the signal object's initial value to an expression that includes a variable defined in the MATLAB workspace. You can then tune the initial value by changing the value of the corresponding workspace variable during the simulation.

For example, suppose you want to tune the initial value of a Memory block state named `M1`. To do this, you might create a signal object named `M1`, set its storage class to `'ExportedGlobal'`, set its initial value to `K` (`M1.InitialValue='K'`), where `K` is a workspace variable in the MATLAB workspace, and set the corresponding initial condition parameter of the Memory block to `[]` to avoid consistency errors. You could then change the initial value of the Memory block's state any time during the simulation by changing the value of K at the MATLAB command line and updating the block diagram (e.g., by typing **Ctrl+D**).

Note To be tunable via a signal object, a signal or state's corresponding initial condition parameter must be tunable, e.g., the inline parameter optimization for the model containing the signal or state must be off or the parameter must be declared tunable in the Model Parameter Configuration dialog box. For more information, see "Tunable Parameters" on page 1-9 and "Changing the Values of Block Parameters During Simulation" on page 4-12.

Example: Using a Signal Object to Initialize a Subsystem Output

The following example shows a signal object specifying the initial output of an enabled subsystem.



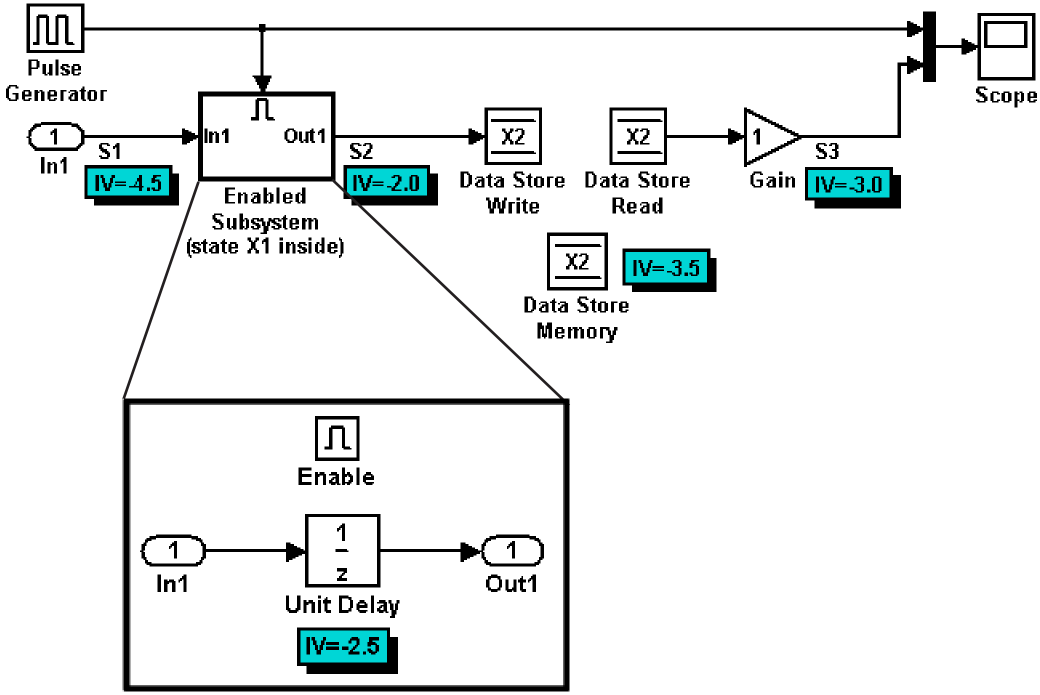
```
s = Simulink.Signal;
s.RTWInfo.StorageClass='ExportedGlobal';
s.InitialValue='4.5';
```

Signal s is initialized to 4.5. To avoid a consistency error, the initial value of the enabled subsystem's Outport block must be [] or 4.5.

If you need a signal object and its initial value setting to persist across Simulink sessions, see "Creating Persistent Data Objects" on page 7-22.

Initialization Behavior Summary for Signal Objects

The following model and table show different types of signals and discrete states that you can initialize and the simulation behavior that results for each.



Signal or Discrete State	Description	Behavior
S1	Root inport	• Initialized to `S1.InitialValue`. • If you use the **Data Import/Export** pane of the Configuration Parameters dialog box to load an input value from the model's base workspace, the value is set, and may differ, at each time step. Otherwise, the value remains constant.
X1	Unit Delay block — Block with a discrete state that has an initial condition	• Initialized to `X1.InitialValue`. • Simulink checks whether `X1.InitialValue` matches the initial condition specified for the block and displays an error if a mismatch occurs. • At first write, the output equals `X1.InitialValue` and the state equals `S1`. • At each time step after the first write, the output equals the state and the state is updated to equal `S1`. • If the block is inside an enabled subsystem, you can use the initial value as a reset value if the subsystem's Enable block parameter **States when enabling** is set to `reset`.
X2	Data Store Memory block	• Data type work (Dwork) vector initialized to `X2.InitialValue`. For information on work vectors, see "Work Vectors" in Writing S-Functions. • Simulink checks whether `X2.InitialValue` matches the initial condition specified for the block, and displays an error if a mismatch occurs. • Data Store Write blocks overwrite the value.

Signal or Discrete State	Description	Behavior
S2	Output of an enabled subsystem	• Initialized to `S2.InitialValue` or the value of the Outport block. If multiple initial values are specified for the same signal, all initial values must be the same. • The first write occurs when the subsystem is enabled. The block feeding the subsystem output sets the value. • The initial value is also used as a reset value if the subsystem's Enable block parameter **States when enabling** or Outport block parameter **Output when disabled** is set to `reset`.
S3	Persistent signals	• Initialized to `S3.InitialValue`. • The output value is reset by the block at each time step. • Affects code generation only. For simulation, setting the initial value for `S3` is irrelevant because the values are overwritten at the model's simulation start time.

Working with Test Points

A *test point* is a signal that Simulink guarantees to be observable when using a Floating Scope block in a model. Simulink allows you to designate any signal in a model as a test point. Designating a signal as a test point exempts the signal from model optimizations, such as signal storage reuse (see "Signal storage reuse" on page 11-92) and block reduction (see "Implement logic signals as boolean data (vs. double)" on page 11-92), that can render signals inaccessible and hence unobservable during simulation.

Otherwise, this feature is primarily intended for use when generating code from a model with Real-Time Workshop. For more information about test points in the context of code generation, see "Declaring Test Points" in the Real-Time Workshop documentation.

- "Designating a Signal as a Test Point" on page 5-56
- "Displaying Test Point Indicators" on page 5-57

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 5-42).

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.

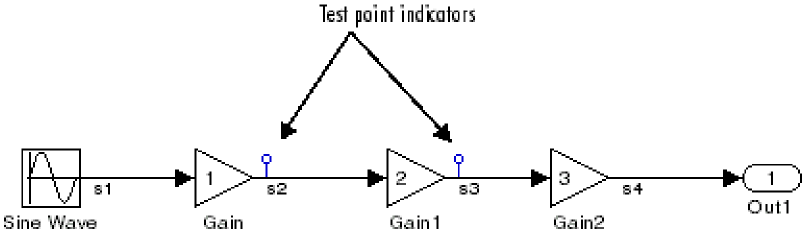
If all these conditions are true, you can designate the signal as a test point by setting the associated object's storage class property to any value but auto.

Displaying Test Point Indicators

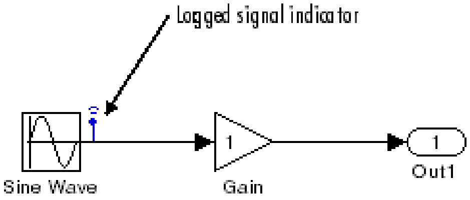
By default, Simulink displays an indicator next to each signal that serves as a test point.

Note Simulink displays indicators only for signals whose **Test point** option is checked in the **Signal Properties** dialog box. If you designate a test point using a Simulink.Signal object, Simulink does not display an indicator.

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. Some of the options display properties of composite signals. See Chapter 6, "Using Composite Signals" for details.

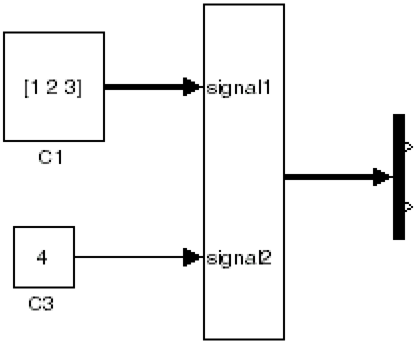
For more information, see

- "Display Options" on page 5-59
- "Signal Names" on page 5-61
- "Signal Labels" on page 5-61
- "Displaying Signals Represented by Virtual Signals" on page 5-62

Display Options

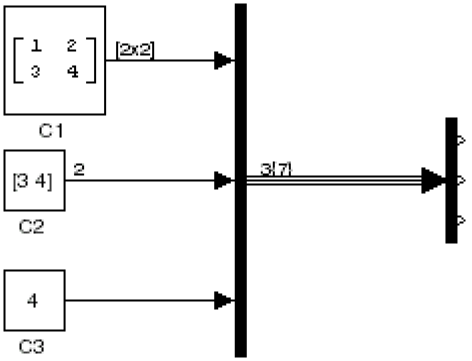
Wide nonscalar lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.



Signal dimensions

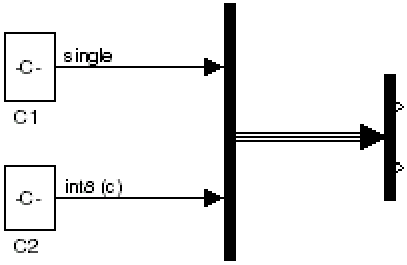
Display the dimensions of nonscalar signals next to the line that carries the signal.



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`$_1$`xN`$_2$`]` where `N`$_i$ is the size of the `i`th dimension of the signal. If the line represents a bus carrying signals of the same data type, Simulink displays `N{M}` where `N` is the number of signals carried by the bus and `M` is the total number of signal elements carried by the bus. If the bus carries signals of different data types, Simulink displays only the total number of signal elements `{M}`.

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 5-42)
- 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" in the online Simulink documentation).

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's Assertion blocks and Simulink Verification and Validation's Model Coverage Tool. For a description of the Model Coverage Tool, see the "Simulink Verification and Validation User's Guide" on the MathWorks Web site (`www.mathworks.com`).

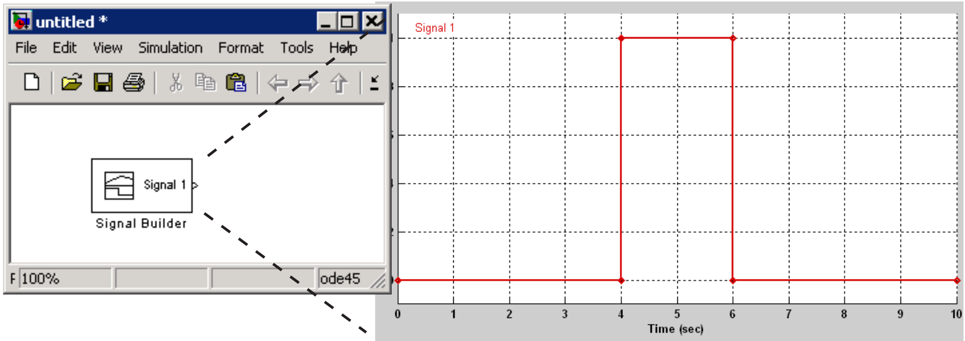
For more information, see

- "Creating a Signal Group Set" on page 5-63
- "Signal Builder Dialog Box" on page 5-64
- "Editing Signal Groups" on page 5-67
- "Editing Signals" on page 5-67
- "Editing Waveforms" on page 5-70
- "Signal Builder Time Range" on page 5-76
- "Exporting Signal Group Data" on page 5-77
- "Printing, Exporting, and Copying Waveforms" on page 5-77
- "Simulating with Signal Groups" on page 5-78
- "Simulation Options Dialog Box" on page 5-79

Creating a Signal Group Set

To create an interchangeable set of signal groups:

1 Drag an instance of the Signal Builder block from the Simulink Sources library and drop it into your model.



By default, the block represents a single signal group containing a single signal source that outputs a square wave pulse.

2 Use the block's signal editor (see "Signal Builder Dialog Box" on page 5-64) to create additional signal groups, add signals to the signal groups, modify existing signals and signal groups, and select the signal group that the block outputs.

Note Each signal group must contain the same number of signals.

3 Connect the output of the block to your diagram.

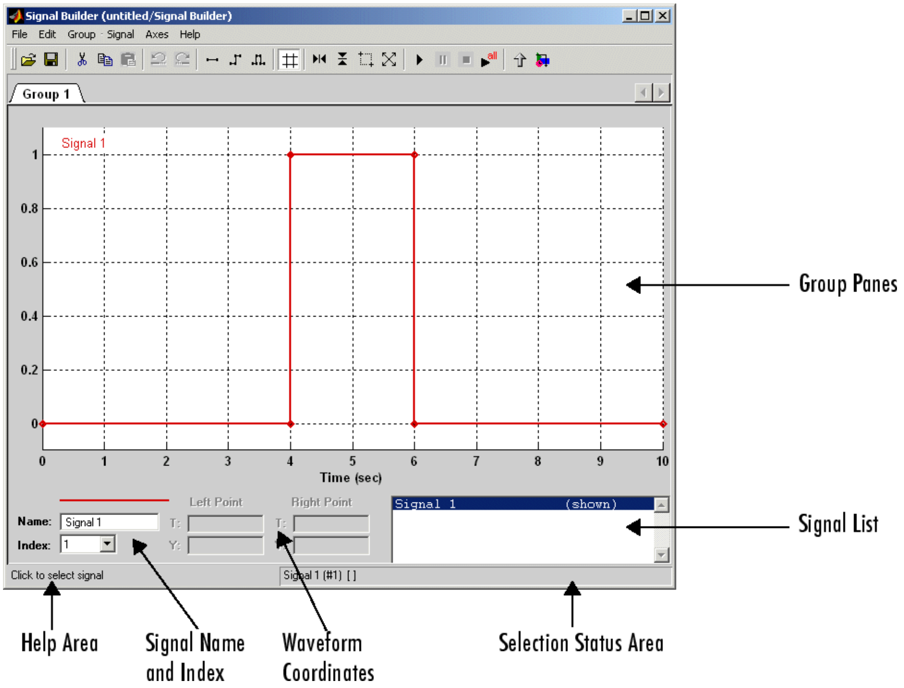
The block displays an output port for each signal that the block can output.

You can create as many Signal Builder blocks as you like in a model, each representing a distinct set of interchangeable groups of signal sources. See "Simulating with Signal Groups" on page 5-78 for information on using signal groups in a model.

Signal Builder Dialog Box

The Signal Builder block's dialog box allows you to define the 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 5-76). 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 5-67) 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 5-70).

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 5-69).

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 5-69).

Help Area

Displays context-sensitive tips on using **Signal Builder** dialog box features.

Editing Signal Groups

The Signal Builder dialog box allows you to create, rename, move, and delete signal groups from the set of groups represented by a Signal Builder block.

Creating and Deleting Signal Groups

To create a signal group, you must copy an existing signal group and then modify it to suit your needs. To copy an existing signal group, select its tab and then select **Copy** from the Signal Builder's **Group** menu. To delete a group, select its tab and then select **Delete** from the **Group** menu.

Renaming Signal Groups

To rename a signal group, select the group's tab and then select **Rename** from the Signal Builder's **Group** menu. A dialog box appears. Edit the existing name in the dialog box or enter a new name. Click **OK**.

Moving Signal Groups

To reposition a group in the stack of group panes, select the pane and then select **Move Right** from the Signal Builder's **Group** menu to move the group lower in the stack or **Move Left** to move the pane higher in the stack.

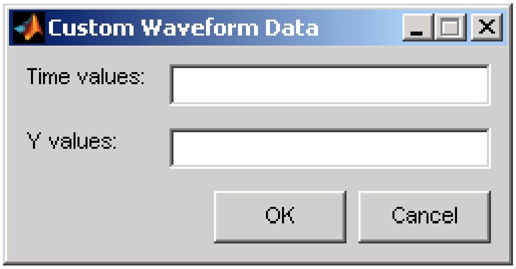
Editing Signals

The **Signal Builder** dialog box allows you to create, cut and paste, hide, and delete signals from signal groups.

Creating Signals

To create a signal in the currently selected signal group, select **New** from the Signal Builder's **Signal** menu. A menu of waveforms appears. The menu includes a set of standard waveforms (**Constant**, **Step**, etc.) and a **Custom** waveform option. Select one of the waveforms. If you select a standard waveform, the Signal Builder adds a signal having that waveform to the currently selected group.

If you select **Custom**, a custom waveform dialog box appears.



The dialog box allows you to specify a custom piecewise linear waveform to be added to the groups defined by the Signal Builder block. Enter the custom waveform's time coordinates in the **Time values** field and the corresponding signal amplitudes in the **Y values** field. The entries in either field can be any MATLAB expression that evaluates to a vector. The resulting vectors must be of equal length. Click **OK**. The Signal Builder adds a signal having the specified waveform to the currently selected group.

Copying and Pasting Signals

To copy a signal from one group and paste it into another group as a new signal:

1. Select the signal you want to copy.
2. Select **Copy** from the Signal Builder's **Edit** menu or click the corresponding button from the toolbar.
3. Select the group into which you want to paste the signal.
4. Select **Paste** from the Signal Builder's **Edit** menu or click the corresponding button on the toolbar.

To copy a signal from one axes and paste it into another axes to replace its signal:

1. Select the signal you want to copy.

2. Select **Copy** from the Signal Builder's **Edit** menu or click the corresponding button from the toolbar.
3. Select the signal on the axes that you want to replace.
4. Select **Paste** from the Signal Builder's **Edit** menu or click the corresponding button on the toolbar.

Deleting Signals

To delete a signal, select the signal and choose **Delete** or **Cut** from the Signal Builder's **Edit** menu. As a result, Simulink deletes the signal from the current group. Since each signal group must contain the same number of signals, Simulink also deletes all signals sharing the same index in the other groups.

Renaming a Signal

To rename a signal, select the signal and choose **Rename** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field that displays the signal's current name. Edit or replace the current name with a new name. Click **OK**. Or edit the signal's name in the **Name** field in the lower-left corner of the **Signal Builder** dialog box.

Changing a Signal's Index

To change a signal's index, select the signal and choose **Change Index** from the Signal Builder's **Signal** menu. A dialog box appears with an edit field containing the signal's existing index. Edit the field and select **OK**. Or select an index from the **Index** list in the lower-left corner of the Signal Builder window.

Hiding Signals

By default, the **Signal Builder** dialog box displays the 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 5-66).

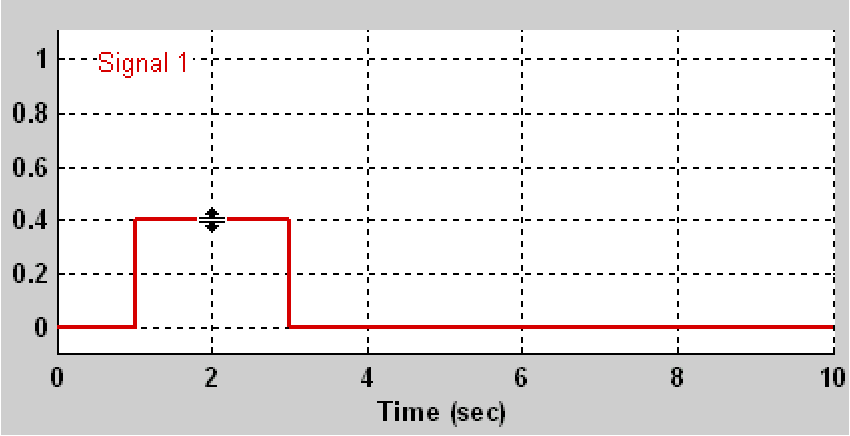
Editing Waveforms

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.

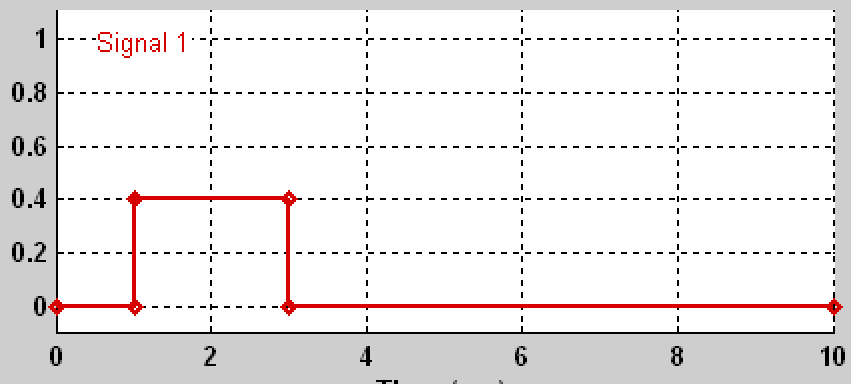
Reshaping a Waveform

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.

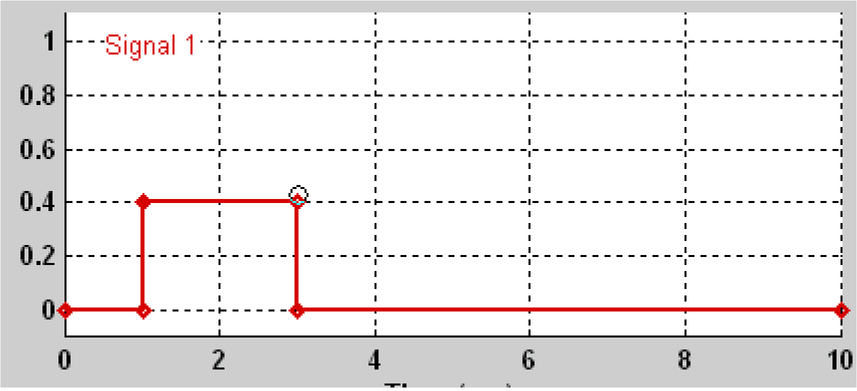


The Signal Builder displays the waveform's points to indicate that the waveform is selected.

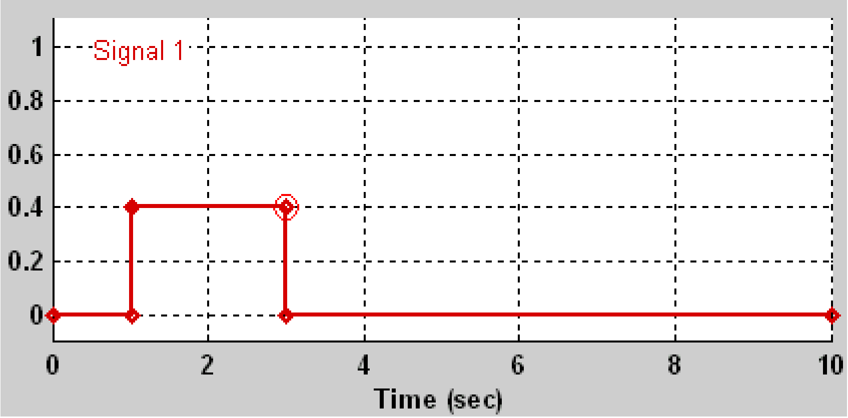


To deselect a waveform, left-click any point on the waveform graph that is not on the waveform itself or press the **Esc** key.

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.

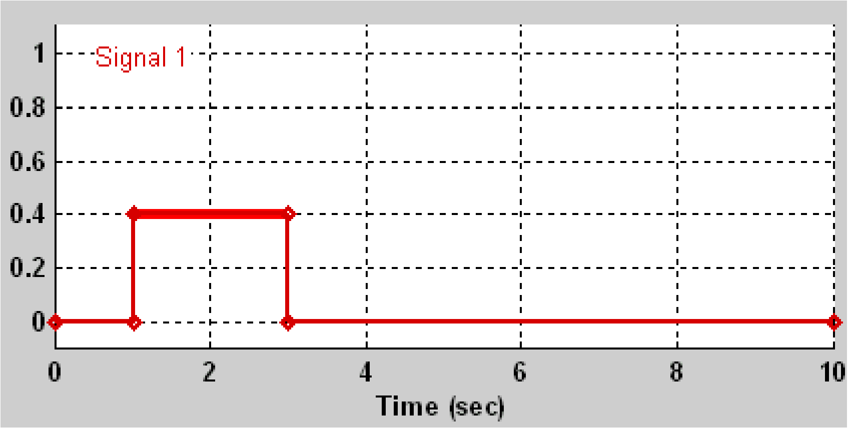


Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate that it is selected.



To deselect the point, press the **Esc** key.

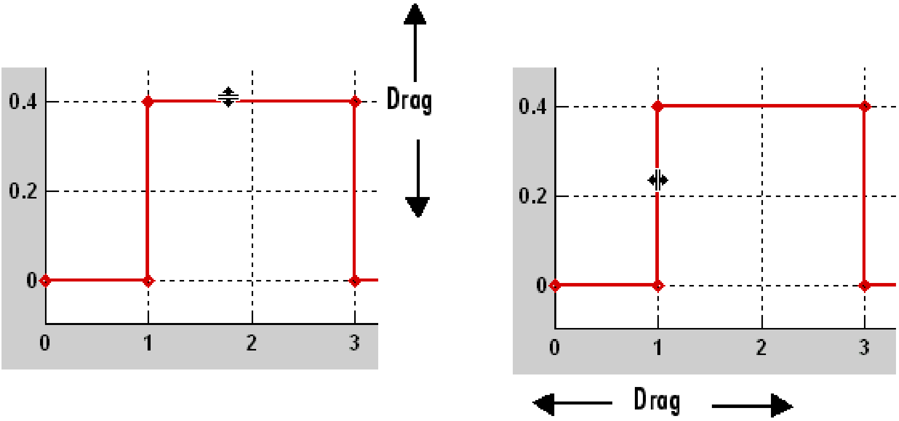
Selecting Segments. To select a line segment, first select the waveform that contains it. Then left-click the segment. The Signal Builder thickens the segment to indicate that it is selected.



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 5-74) or by `0.1` inches if the snap grid is not enabled.

Dragging Segments. To drag a line segment to a new location, position the mouse cursor over the line segment. The mouse cursor changes shape to show the direction in which you can drag the segment.



Press the left mouse button and drag the segment in the direction indicated to the desired location. You can also use the arrow keys on your keyboard to move the selected line segment.

Dragging points. To drag a point along the signal amplitude (vertical) axis, move the mouse cursor over the point. The cursor changes shape to a circle to indicate that you can drag the point. Drag the point parallel to the *y*-axis to the desired location. To drag the point along the time (horizontal) axis, press the **Shift** key while dragging the point. You can also use the arrow keys on your keyboard to move the selected point.

Snap Grid. Each waveform axis contains an invisible snap grid that facilitates precise positioning of waveform points. The origin of the snap grid coincides with the origin of the waveform axis. When you drop a point or segment that you have been dragging, the Signal Builder moves the point or the segment's points to the nearest point or points on the grid, respectively. The Signal Builder's **Axes** menu allows you to specify the grid's horizontal (time) axis and vertical (amplitude) axis spacing independently. The finer the spacing, the more freedom you have in placing points but the harder it is to position points precisely. By default, the grid spacing is 0, which means that you can place points anywhere on the grid; i.e., the grid is effectively off. Use the **Axes** menu to select the spacing that you prefer.

Inserting and Deleting points. To insert a point, first select the waveform. Then hold down the **Shift** key and left-click the waveform at the point where you want to insert the point. To delete a point, select the point and press the **Del** key.

Editing Point Coordinates. To change the coordinates of a point, first select the point. The Signal Builder displays the current coordinates of the point in the **Left Point** edit fields at the bottom of the **Signal Builder** dialog box. To change the amplitude of the selected point, edit or replace the value in the **Y** field with the new value and press **Enter**. The Signal Builder moves the point to its new location. Similarly edit the value in the **T** field to change the time of the selected point.

Editing Segment Coordinates. To change the coordinates of a segment, first select the segment. The Signal Builder displays the current coordinates of the endpoints of the segment in the **Left Point** and **Right Point** edit fields at the bottom of the **Signal Builder** dialog box. To change a coordinate, edit the value in its corresponding edit field and press **Enter**.

Changing the Color of a Waveform

To change the color of a 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**.

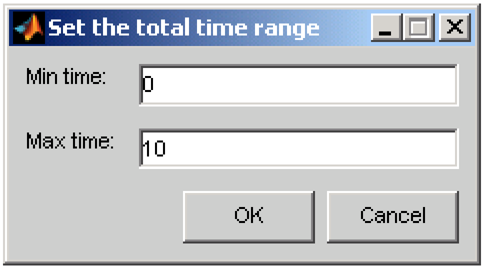
Signal Builder Time Range

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 5-76).

If the simulation starts before the start time of a block's time range, the block extrapolates its initial output from its first two defined outputs. If the simulation runs beyond the block's time range, the block by default outputs values extrapolated from the last defined signal values for the remainder of the simulation. The Signal Builder's **Simulation Options** dialog box allows you to specify other final output options (see "Signal values after final time" on page 5-79 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

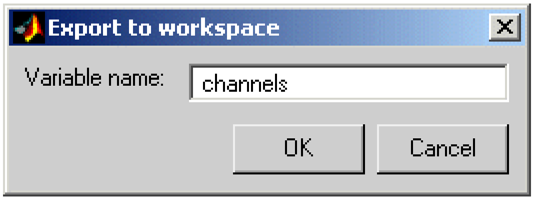
Min time: 0

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.



The Signal Builder exports the data by default to a workspace variable named `channels`. To export to a differently named variable, enter the variable's name in the **Variable name** field. Click **OK**. The Signal Builder exports the data to the workspace as the value of the specified variable.

The exported data is an array of structures. The structure's `xData` and `yData` fields contain the coordinate points defining signals in the currently selected signal group. You can access the coordinate values defining signals associated with other signal groups from the structure's `allXData` and `allYData` fields.

Printing, Exporting, and Copying Waveforms

The **Signal Builder** dialog box allows you to print, export, and copy the waveforms visible in the active signal group.

To print the waveforms to a printer, select **Print** from the block's **File** menu.

You can also export the waveforms to other destinations by using the **Export** option from the block's **File** menu. From this submenu, select one of the following destinations:

- **To File** — Converts the current view to a graphics file.

  Select the format of the graphics file from the **Save as type** drop-down list on the resulting **Export** dialog box.

- **To Figure** — Converts the current view to a MATLAB figure window.

To copy the waveforms to the system clipboard for pasting into other applications, select **Copy Figure To Clipboard** from the block's **Edit** menu.

Simulating with Signal Groups

You can use standard simulation commands to run models containing Signal Builder blocks or you can use the Signal Builder's **Run all** command (see "Running All Signal Groups" on page 5-78).

Activating a Signal Group

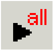
During a simulation, a Signal Builder block always outputs the active signal group. The active signal group is the group selected in the **Signal Builder** dialog box for that block, if the dialog box is open, otherwise the group that was selected when the dialog box was last closed. To activate a group, open the group's **Signal Builder** dialog box and select the group.

Running Different Signal Groups in Succession

The Signal Builder's toolbar includes the standard Simulink buttons for running a simulation. This facilitates running several different signal groups in succession. For example, you can open the dialog box, select a group, run a simulation, select another group, run a simulation, etc., all from the Signal Builder's dialog box.

Running All Signal Groups

To run all the signal groups defined by a Signal Builder block, open the block's dialog box and click the **Run all** button

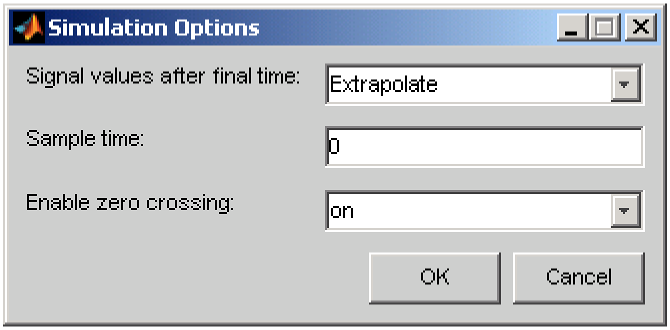


from the Signal Builder's toolbar. The **Run all** command runs a series of simulations, one for each signal group defined by the block. If you installed Simulink Verification and Validation on your system and are using the Model Coverage Tool, the **Run all** command configures the tool to collect and save coverage data for each simulation in the MATLAB workspace and display a report of the combined coverage results at the end of the last simulation. This allows you to quickly determine how well a set of signal groups tests your model.

Note To stop a series of simulations started by the **Run all** command, enter **Ctrl+C** at the MATLAB command line.

Simulation Options Dialog Box

The **Simulation Options** dialog box allows you to specify simulation options pertaining to the Signal Builder. To display the dialog box, select **Simulation Options** from the Signal Builder's **File** menu. The dialog box appears.



Simulation Options

Signal values after final time: Extrapolate

Sample time: 0

Enable zero crossing: on

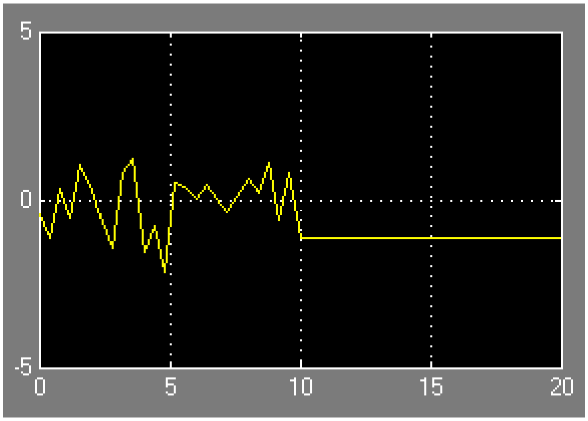
The dialog box allows you to specify the following options.

Signal values after final time

The setting of this control determines the output of the Signal Builder block if a simulation runs longer than the period defined by the block. The options are

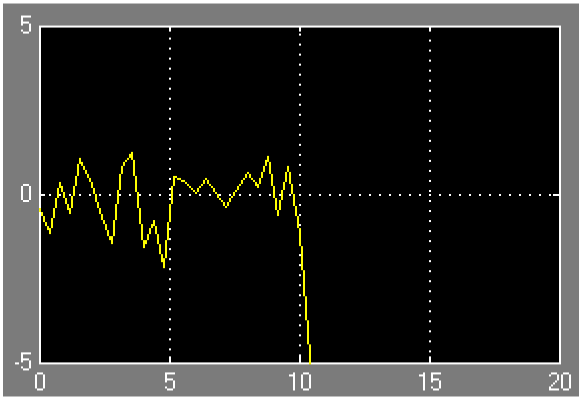
- `Hold final value`

Selecting this option causes the Signal Builder block to output the last defined value of each signal in the currently active group for the remainder of the simulation.



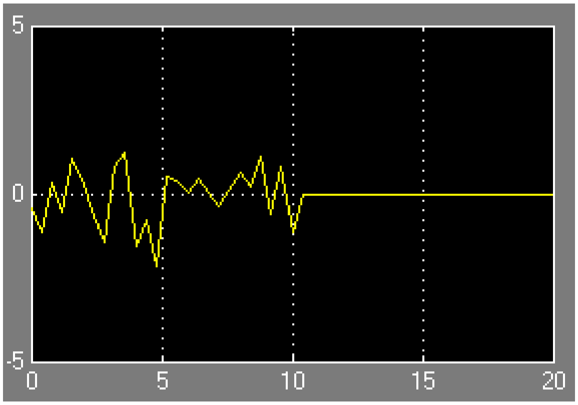
- `Extrapolate`

  Selecting this option causes the Signal Builder block to output values extrapolated from the last defined value of each signal in the currently active group for the remainder of the simulation.



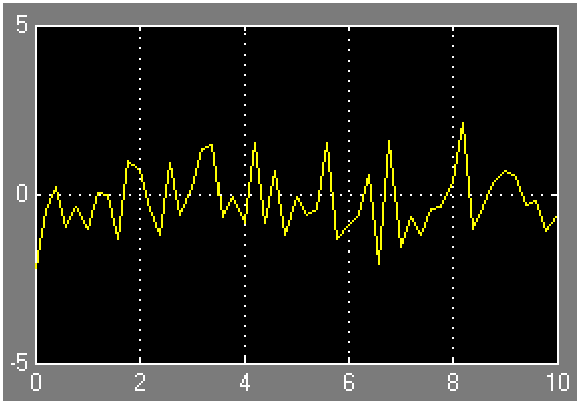
- `Set to zero`

Selecting this option causes the Signal Builder block to output zero for the remainder of the simulation.

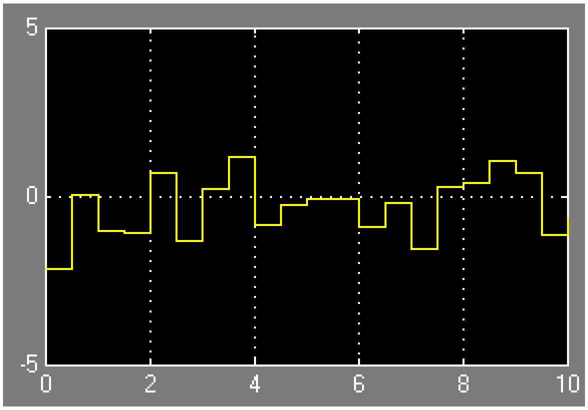


Sample time

Determines whether the Signal Builder block outputs a continuous (the default) or a discrete signal. If you want the block to output a continuous signal, enter 0 in this field. For example, the following display shows the output of a Signal Builder block set to output a continuous Gaussian waveform over a period of 10 seconds.



If you want the block to output a discrete signal, enter the sample time of the signal in this field. The following example shows the output of a Signal Builder block set to emit a discrete Gaussian waveform having a `0.5` second sample time.



Enable zero crossing

Specifies whether the Signal Builder block detects zero-crossing events (enabled by default). For more information, see "Zero-Crossing Detection" on page 1-20.

Using Composite Signals

About Composite Signals

A *composite signal* is a signal that is composed of other signals. It is analogous to a bundle of wires held together by tie wraps. Composite signals have only one purpose: to reduce visual complexity by grouping signals that run in parallel over some or all of their course. For information about individual signals, as distinct from the composite signals described in this chapter, see "Signal Basics" on page 5-3.

Simulink provides two types of composite signals: muxes and buses. Muxes are simpler and easier to use, but provide only a limited subset of the capabilities of buses. The two types perform similarly where their capabilities overlap, but they differ architecturally. Muxes are the older type, and are supported for compatibility with existing applications. In general, new applications should use buses. Existing applications that use muxes can be left unchanged or converted to use buses.

Many composite signals are virtual: they exist only graphically, and have no effect on simulation or generated code. All muxes are virtual, but a bus can be virtual or nonvirtual. Nonvirtual buses usually do not affect the results of simulation. They appear as structures in generated code and can affect generated code performance. See "Virtual and Nonvirtual Buses" on page 6-15 for details.

In some cases, muxes and buses can be intermixed; implicit conversion occurs when needed. However, The MathWorks discourages intermixing muxes and buses. The practice may become unsupported at some future time, and should not be used in new applications. Simulink provides diagnostics that report cases where muxes are mixed with buses, and includes capabilities that you can use to upgrade a model to eliminate such mixtures. This chapter describes these and other composite signal capabilities.

Using Muxes

A *mux* is an indexed vector that implements a composite signal. In Simulink documentation, indexed vectors are also called “vectors” and “wide signals”, and both “mux” and “vector” appear in Simulink GUI labels and API names. The underlying software construct, an indexed vector, is always the same; the difference is the context in which the construct appears.

Mux signals are virtual signals, and the blocks that manipulate muxes are virtual blocks. All signals in a mux must have the same attributes. The Signal Routing library provides two blocks that you can use for implementing muxes:

Mux
Combine several input signals into a mux signal

Demux
Extract and output the elements of a mux signal

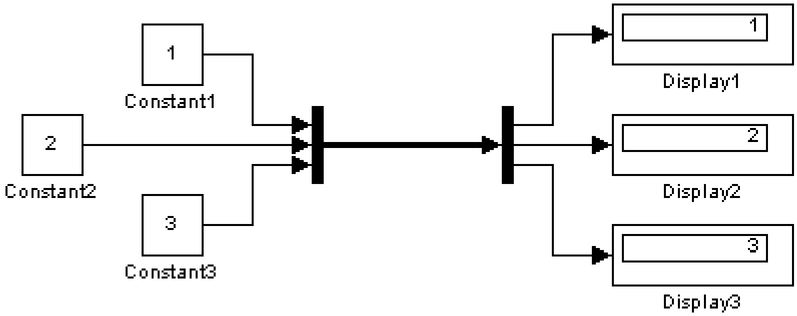
To create a mux signal:

1. Clone a Mux and Demux block from the Signal Routing library.

2. Set the Mux blocks **Number of inputs** and the Demux block’s **Number of outputs** properties to the desired values.

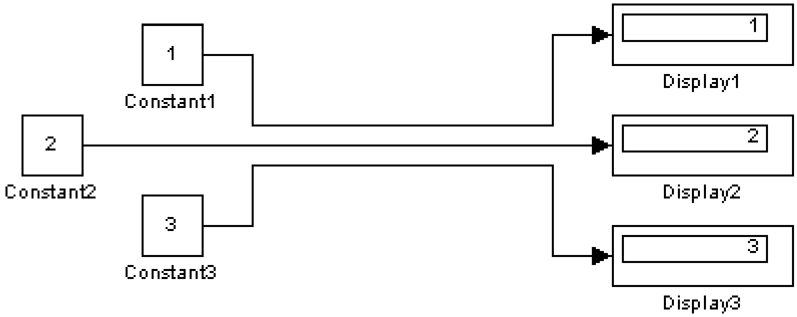
3. Connect the Mux, Demux, and other blocks as needed to implement the desired composite signal.

The next figure shows three signals that are input to a Mux block, transmitted as a mux signal to a Demux block, and output as separate signals.



The Mux and Demux blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the mux signal, is wide because the model has been built with **Format > Port/Signal Displays > Wide Nonscalar Lines** enabled in the model menu. See "Displaying Signal Properties" on page 5-59 for details.

Because the Mux and Demux blocks are virtual blocks, and a mux signal is a virtual signal, they have no effect on simulation or code generation. Thus the simulation results and generated code for the above model would be exactly the same if the Mux blocks were eliminated and the mux signal replaced by three nonvirtual signals:



Signals input to a Mux block can be virtual composite signals, but the resulting mux is flat, not hierarchical. The signals in the mux appear in the order in which they were input to the Mux block. Use a bus rather than a mux if you want to nest composite signals. See “Using Buses” on page 6-5 for details.

Signals input to a Mux block should all have the same attributes. If they do not, the block will output a bus rather than a mux, unless you have configured the model to disable this practice. The MathWorks discourages using Mux blocks to create buses. See “Intermixing Composite Signal Types” on page 6-21 for details.

If a Demux block has more outputs than the number of signals in the input mux, an error occurs. A Demux block can have fewer outputs than the number of signals in the input mux. See the Demux block documentation for details.

A Demux block can input a bus unless you have configured the model to disable this practice. The MathWorks discourages using Demux blocks to access buses. See “Intermixing Composite Signal Types” on page 6-21 for details.

Using Buses

A bus is a composite signal that is implemented as a hierarchical structure. The components of a bus can have different attributes and can themselves be composite signals (buses or muxes). Bus signals can be virtual or nonvirtual; see “Virtual Signals” on page 5-10 for details. The Signal Routing library provides three blocks that you can use for implementing buses:

Bus Creator
Create a signal bus. This block is virtual or nonvirtual to match the bus type.

Bus Selector
Select signals from a bus. This block is always virtual.

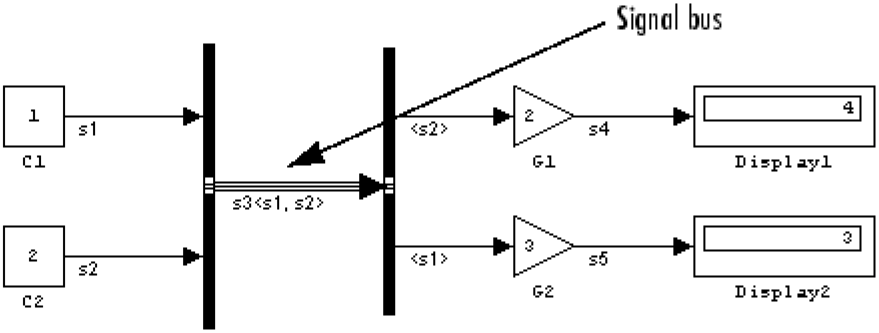
Bus Assignment
Assign values to specified bus elements. This block is always virtual.

To create a bus signal with default properties:

1 Clone a Bus Creator and Bus Selector block from the Signal Routing library.

2 Connect the Bus Creator, Bus Selector, and other blocks as needed to implement the desired composite signal.

The next figure shows two signals that are input to a Bus Creator block, transmitted as a bus signal to a Bus Selector block, and output as separate signals.

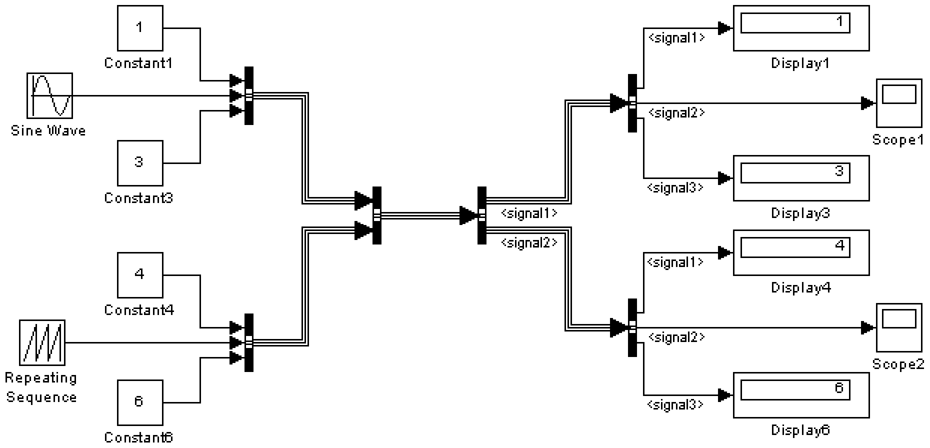


The Bus Creator and Bus Selector blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the bus signal, is tripled because the model has been built, and the middle line is solid because the bus is virtual. See “Signal Line Styles” on page 5-4 for details. You can also display other bus signal characteristics graphically, as described under “Displaying Signal Properties” on page 5-59.

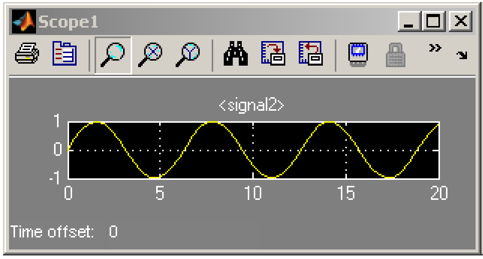
For more information, see the reference documentation for the Bus Creator, Bus Selector, and Bus Assignment blocks.

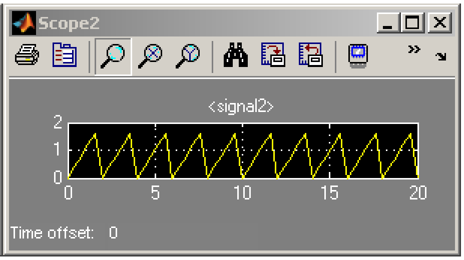
Nesting Buses

Buses can be nested to any depth. Simulink automatically handles most of the complexities involved. For example, this example shows six signals nested into two buses, which are nested into one, followed by separation into two buses and then into six separate signals:



The six signals retain their separate identities just as if no bus creation and selection occurred, as shown by the Display and Scope blocks.





Specifying nonvirtual buses, like those in the previous figure, requires only cloning blocks, setting parameters, and connecting signals. Bus Creator and Bus Selector blocks have two ports by default. See the Bus Creator and Bus Selector block documentation for information about how to specify buses of different widths.

Note Nested buses must all be either virtual or nonvirtual: a bus cannot contain a mixture of virtual and nonvirtual nested buses.

Using Bus Objects

- "Creating Bus Objects with the Bus Editor" on page 6-8
- "Creating and Saving Bus Objects with the API" on page 6-12
- "Associating Bus Objects with Model Entities" on page 6-12

The properties specified for bus blocks and bus signals in their Parameters dialogs are adequate for defining virtual buses and performing limited error checking. To define a nonvirtual bus or provide complete bus signal error checking, you must supply additional information by providing a *bus object*. Referenced models, function-call triggered subsystems, and all other nonvirtual systems that can input and output buses require nonvirtual buses and hence need bus objects. See "Virtual and Nonvirtual Buses" on page 6-15 for more information.

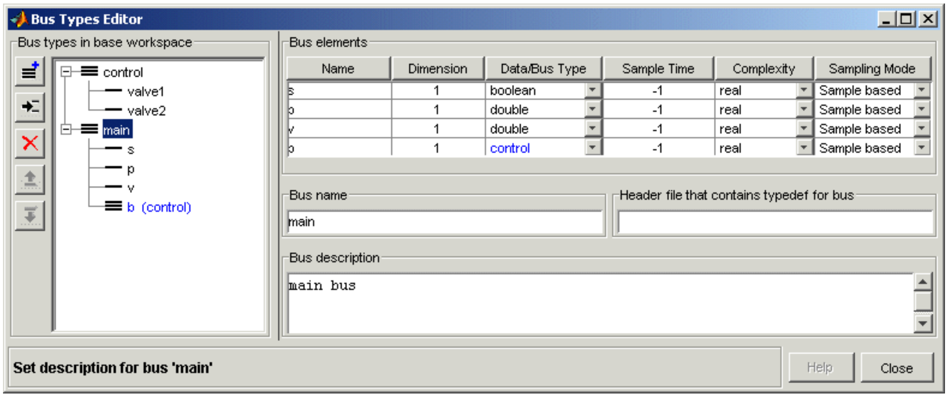
A bus object is an instance of the class `Simulink.Bus` that completely specifies the properties of a bus signal. A bus object can exist in the base workspace, where it is available to all models, or on an individual model workspace, where it is available only to that model. Any bus object that is used by more than one model, as in model referencing, must be defined in the base workspace.

Creating Bus Objects with the Bus Editor

The Simulink Bus Editor allows you to create a new bus object or change the properties of an existing bus object. You can open the Bus Editor in any of these ways:

- Select **Bus Editor** from the model editor's **Tools** menu.
- Click the **Launch Bus Editor** button on a bus object's dialog box in the Model Explorer.
- Enter `buseditor` at the MATLAB command line.

After you have performed any of these actions, the Bus Types Editor appears:



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

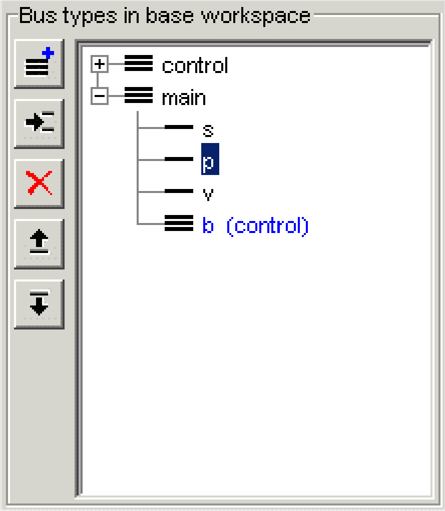
Set description for bus 'main'

You can use the Bus Types Editor to specify the following properties. Each has its own section in the dialog.

- "Bus types in base workspace" on page 6-9
- "Bus elements" on page 6-11
- "Bus name" on page 6-11
- "Header file" on page 6-11
- "Bus description" on page 6-11

Bus types in base workspace

This section 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 base 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. 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. Selecting any element displays the element's properties in the Bus Types Editor's **Bus elements** table.

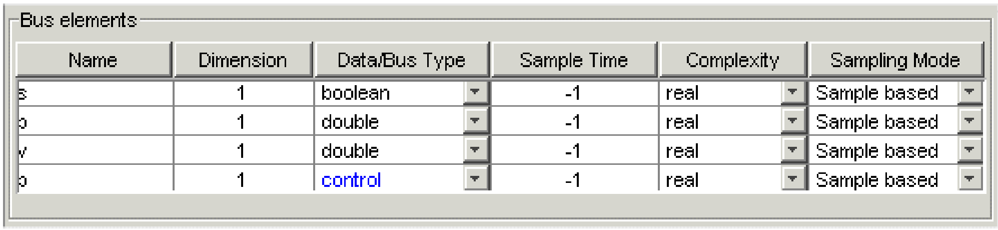
Editing Buttons. This group of buttons allows you to create and modify bus objects in the base workspace. It includes the following buttons.

Command	Icon	Description
Create	☰+	Create a bus object in the base workspace.
Insert	➔☰	Insert an element in the bus object selected in the Bus Editor's bus object hierarchy pane.

Command	Icon	Description
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 section 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
o	1	double	-1	real	Sample based
v	1	double	-1	real	Sample based
o	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.

Creating and Saving Bus Objects with the API

A bus object is an instance of `Simulink.Bus`, and the elements that comprise it are instances of `Simulink.BusElement`. Simulink also provides the following functions that you can use with bus objects:

`Simulink.Bus.cellToObject`
: Convert a cell array containing bus information to bus objects in the base workspace

`Simulink.Bus.createObject`
: Create bus objects for blocks, optionally saving them in an M-file in a specified format

`Simulink.Bus.save`
: Save specified buses or all buses from the base workspace to an M-file in a specified format

When you use `Simulink.SubSystem.convertToModelReference` to convert an atomic subsystem to a referenced model, you can save any bus objects created during the conversion to an M-file.

For more information about the classes and functions mentioned, see the Simulink reference documentation.

Associating Bus Objects with Model Entities

You can associate a bus object with any bus signal, a Bus Creator block, or an Inport or Outport block.

Associating Bus Objects with Bus Signals

To associate a bus object with a bus signal:

1. Select the signal and choose **Edit > Signal Properties**, or choose **Signal Properties** from the signal's context menu.

2. Give the signal the same name as the bus signal you want to use. Multiple signals can use the same name.

3. Check **Signal name must resolve to Simulink data object**.

4. Click **OK** or **Apply**.

See “Signal Properties Dialog Box” on page 5-42 for more information.

Associating Bus Objects with Bus Creator Blocks

To associate a bus object with a Bus Creator block:

1. Open the Block Parameters dialog.
2. Select **Specify properties via bus object**.
3. Enter the bus object name in the **Bus object** field.
4. Click **OK** or **Apply**.

See the Bus Creator block documentation for more information.

Associating Bus Objects with Inport and Outport Blocks

Inport and Outport blocks use bus objects to provide the specifications necessary for a bus signal to cross the boundary between nonvirtual subsystems. See “Virtual and Nonvirtual Buses” on page 6-15 for more information.

To associate a bus object with an Inport or Outport block:

1. Open the Block Parameters dialog.
2. Select the **Signal Specification** tab.
3. Select **Specify properties via bus object**.
4. Enter the bus object name in the **Bus object for validating input bus** field.
5. Click **OK** or **Apply**.

See the Inport and Outport block documentation for more information.

Checking for Bus Errors

Simulink uses bus block and bus signal parameter values to check for bus errors insofar as possible, but these values do not support completely rigorous checking. You can specify complete bus error checking by associating a bus object with any bus signal, Bus Creator block, or Inport block, as described in "Associating Bus Objects with Model Entities" on page 6-12.

When a bus object is associated with a model entity, any deviation from the bus properties specified in the object causes an error. The requirements specified by a bus object apply only to the entity with which the object is directly associated; they do not propagate to other entities in the model. To apply the requirements to additional entities, explicitly associate the applicable bus object with each one.

Virtual and Nonvirtual Buses

- "Specifying Nonvirtual Buses" on page 6-16
- "Bus-Capable Blocks" on page 6-16

A bus signal can be *virtual*, meaning that it is just a graphical convenience that has no functional effect, or *nonvirtual*, meaning that the signal occupies its own storage. During simulation, a block connected to a virtual bus reads inputs and writes outputs by accessing the memory allocated to the component signals. These signals are typically noncontiguous, and no intermediate memory exists. Simulation results and generated code are exactly the same as if the bus did not exist, which functionally it does not.

By contrast, a block connected to a nonvirtual bus reads inputs and writes outputs by accessing copies of the component signals. Simulink maintains the copies in a contiguous area of memory allocated to the bus. Such a bus is represented by a structure in generated code, which can be helpful when tracing the correspondence between the model and the code.

Compared with nonvirtual buses, virtual buses reduce memory requirements because they do not require a separate contiguous storage block, and execute faster because they do not require copying data to and from that block. In rare cases, virtual and nonvirtual buses that are otherwise identical give different simulation or code execution results.

Virtual buses are the default throughout Simulink except where nonvirtual buses are explicitly required. All nonvirtual subsystems that accept buses, such as referenced models, require nonvirtual buses. A nonvirtual subsystem therefore needs bus objects on all bus inports and outports, as described in "Using Bus Objects" on page 6-8 and this section. Some nonvirtual subsystems impose additional requirements on buses.

Some buses that are not directly connected to nonvirtual subsystems must be nonvirtual also, as described in "Connecting Buses to Inports" on page 6-18 and "Connecting Buses to Root Level Inports" on page 6-19. Not all blocks can accept buses. See "Bus-Capable Blocks" on page 6-16 for more about which blocks can handle which types of buses. You can use a Signal Conversion block to convert a nonvirtual to a virtual bus, and vice versa.

Note Nested buses must all be either virtual or nonvirtual: a bus cannot contain a mixture of virtual and nonvirtual nested buses.

Specifying Nonvirtual Buses

Bus signals do not specify whether they are virtual or nonvirtual; they inherit that specification from the block in which they originate or terminate. Every block that creates or requires a nonvirtual bus must have an associated bus object. Those blocks are:

- Bus Creator
- Inport
- Outport

To specify that a bus is nonvirtual:

1. Associate the block with a bus object, as described in "Associating Bus Objects with Model Entities" on page 6-12.
2. Open the **Block Parameters** dialog of the Bus Creator, Inport, or Outport block.
3. Do one of the following, depending on the type of the block:
   - **Bus Creator:** Select **Output as Virtual Bus**.
   - **Inport:** Select **Signal Specification > Output as nonvirtual bus**.
   - **Outport:** Select **Signal Specification > Output as nonvirtual bus in parent model**.
4. Click **OK** or **Apply**.

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

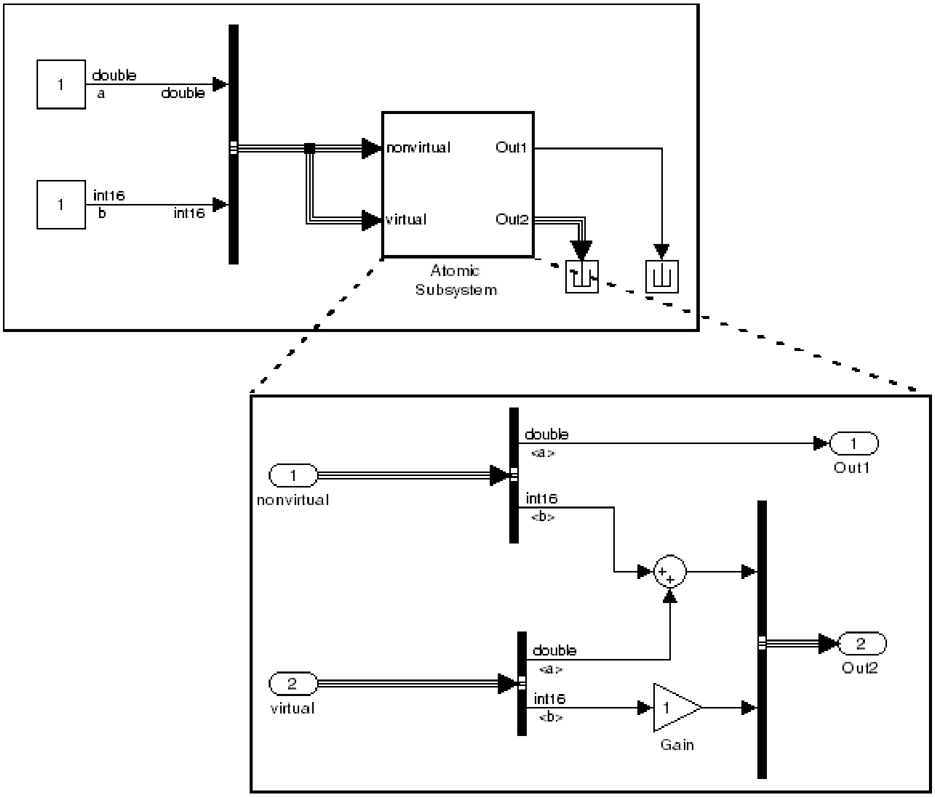
- Merge
- Switch
- Multiport Switch
- Rate Transition
- Unit Delay
- Zero-Order Hold

Some bus-capable blocks impose constraints on bus propagation through them. See the documentation for the blocks in Blocks-Alphabetical List for more information.

Connecting Buses to Inports

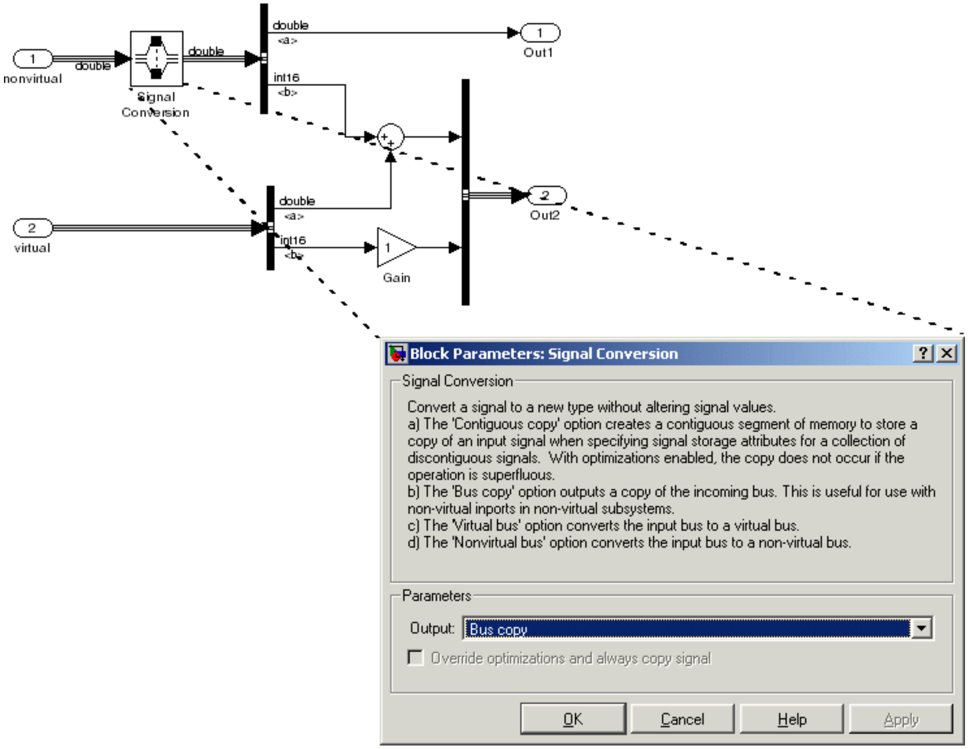
Generally, an Inport block is a virtual block and hence accepts a bus as input. However, an Inport block is nonvirtual if it resides in a conditionally executed or atomic subsystem, including a referenced model, and it or any of its components is directly connected to an output of the subsystem. In such 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 thereby 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 Root Level 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 7-12 and the Simulink.Bus class in the Simulink documentation for more information.

Intermixing Composite Signal Types

Muxes are implemented as indexed vectors, while buses are implemented as structures. Buses can do anything muxes can do and more, but muxes and virtual buses are functionally interchangeable for flat composites of signals that all have the same attributes. See "Using Muxes" on page 6-3 and "Using Buses" on page 6-5 for summaries of the two capabilities and their similarities and differences.

For convenience, Simulink by default allows muxes and virtual buses to be intermixed where automatic conversion is possible. Intermixed composite signals can occur only with muxes and virtual buses where all constituent signals have the same attributes and no nested buses exist. Trying to include a nonvirtual bus in a composite signal with a mux or virtual bus generates an error. All intermixed composite signals fall into one of two categories:

- Mux blocks used to create virtual buses, such as a Mux block that outputs to a Bus Selector block
- Virtual bus signals treated as muxes, such as a bus signal that inputs directly to a Gain block

Neither of these mixtures is compatible with strong type checking, so both increase the likelihood of run-time errors. The MathWorks therefore discourages mixing muxes and virtual buses in new applications, and encourages upgrading existing applications to avoid such mixtures. Simulink provides Configuration Parameters diagnostics, Model Advisor checks, and other tools for detecting and correcting intermixed composite signals.

Note Do not confuse intermixed composite signals with nested buses. A *mixed composite signal* occurs only when some blocks treat a signal as a mux, while other blocks treat that same signal as a bus. See "Nesting Buses" on page 6-6 for information about including one bus within another.

Using Diagnostics for Mixed Composite Signals

Simulink provides two controls on the **Diagnostics > Connectivity** pane that you can use to detect mixed composite signals:

- **Mux blocks used to create bus signals**
- **Bus signal treated as vector**

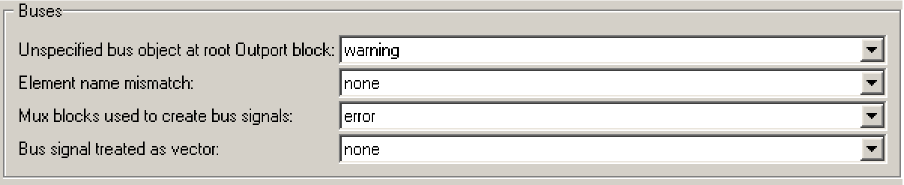
You can use these diagnostics as described in this section, or use the Model Advisor to perform the same checks and also obtain advice about corrections, as described in "Consulting Model Advisor". For complete information about the **Connectivity** pane, see "Connectivity Diagnostics" on page 11-121.

Mux blocks used to create bus signals

To detect and correct muxes that are used as buses:

1 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to `warning` or `error`.

2 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Bus signal treated as vector** to `none`.



Buses

Unspecified bus object at root Outport block: warning

Element name mismatch: none

Mux blocks used to create bus signals: error

Bus signal treated as vector: none

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

4 Build the model.

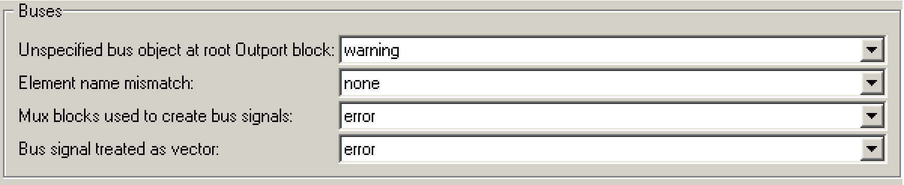
5. Replace blocks as needed to correct any cases of Mux blocks used to create buses. You can use the `slreplace_mux` function to replace all such Mux blocks in a single operation.

For complete information about this option, see the reference documentation for "Mux blocks used to create bus signals" on page 11-122.

Bus signal treated as vector

To detect and correct buses that are used as if they were muxes (vectors):

1. Correct any cases of Mux blocks used to create buses as described above.
2. Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to `error`.
3. Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Bus signal treated as vector** to `warning` or `error`.



Buses

Unspecified bus object at root Outport block: warning

Element name mismatch: none

Mux blocks used to create bus signals: error

Bus signal treated as vector: error

4. Click **OK** or **Apply**.
5. Build the model.
6. Correct the model where needed as described under "Correcting Buses Used as Muxes" on page 6-26.

For complete information about this option, see the reference documentation for "Mux blocks used to create bus signals" on page 11-122.

Note **Bus signal treated as vector** is disabled unless **Mux blocks used to create bus signals** is set to **error**. Setting **Bus signal treated as vector** to **error** has no effect unless you have previously corrected all cases of Mux blocks used to create buses.

Equivalent Simulink Parameter Values

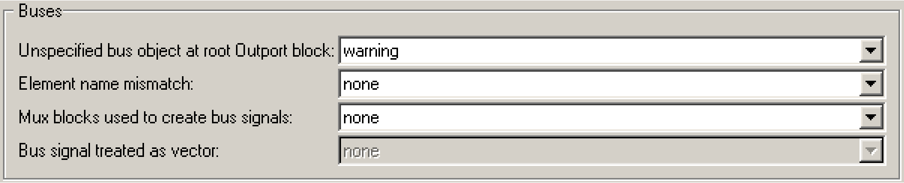
Due to the requirement that **Mux blocks used to create bus signals** be `error` before **Bus signal treated as vector** is enabled, one Simulink parameter, `StrictBusMsg`, can specify all permutations of the two controls. The parameter can have one of five values. The following table shows these values and the equivalent GUI control settings:

Value of `StrictMusMsg` `(API)`	Mux blocks used to create bus signals (GUI)	Bus signal treated as vector (GUI)
`None`	`none`	`none`
`Warning`	`warning`	`none`
`ErrorLevel1`	`error`	`none`
`WarnOnBusInputToNonBusBlock`	`error`	`warning`
`ErrorOnBusInputToNonBusBlock`	`error`	`error`

Using the Model Advisor for Mixed Composite Signals

The Model Advisor provides a convenient way to run both composite signal diagnostics and obtain advice about corrections. To use the Model Advisor to detect and correct mixed composite signals:

1 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to `none`.



Buses

Unspecified bus object at root Outport block: warning

Element name mismatch: none

Mux blocks used to create bus signals: none

Bus signal treated as vector: none

2 Click **OK** or **Apply**.

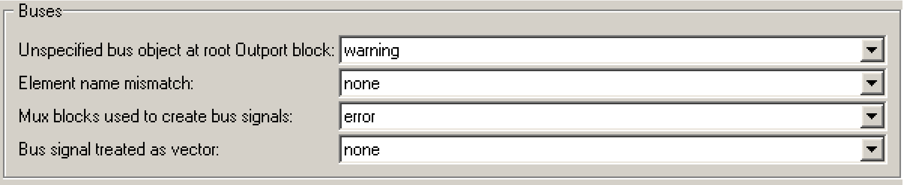
3 Select and run the Model Advisor check **Simulink > Check for proper bus usage**.

The Model Advisor reports any cases of Mux blocks used to create bus signals.

4 Follow the Model Advisor's suggestions to correct any errors reported by the check. You can use the `slreplace_mux` function to replace all such errors in a single operation.

5 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to `error`.

6 Set **Configuration Parameters > Diagnostics > Connectivity > Bus signal treated as vector** to `none`.



Buses

Unspecified bus object at root Outport block: warning

Element name mismatch: none

Mux blocks used to create bus signals: error

Bus signal treated as vector: none

7 Click **OK** or **Apply**.

8 Again run the Model Advisor check **Simulink > Check for proper bus usage**.

The Model Advisor reports any cases of bus signals treated as muxes (vectors).

9 Follow the Model Advisor's suggestions and the information in "Correcting Buses Used as Muxes" on page 6-26 to correct any errors discovered by the check.

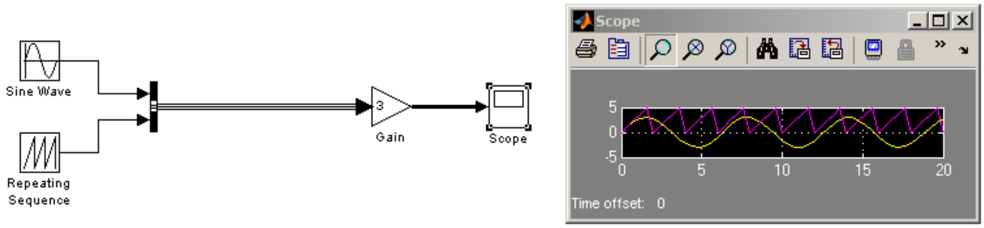
Instructions for using the Model Advisor appear in "Consulting Model Advisor" on page 3-121.

Correcting Buses Used as Muxes

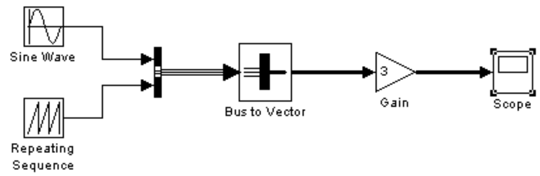
When you discover a bus signal used as a mux, one answer is to reorganize the model by replacing blocks so that the mixture no longer occurs. Where that is undesirable or unfeasible, Simulink provides two capabilities to address the problem:

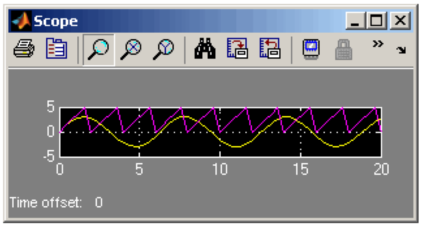
- The Bus to Vector block (Signal Attributes library), which you can insert into any bus used implicitly as a mux to explicitly convert the bus to a mux (vector).
- The `Simulink.BlockDiagram.addBusToVector` function, which automatically inserts Bus to Vector blocks wherever needed.

For example, this figure shows a model that uses a bus as a mux by inputting the bus to a Gain block.



This figure shows the same model, rebuilt after inserting a Bus to Vector block into the bus.





Note that the results of simulation are the same in either case. The Bus to Vector block is virtual, and never affects simulation results, code generation, or performance. For more information, see the reference documentation for the Bus to Vector block and the `Simulink.BlockDiagram.addBusToVector` function.

Bus to Vector Block Backward Compatibility

If you use **Save As** to save a model in a version of Simulink before R2007a (V6.6), Simulink does the following:

- Set the `StrictBusMsg` parameter to `error` if its value is `WarnOnBusInputToNonBusBlock` or `ErrorOnBusInputToNonBusBlock`.
- Replace each Bus to Vector block in the model with a null subsystem that outputs nothing.

The resulting model specifies strong type checking for Mux blocks used to create buses. Before you can use the model, you must reconnect or otherwise correct each signal that contained a Bus to Vector block but is now interrupted by a null subsystem.

Avoiding Mixed Composite Signals When Developing Models

The MathWorks discourages the use of mixed composite signals, and may cease to support them at some future time. The MathWorks therefore recommends upgrading existing models to eliminate any mixed composite signals, and permanently setting **Mux blocks used to create bus signals** and **Bus signal treated as vector** to `error` in all new models and all existing models that may undergo further development.

Note The Bus to Vector block is intended only for use in existing models to facilitate the elimination of implicit conversion of buses into muxes. New models and new parts of existing models should avoid mixing composite signals, and should not use Bus to Vector blocks for any purpose.

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 7-10). In that way, you can avoid introducing data type errors that prevent your model from running to completion or simulating at all.

The following sections further describe working with data types in Simulink:

- "Data Types Supported by Simulink" on page 7-3
- "Fixed-Point Data" on page 7-4

- “Fixed-Point Tool” on page 7-4
- “Block Support for Data and Numeric Signal Types” on page 7-5
- “Creating Signals of a Specific Data Type” on page 7-5
- “Specifying Block Output Data Types” on page 7-6
- “Displaying Port Data Types” on page 7-10
- “Data Type Propagation” on page 7-10
- “Data Typing Rules” on page 7-10
- “Typecasting Signals” on page 7-11

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 “Blocks — Alphabetical List” in the online Simulink reference for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

To view a table that summarizes the data types supported by the blocks in the Simulink® block libraries, execute the following command at the MATLAB® command line:

```
showblockdatatypetable
```

Fixed-Point Data

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 Tool" 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 Tool

Most of the functionality in the Fixed-Point Tool is for use with the Simulink Fixed Point product. However, even if you do not have Simulink Fixed Point, you can use data type override mode to simulate a model that specifies fixed-point data types. In this mode, Simulink replaces fixed-point values with floating-point values when simulating the model. Data type override mode allows you to share fixed-point models with people in your company who do not have Simulink Fixed Point.

To simulate a model in data type override mode:

1. Select **Fixed-Point Settings** from the Simulink **Tools** menu.

   The Fixed-Point Tool appears.

2. Set the **Logging mode** parameter to `Force off`.

3. Set the **Data type override** parameter to `True doubles` or `True singles`.

Note If you specify a `fi` object as a fixed-point parameter in your model, you need a Fixed-Point Toolbox license to simulate the model in data type override mode.

Block Support for Data and Numeric Signal Types

All Simulink blocks accept signals of type `double` by default. Some blocks prefer `boolean` input and others support multiple data types on their inputs. See Simulink Blocks in *Simulink Reference* for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

Creating Signals of a Specific Data Type

You can introduce a signal of a specific data type into a model in any of the following ways:

- Load signal data of the desired type from the MATLAB workspace into your model via a root-level Inport block or a From Workspace block.
- Create a Constant block in your model and set its parameter to the desired type.

- Use a Data Type Conversion block to convert a signal to the desired data type.

Specifying Block Output Data Types

Simulink blocks determine the data type of their outputs by default. In addition, many blocks allow you to override the default output type and specify an output data type explicitly, using the **Output data type** parameter. This field appears when you set the **Output data type mode** parameter on a block's **Signal Data Types** pane to `Specify via dialog`. You can use either data type functions or data type objects to specify the output data type in a block's **Output data type** field. Data type objects require more work initially but simplify making model-wide changes to output data types (see "Working with Data Objects" on page 7-12) and allow you to use application-specific aliases for data types. For more information, see

- "Specifying Output Data Types Via Simulink Data Type Functions" on page 7-6
- "Specifying Output Data Types Via Simulink Data Objects" on page 7-8

Specifying Output Data Types Via Simulink Data Type Functions

You can use the following data type functions in a block's **Output data type** field to specify its output data type.

Name	Description
`float`	Create a MATLAB structure describing a floating-point data type
`sfix`	Create a MATLAB structure describing a signed generalized fixed-point data type
`sfrac`	Create a MATLAB structure describing a signed fractional data type (requires a Simulink Fixed Point license)
`sint`	Create a MATLAB structure describing a signed integer data type
`ufix`	Create a MATLAB structure describing an unsigned generalized fixed-point data type

Name	Description
`ufrac`	Create a MATLAB structure describing an unsigned fractional data type (requires a Simulink Fixed Point license)
`uint`	Create a MATLAB structure describing an unsigned integer data type

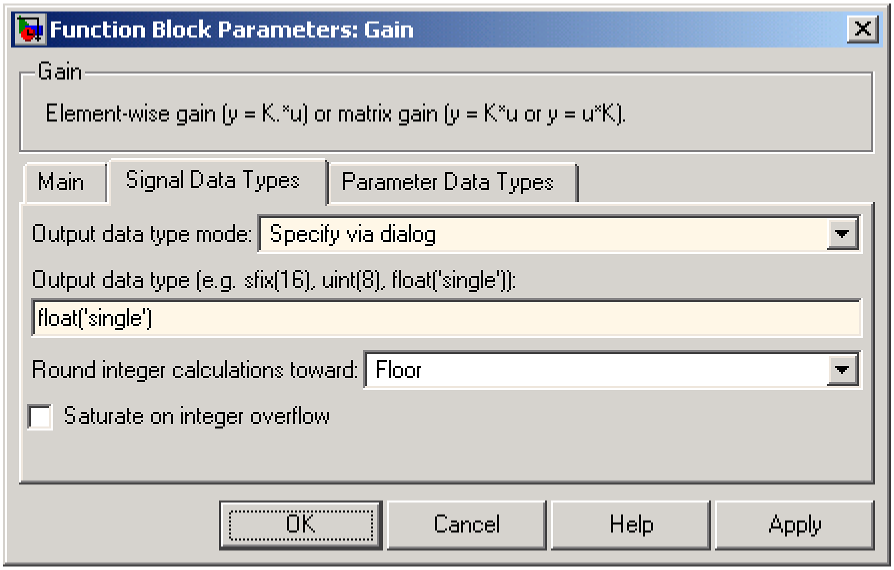
These functions accept one or more arguments that specify the data type and return a MATLAB structure that specifies the same data type. For example, to use the `float` function to set the output data type of a Gain block to `single`:

1. Double-click the Gain block to open its block parameter dialog box.

2. On the **Signal Data Types** pane, set the **Output data type mode** parameter to `Specify via dialog`.

   The **Output data type** field appears.

3. In the **Output data type** field, enter `float('single')` to specify the output data type as `single`.



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Output data type mode: Specify via dialog

Output data type (e.g. sfix(16), uint(8), float('single')): float('single')

Round integer calculations toward: Floor

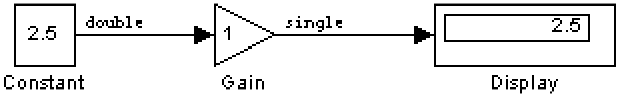
□ Saturate on integer overflow

4 Click **OK** to apply the changes and close the dialog box.

5 Select **Format > Port/Signal Displays > Port Data Types** to display port data types (see “Displaying Port Data Types” on page 7-10).

6 Select **Edit > Update Diagram** (**Ctrl+D**) to update your diagram.

The diagram reveals that the data type of the Gain block’s output is `single`:



Specifying Output Data Types Via Simulink Data Objects

You can specify data types for block outputs using instances of either the `Simulink.NumericType` class or the `Simulink.AliasType` class. Each of these classes allows you to create an object in the MATLAB workspace that describes a data type; however, data type aliases also allow you to customize the data type name that appears in the Simulink model window and the code generated by Real-Time Workshop. For more information about Simulink data objects, see “Working with Data Objects” on page 7-12.

Suppose you want the output data type of a Gain block to be `single`. You can achieve this using a numeric type object as follows:

1 Enter the following command at the MATLAB prompt to create an instance of the `Simulink.NumericType` class:

```
my_datatype = Simulink.NumericType
```

Note Alternatively, you can use the `fixdt` function to instantiate the `Simulink.NumericType` class.

2 Specify `single` for the `DataTypeMode` property of the numeric type object:

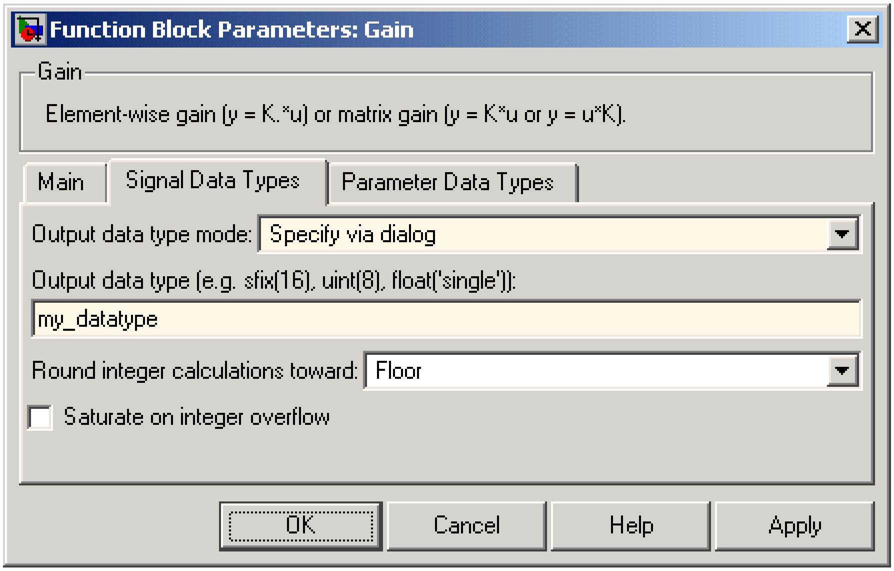
```
my_datatype.DataTypeMode = 'single'
```

3 Double-click the Gain block in the model window to access its block parameter dialog box.

4 On the **Signal Data Types** pane, set the **Output data type mode** parameter to `Specify via dialog`.

The **Output data type** field appears.

5 In the **Output data type** field, enter the numeric type object name as shown here:



Function Block Parameters: Gain

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Output data type mode: Specify via dialog

Output data type (e.g. sfix(16), uint(8), float('single')):

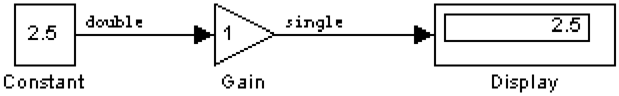
my_datatype

Round integer calculations toward: Floor

□ Saturate on integer overflow

Click the **OK** button to apply the changes and close the dialog box.

6 Simulate your model with the **Port Data Types** option enabled (see “Displaying Port Data Types” on page 7-10). Note that the data type of the Gain block output is `single`:



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, press **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 7-11 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.
- In general, a block outputs the data type that appears at its inputs.

  Significant exceptions include Constant blocks and Data Type Conversion blocks, whose output data types are determined by block parameters.
- Virtual blocks accept signals of any type on their inputs.

Examples of virtual blocks include Mux and Demux blocks and unconditionally executed subsystems.

- The elements of a signal array connected to a port of a nonvirtual block must be of the same data type.
- The signals connected to the input data ports of a nonvirtual block cannot differ in type.
- Control ports (for example, Enable and Trigger ports) accept any data type.
- Solver blocks accept only `double` signals.
- Connecting a non-`double` signal to a block disables zero-crossing detection for that block.

Typecasting Signals

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.

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 following topics describe features and procedures for working with all data objects, regardless of type. For information on working with specific kinds of data objects, see "Data Object Classes" in the online Simulink reference.

- "About Data Object Classes" on page 7-13
- "About Data Object Methods" on page 7-13
- "Using the Model Explorer to Create Data Objects" on page 7-15
- "About Object Properties" on page 7-17
- "Changing Object Properties" on page 7-17
- "Handle Versus Value Classes" on page 7-19
- "Comparing Data Objects" on page 7-21
- "Saving and Loading Data Objects" on page 7-21
- "Using Data Objects in Simulink Models" on page 7-22
- "Creating Persistent Data Objects" on page 7-22
- "Data Object Wizard" on page 7-22

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” 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 7-28).

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.

About Data Object Methods

Data classes define functions, called methods, for creating and manipulating the objects that they define. A class may define any of the following kinds of methods.

Dynamic Methods

A dynamic method is a method whose identity depends on its name and the class of an object specified implicitly or explicitly as its first argument. You can use either function or dot notation to specify this object, which must be an instance of the class that defines the method or an instance of a subclass of the class that defines the method. For example, suppose class A defines a method called setName that assigns a name to an instance of A. Further, suppose the MATLAB workspace contains an instance of A assigned to the variable obj. Then, you can use either of the following statements to assign the name 'foo' to obj:

```
obj.setName('foo');
setName(obj, 'foo');
```

A class may define a set of methods having the same name as a method defined by one of its super classes. In this case, the method defined by the subclass overrides the behavior of the method defined by the parent class. Simulink determines which method to invoke at runtime from the class of the object that you specify as its first or implicit argument. Hence, the term dynamic method.

Note Most Simulink data object methods are dynamic methods. Unless the documentation for a method specifies otherwise, you can assume that a method is a dynamic method.

Static Methods

A static method is a method whose identity depends only on its name and hence cannot change at runtime. To invoke a static method, use its fully qualified name, which includes the name of the class that defines it followed by the name of the method itself. For example, Simulink.ModelAdvisor class defines a static method named getModelAdvisor. The fully qualified name of this static method is Simulink.ModelAdvisor.getModelAdvisor. The following example illustrates invocation of a static method.

```
ma = Simulink.ModelAdvisor.getModelAdvisor('vdp');
```

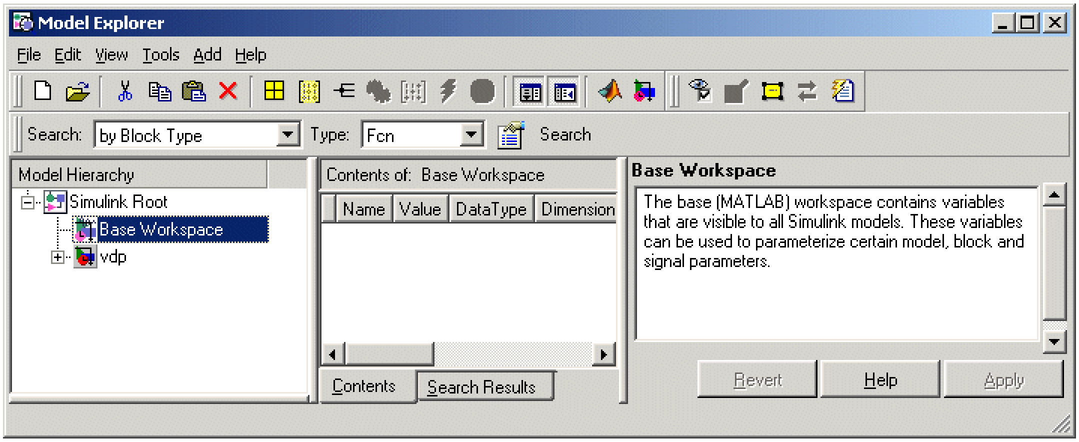
Constructors

Every data class defines a method for creating instances of that class. The name of the method is the same as the name of the class. For example, the name of the `Simulink.Parameter` class's constructor is `Simulink.Parameter`. The constructors defined by Simulink data classes take no arguments.

The value returned by a constructor depends on whether its class is a handle class or a value class. The constructor for a handle class returns a handle to the instance that it creates if the class of the instance is a handle class; otherwise, it returns the instance itself (see "Handle Versus Value Classes" on page 7-19.

Using the Model Explorer to Create Data Objects

You can use the Model Explorer (see "The Model Explorer" on page 10-2) as well as MATLAB commands to create data objects. To use the Model Explorer, first select the workspace in which you want to create the object in the Model Explorer's **Model Hierarchy** pane.



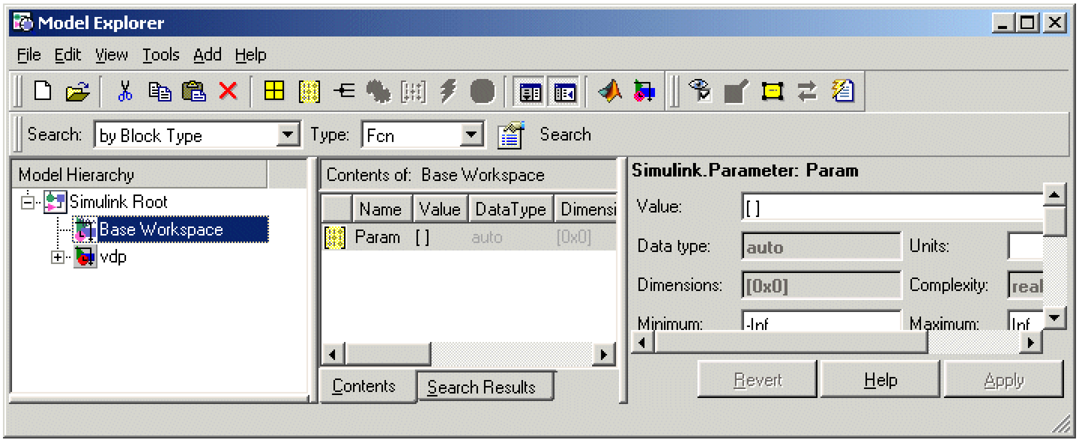
Search: by Block Type

Type: Fcn

Base Workspace

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

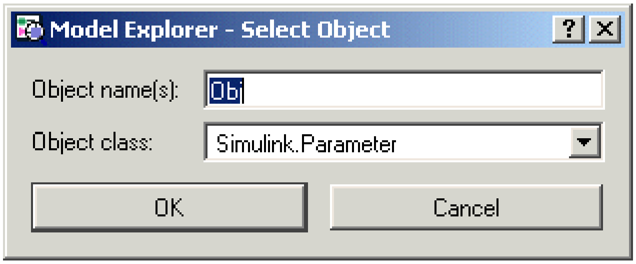
Contents of: Base Workspace

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.

About Object Properties

Object properties are variables associated with an object that specify properties of the entity that the object represents, for example, the size of a data type. The object's class defines the names, value types, default values, and valid value ranges of the object's properties.

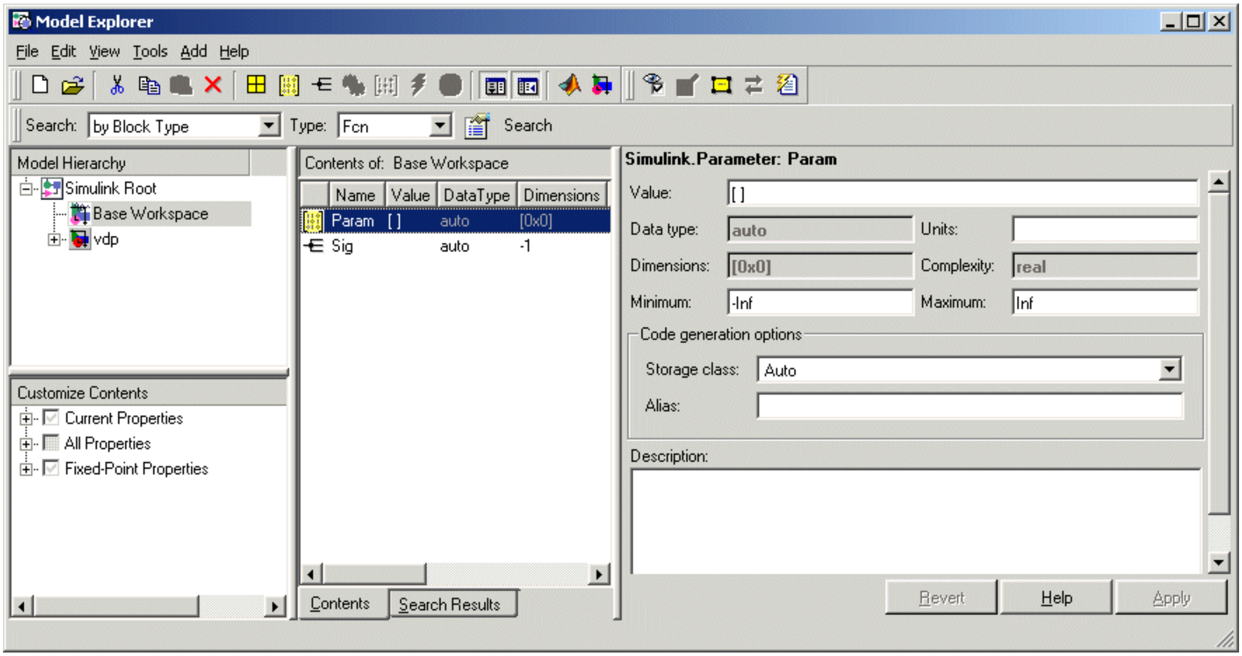
Changing Object Properties

You can use either the Model Explorer (see "Using the Model Explorer to Change an Object's Properties" on page 7-17) or MATLAB commands to change a data object's properties (see "Using MATLAB Commands to Change Workspace Data" on page 3-109).

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 10-9. 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 7-19), PROPERTY is the property's name, and VALUE is the property's value. For example, the following MATLAB code creates a data type alias object (i.e., an instance of Simulink.AliasType) and sets its base type to uint8:

```
gain= Simulink.AliasType;
gain.DataType = 'uint8';
```

You can use dot notation recursively to get and set the properties of objects that are values of other object's properties, e.g.,

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

Handle Versus Value Classes

Simulink data object classes fall into two categories: value classes and handle classes.

About Value Classes

The constructor for a *value* class (see "Constructors" on page 7-15) returns an instance of the class and the instance is permanently associated with the MATLAB variable to which it is initially assigned. Reassigning or passing the variable to a function causes MATLAB to create and assign or pass a copy of the original object.

For example, Simulink.NumericType is a value class. Executing the following statements

```
>> x = Simulink.NumericType;
>> y = x;
```

creates two instances of class Simulink.NumericType in the workspace, one assigned to the variable x and the other to y.

About Handle Classes

The constructor for a *handle* class returns a handle object. The handle can be assigned to multiple variables or passed to functions without causing a copy of the original object to be created. For example, Simulink.Parameter class is a handle class. Executing

```
>> x = Simulink.Parameter;
>> y = x;
```

creates only one instance of `Simulink.Parameter` class in the MATLAB workspace. Variables x and y both refer to the instance via its handle.

A program can modify an instance of a handle class by modifying any variable that references it, e.g., continuing the previous example,

```
>> x.Description = 'input gain';
>> y.Description

ans =
input gain
```

Most Simulink data object classes are value classes. Exceptions include `Simulink.Signal` and `Simulink.Parameter` class.

You can determine whether a variable is assigned to an instance of a class or to a handle to that class by evaluating it at the MATLAB command line. MATLAB appends the text `(handle)` to the name of the object class in the value display, e.g.,

```
>> gain = Simulink.Parameter

gain =

Simulink.Parameter (handle)
        RTWInfo: [1x1 Simulink.ParamRTWInfo]
    Description: ''
       DocUnits: ''
            Min: -Inf
            Max: Inf
          Value: []
       DataType: 'auto'
     Complexity: 'real'
     Dimensions: [0 0]
```

Copying Handle Classes

Use the copy method of a handle class to create copies of instances of that class. For example, Simulink.ConfigSet is a handle class that represents model configuration sets. The following code creates a copy of the current model's active configuration set and attaches it to the model as an alternate configuration geared to model development.

```
activeConfig = getActiveConfigSet(gcs);
develConfig = activeConfig.copy;
develConfig.Name = 'develConfig';
attachConfigSet(gcs, develConfig);
```

Comparing Data Objects

Simulink data objects provide a method, named isContentEqual, that determines whether object property values are equal. This method compares the property values of one object with those belonging to another object and returns true (1) if all of the values are the same or false (0) otherwise. For example, the following code instantiates two signal objects (A and B) and specifies values for particular properties.

```
A = Simulink.Signal;
B = Simulink.Signal;
A.DataType = 'int8';
B.DataType = 'int8';
A.InitialValue = '1.5';
B.InitialValue = '1.5';
```

Afterward, use the isContentEqual method to verify that the object properties of A and B are equal.

```
>> result = A.isContentEqual(B)

result =

     1
```

Saving and Loading Data Objects

You can use the 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 Class Designer** (see “Subclassing Simulink Data Classes” on page 7-28) or at the command line and save them in a MAT-file (see “Saving and Loading Data Objects” on page 7-21). Then use either the script or a load command as the `PreLoadFcn` callback routine for the model that uses the objects. For example, suppose you save the data objects in a file named `data_objects.mat` and the model to which they apply is open and active. Then, entering the following command

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

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

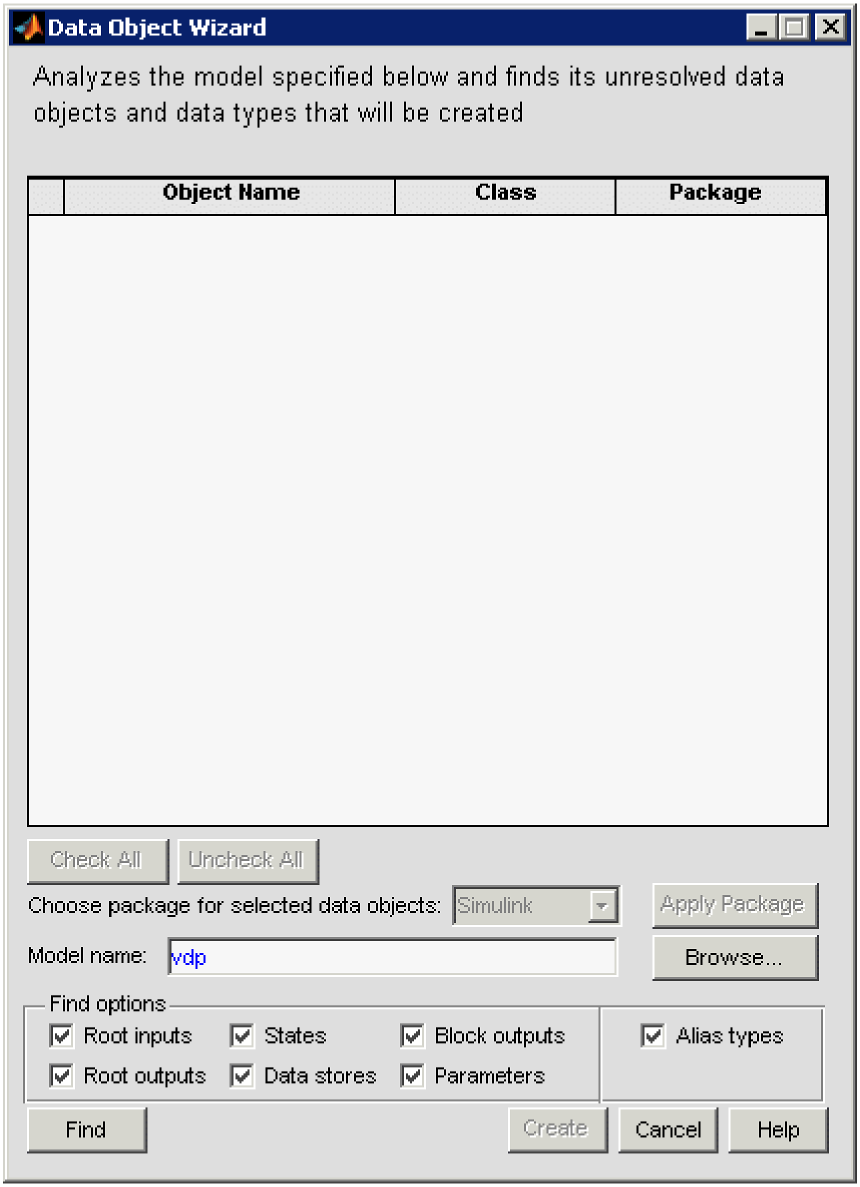
Data Object Wizard

The Data Object Wizard allows you to determine quickly which model data are not associated with data objects and to create and associate data objects with the data.

To use the wizard to create data objects:

1 Select **Tools > Data Object Wizard** from the Model Editor’s tool bar.

The Data Object Wizard appears.



Analyzes the model specified below and finds its unresolved data objects and data types that will be created

	Object Name	Class	Package

Choose package for selected data objects: Simulink

Model name: vdp

Find options

- ☑ Root inputs
- ☑ Root outputs
- ☑ States
- ☑ Data stores
- ☑ Block outputs
- ☑ Parameters
- ☑ Alias types

2 Enter, if necessary, the name of the model you want to search in the wizard's **Model name** field.

By default the wizard displays the name of the model from which you opened the wizard. You can enter the name of another model in this field. If the model is not open, the wizard opens the model.

3 In **Find options**, uncheck any of the data object types that you want the search to ignore.

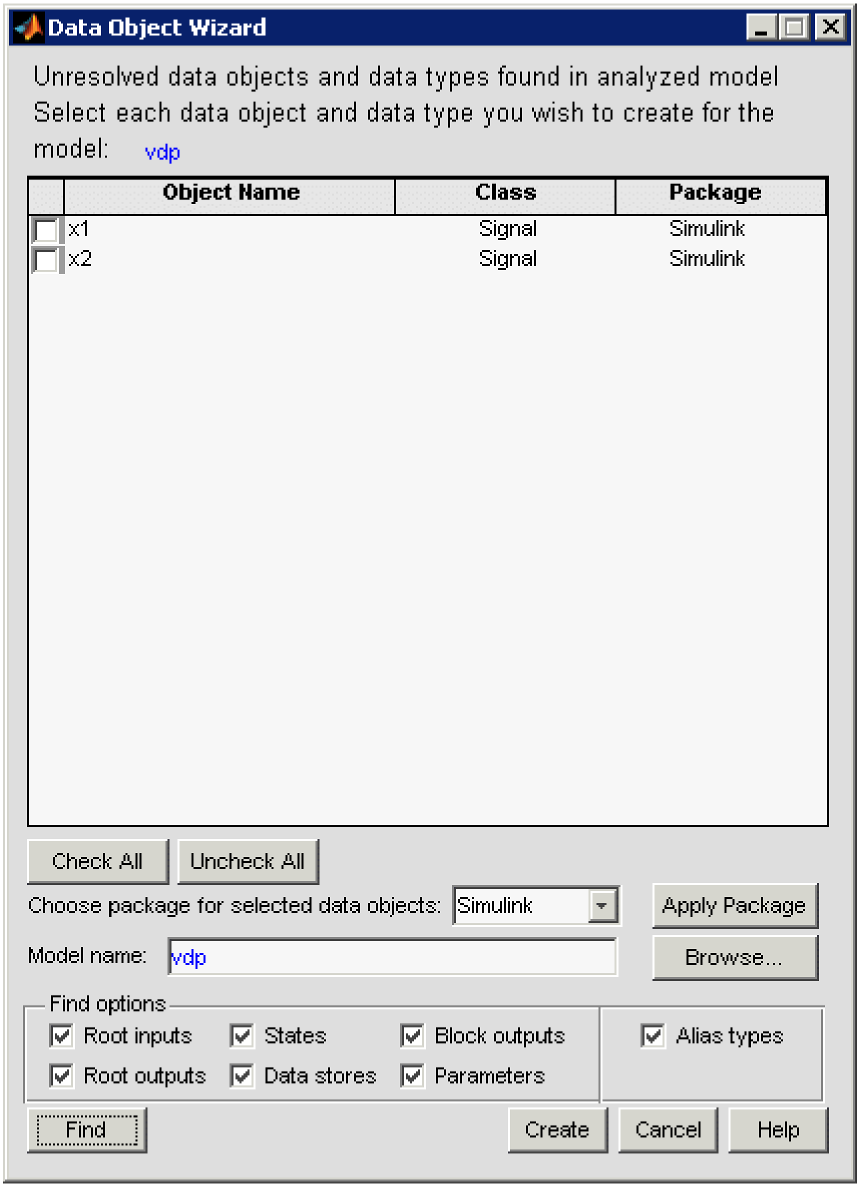
The search options include:

Option	Description
Root inputs	Named signals from root-level input ports
Root outputs	Named signals from root-level output ports
States	States associated with any instances of the following discrete block types: Discrete Filter Discrete State-Space Discrete-Time Integrator Discrete Transfer Fcn Discrete Zero-Pole Memory Unit Delay
Data stores	Data stores (see "Working with Data Stores" on page 3-115)
Block outputs	Named signals emitted by non-root-level blocks.

Option	Description
Parameters	• Parameters of any instances of the following block types: Constant Gain Lookup Table Lookup Table (2-D) Relay • Stateflow data with a **Scope** of `Parameter`. See "Bringing Simulink Parameters into Stateflow" in the online Stateflow documentation for more information.
Alias types	Data whose data type is a registered custom data type. This option applies only if you are generating code from the model. See Creating Data Objects with Data Object Wizard in the Real-Time Workshop Embedded Coder documentation for more information.

4 Click the wizard's **Find** button.

The wizard displays the search results in the data objects table.



Data Object Wizard

Unresolved data objects and data types found in analyzed model
Select each data object and data type you wish to create for the model: vdp

	Object Name	Class	Package
□	x1	Signal	Simulink
□	x2	Signal	Simulink

Choose package for selected data objects: Simulink

Model name: vdp

Find options

☑ Root inputs ☑ States ☑ Block outputs ☑ Alias types

☑ Root outputs ☑ Data stores ☑ Parameters

5 Check the data for which you want the wizard to create data objects.

6 If you want the wizard to use data object classes from a package other than Simulink's standard class package to create the data objects, select the package from the **Choose package for selected data objects** list and then select **Apply Package** to confirm your choice.

7 Click **Create**.

   The wizard creates data objects of the appropriate class for the data selected in the search results table.

Note Use the Model Explorer to view and edit the created data objects.

Subclassing Simulink Data Classes

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.

The following topics provide more information:

- "Creating a Data Object Class" on page 7-28
- "Specifying a Parent for a Class" on page 7-32
- "Defining Class Properties" on page 7-34
- "Defining Enumerated Property Types" on page 7-35
- "Creating Initialization Code" on page 7-38
- "Creating a Class Package" on page 7-39

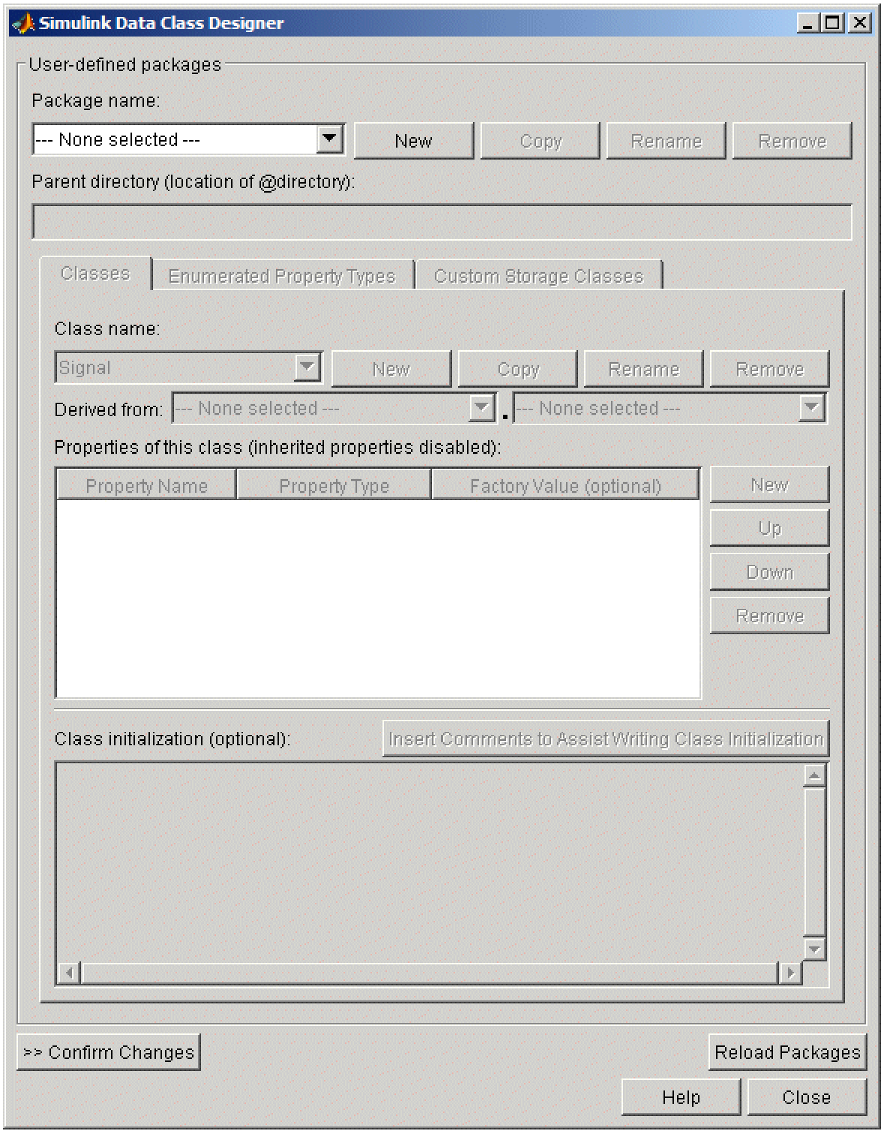
Note You can use the **Data Class Designer** to create custom storage classes. See the *Real-Time Workshop User's Guide* for information on custom storage classes. The **Data Class Designer** is not, however, intended as a tool for code generating enumerated types.

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: --- None selected ---

Class name: Signal

Derived from: --- None selected --- . --- None selected ---

Simulink Data Class Designer
User-defined packages
New
Copy
Rename
Remove
Parent directory (location of @directory):
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 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 7-39 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 **OK** on the **Classes** pane to create the specified class in memory.

6 Select a parent class for the new class (see "Specifying a Parent for a Class" on page 7-32).

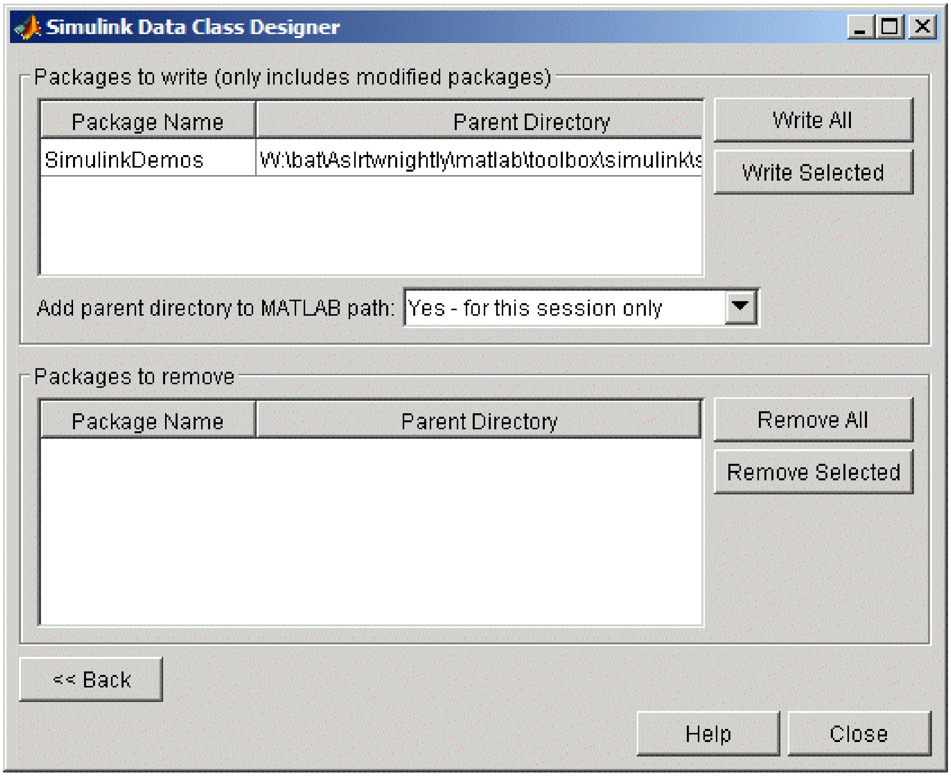
   If you select `Simulink.Signal` or `Simulink.Parameter` as the parent class, Simulink Data Class Designer displays a check box labeled **Create your own custom storage classes for this class**. You can ignore this option if you do not intend to use Real-Time Workshop Embedded Coder to generate code from models that reference this data object class. Otherwise, select this check box to cause Simulink Data Class Designer to create custom storage classes for this data object class (see "Creating Packages with CSC Definitions" for more information).

7 Define the properties of the new class (see "Defining Class Properties" on page 7-34).

8 If necessary, create initialization code for the new class (see "Creating Initialization Code" on page 7-38).

9 Click **Confirm Changes**.

Simulink displays the **Confirm Changes** pane.



Package Name	Parent Directory
SimulinkDemos	W:\bat\Aslrtwnightly\matlab\toolbox\simulink\s

Add parent directory to MATLAB path: Yes - for this session only

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

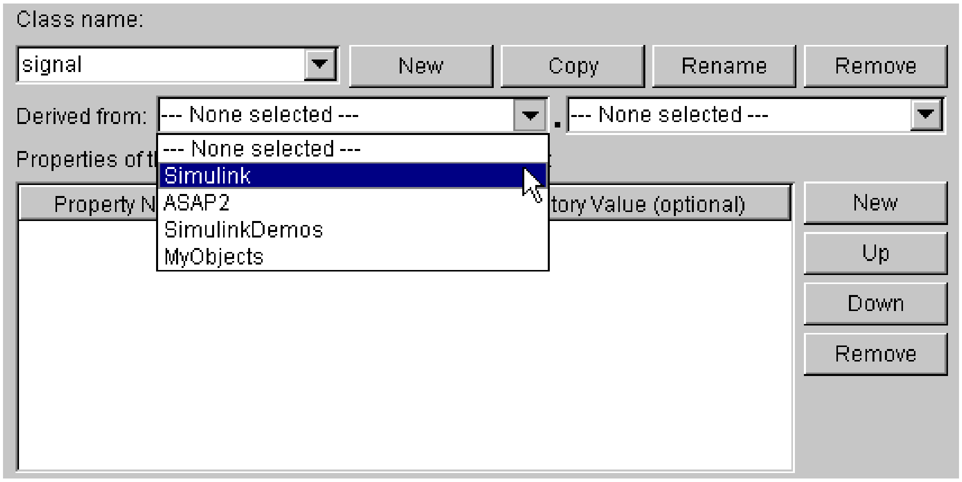
To remove a class definition from the currently selected package, select the class in the **Classes** pane and click **Remove**. Simulink removes the class from the in-memory definition of the class. Save the package that formerly contained the class.

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.

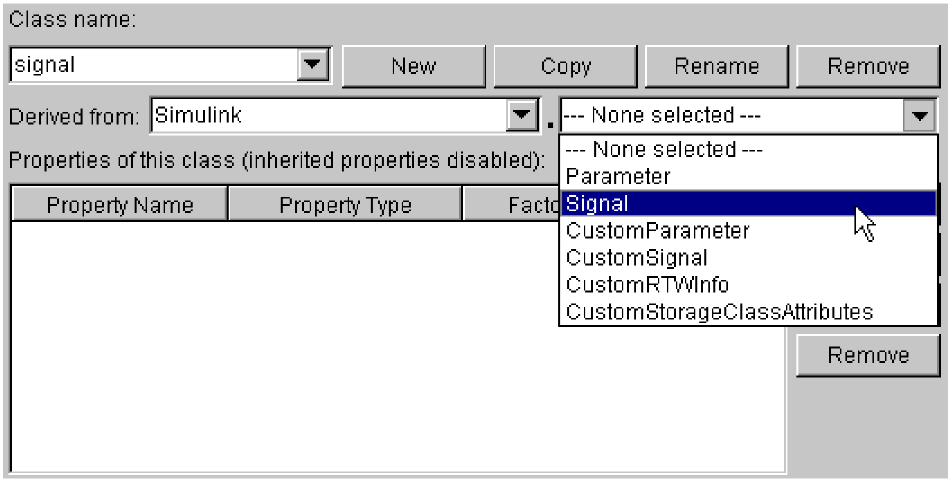
2 Select the package name of the parent class from the left-hand **Derived from** list box.



Class name: signal

Derived from: --- None selected --- . --- None selected ---

3 Select the parent class from the right-hand **Derived from** list.

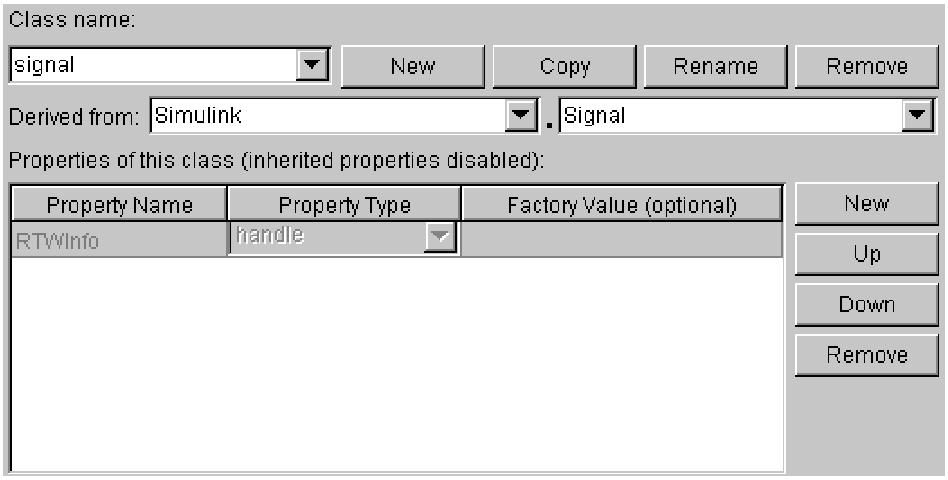


Class name: signal

Derived from: Simulink . --- None selected ---

Properties of this class (inherited properties disabled):

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.

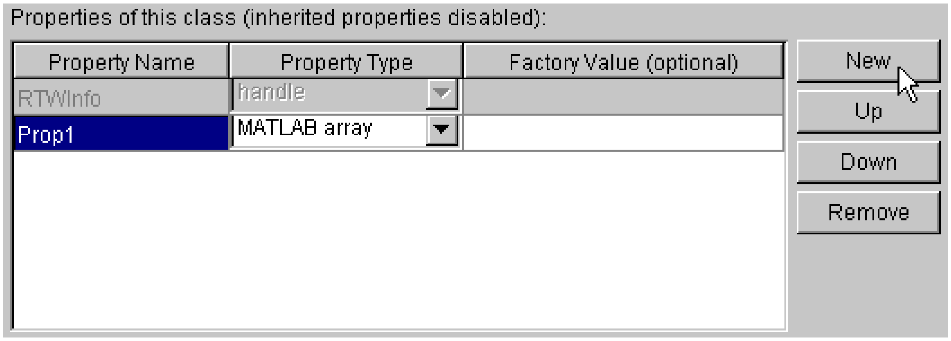
Defining Class Properties

To add a property to a class:

1 Select the name of the class from the **Class name** field on the **Classes** pane.

2 Click 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 7-35).

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 7-38 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.

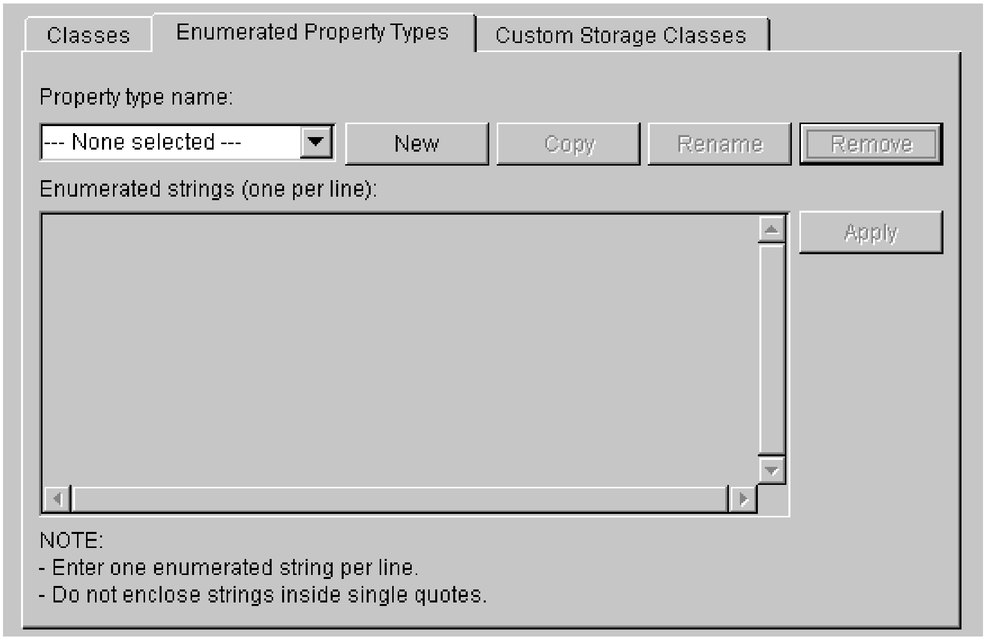
6 Save the package containing the class with new or changed properties.

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**.

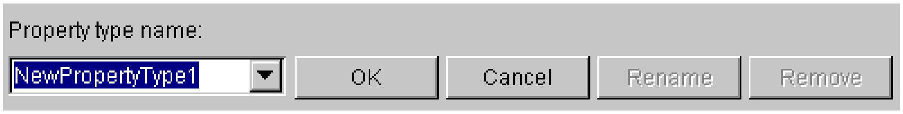


Property type name: --- None selected ---

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.

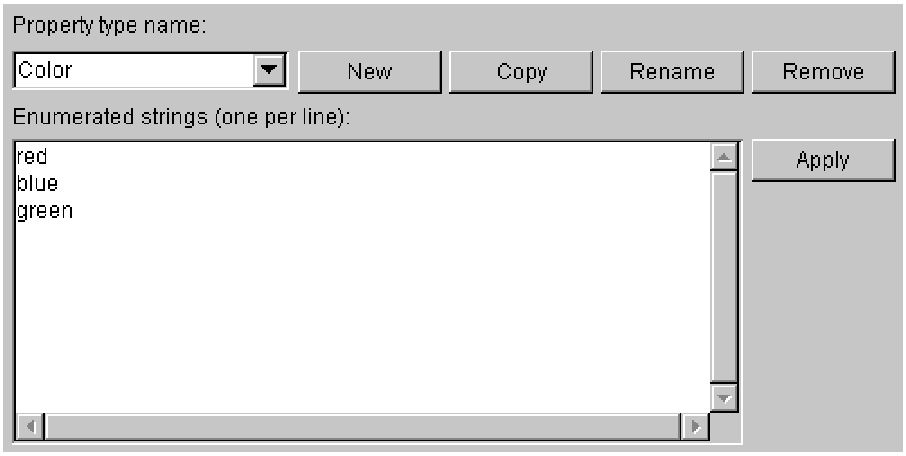
The currently selected package defines an enumerated property type and the type can be referenced only in the package that defines it. However, the name of the enumerated property type must be globally unique. There cannot be any other built-in or user-defined enumerated property with the same name. 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');
```

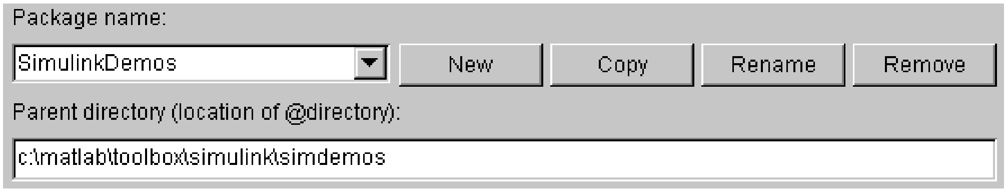
The following code in the instantiation function would perform the required initialization:

```
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:

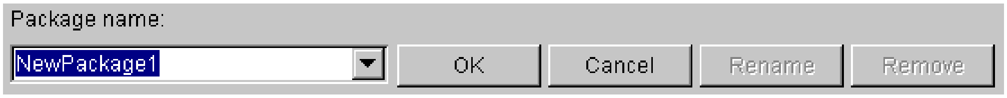
1 Click the **New** button next to the **Package name** field of the Data Class Designer.



Package name: SimulinkDemos

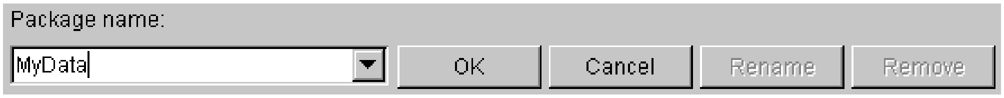
Parent directory (location of @directory): c:\matlab\toolbox\simulink\simdemos

Simulink displays a default package name in the **Package name** field.



Package name: NewPackage1

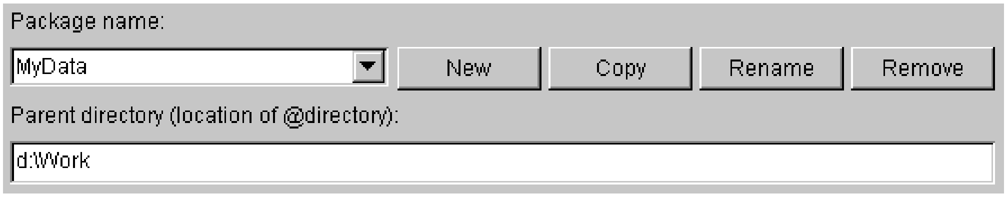
2 Edit the **Package name** field to contain the package name that you want.



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.



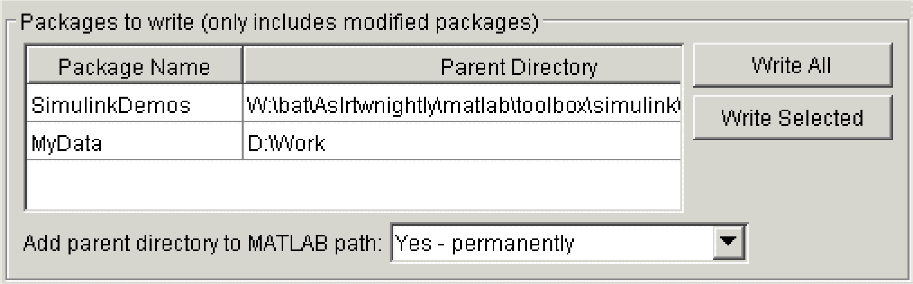
Package name: MyData

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
SimulinkDemos	W:\bat\Aslrtwnightly\matlab\toolbox\simulink\
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** option is set to `Yes - permanently`. The default is `Yes - for this session only`.

This adds the path of the new package's parent directory to the MATLAB path.

7 Click **Write all** or select the new package and click **Write selected** to save the new package.

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.

Working with Lookup Tables

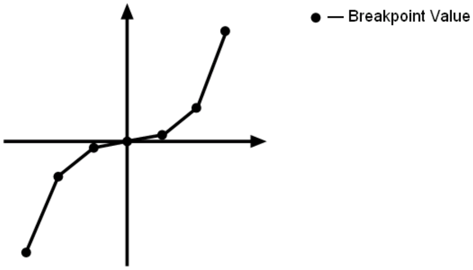
The following sections describe how to select and use lookup table blocks in a Simulink model.

About Lookup Table Blocks

A *lookup table* block uses an array of data to map input values to output values, approximating a mathematical function. Given input values, Simulink performs a "lookup" operation to retrieve the corresponding output values from the table. If the lookup table does not define the input values, the block estimates the output values based on nearby table values.

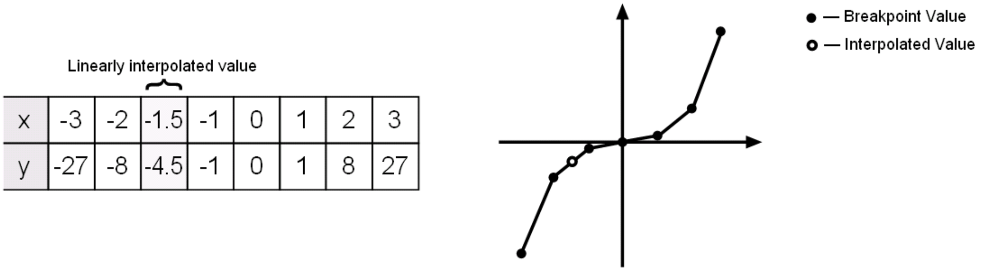
The following example illustrates a one-dimensional lookup table that approximates the function $y = x^3$. The lookup table defines its output (y) data discretely over the input (x) range `[-3, 3]`. The following table and graph illustrate the input/output relationship:

x	-3	-2	-1	0	1	2	3
y	-27	-8	-1	0	1	8	27



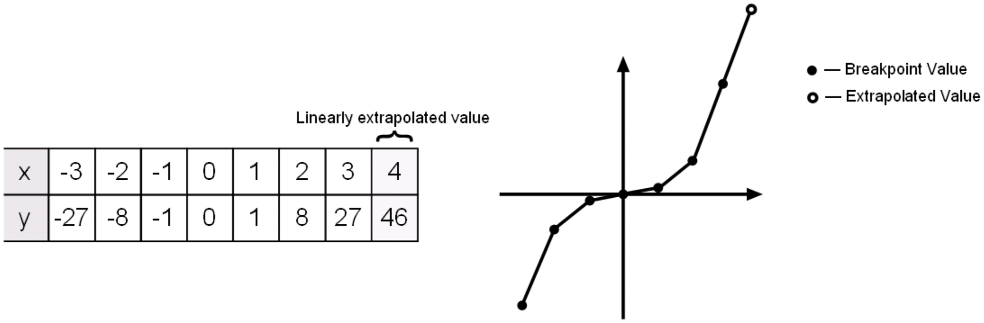
An input of -2 enables the table to look up and retrieve the corresponding output value (-8). Likewise, the lookup table outputs 27 in response to an input of 3.

When the lookup table block encounters an input that does not match any of the table's x values, it can interpolate or extrapolate the answer. For instance, the lookup table does not define an input value of -1.5; however, the block can linearly interpolate the nearest data points (-2, -8) and (-1, -1) to estimate and return a value of -4.5.



x	-3	-2	-1.5	-1	0	1	2	3
y	-27	-8	-4.5	-1	0	1	8	27

Similarly, although the lookup table does not include data for *x* values beyond the range of `[-3, 3]`, the block can extrapolate values using a pair of data points at either end of the table. Given an input value of 4, the lookup table block linearly extrapolates the nearest data points (2, 8) and (3, 27) to estimate an output value of 46.



x	-3	-2	-1	0	1	2	3	4
y	-27	-8	-1	0	1	8	27	46

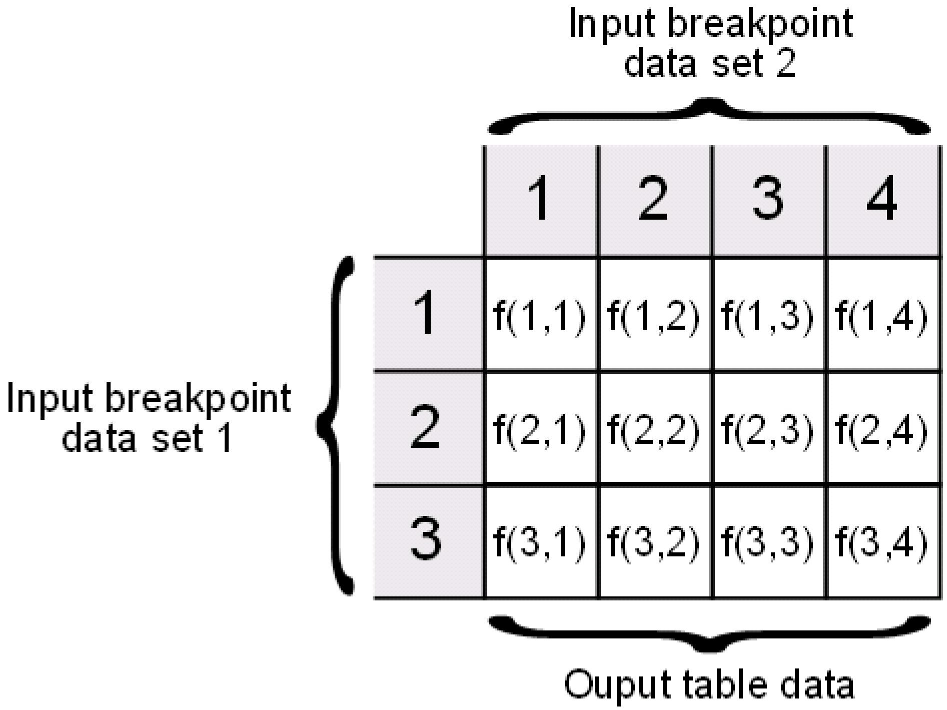
Since table lookups and simple estimations can be faster than mathematical function evaluations, using lookup table blocks may result in speed gains when simulating a model. Consider using lookup tables in lieu of mathematical function evaluations when

- An analytical expression is expensive to compute
- No analytical expression exists, but the relationship has been determined empirically

Simulink provides a broad assortment of lookup table blocks, each geared for a particular type of application. The sections that follow outline the different offerings, suggest how to choose the lookup table best suited to your application, and explain how to interact with the various lookup table blocks.

Anatomy of a Lookup Table

The following figure illustrates the anatomy of a two-dimensional lookup table. Vectors or *breakpoint data sets* and an array, referred to as *table data*, constitute the lookup table.

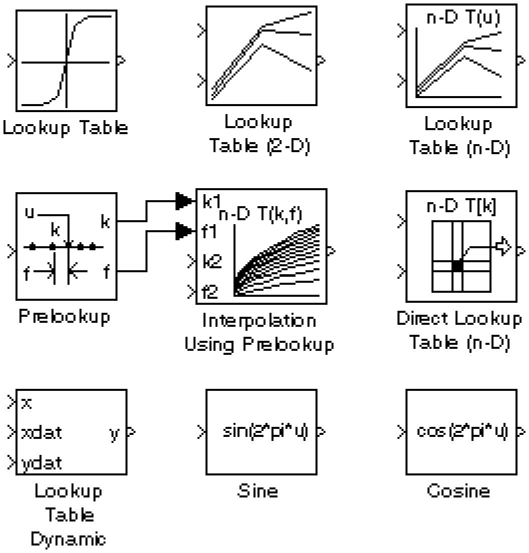


	1	2	3	4
1	f(1,1)	f(1,2)	f(1,3)	f(1,4)
2	f(2,1)	f(2,2)	f(2,3)	f(2,4)
3	f(3,1)	f(3,2)	f(3,3)	f(3,4)

Each breakpoint data set is an index of input values for a particular dimension of the lookup table. The array of table data serves as a sampled representation of a function evaluated at the breakpoint values. Lookup table blocks use breakpoint data sets to relate a table's input values to the output values that it returns.

Lookup Tables Block Library

Simulink offers several lookup table blocks in the Lookup Tables block library shown here:



The following table summarizes the purpose of each block in the library:

Block Name	Description
Lookup Table	Approximate a one-dimensional function.
Lookup Table (2-D)	Approximate a two-dimensional function.
Lookup Table (n-D)	Approximate an N-dimensional function.
Prelookup	Compute index and fraction for Interpolation Using Prelookup block.
Interpolation Using Prelookup	Use the output of a Prelookup block to accelerate approximation of an N-dimensional function.
Direct Lookup Table (n-D)	Index into an N-dimensional table to retrieve the corresponding outputs.
Lookup Table Dynamic	Approximate a one-dimensional function using a dynamically specified table.
Sine	Use a fixed-point lookup table to approximate the sine wave function.
Cosine	Use a fixed-point lookup table to approximate the cosine wave function.

Choosing a Lookup Table

Selecting the lookup table block best suited to your application depends on factors such as the characteristics of your data set, the accuracy and smoothness of the data to be returned, and the dynamics of table inputs. The following sections discuss each of these factors.

- "Data Set Dimensionality" on page 8-8
- "Data Set Numeric and Data Types" on page 8-8
- "Data Accuracy and Smoothness" on page 8-9
- "Dynamics of Table Inputs" on page 8-9
- "Efficiency of Performance" on page 8-9
- "Summary of Lookup Table Block Features" on page 8-10

Data Set Dimensionality

In some cases, the dimensions of your data set dictate which of the lookup table blocks is right for your application. If you are approximating a one-dimensional function, consider using either the Lookup Table or Lookup Table Dynamic block. If you are approximating a two-dimensional function, consider the Lookup Table (2-D) block. Blocks such as the Lookup Table (n-D) and Direct Lookup Table (n-D) allow you to approximate an *N*-dimensional function of even higher order.

Data Set Numeric and Data Types

The numeric and data types of your data set influence the decision of which lookup table block is most appropriate. Although all lookup table blocks support real numbers, the Direct Lookup Table (n-D) block is the only lookup table block that supports complex table data. Similarly, all lookup table blocks support `double` and `single` data types. However, only certain lookup table blocks (e.g., the Lookup Table and Lookup Table (2-D) blocks) support integer and `boolean` data types. Even fewer lookup table blocks support fixed-point data.

Data Accuracy and Smoothness

The desired accuracy and smoothness of the data returned by a lookup table determine which of the blocks should be used. Most blocks provide options to perform interpolation and extrapolation, improving the accuracy of values that fall between or outside of the table data, respectively. For instance, the Lookup Table, Lookup Table (2-D), and Lookup Table Dynamic blocks perform linear interpolation and extrapolation, while the Lookup Table (n-D) block performs either linear or cubic spline interpolation and extrapolation. In contrast, the Direct Lookup Table (n-D) block performs table lookups without any interpolation and extrapolation. You can achieve a mix of interpolation and extrapolation methods by using the Prelookup block with the Interpolation Using Prelookup block.

Dynamics of Table Inputs

The dynamics of the lookup table inputs impact which of the lookup table blocks is ideal for your application. The blocks use a variety of index search methods to relate the lookup table inputs to the table's breakpoint data sets. Most of the lookup table blocks offer a binary search algorithm, which performs well if the inputs change significantly from one time step to the next. The Lookup Table (n-D) and Prelookup blocks offer a linear search algorithm. Using this algorithm with the option that resumes searching from the previous result performs well if the inputs change slowly. Certain lookup table blocks also provide a search algorithm that is tailored for breakpoint data sets composed of evenly spaced breakpoints. Note that you can achieve a mix of index search methods by using the Prelookup block with the Interpolation Using Prelookup block.

Efficiency of Performance

When the efficiency with which lookup tables operate is important, consider using the Prelookup block with the Interpolation Using Prelookup block. These blocks separate the table lookup process into two components — an index search that relates inputs to the table data, followed by an interpolation and extrapolation stage that computes outputs. These blocks enable you to perform a single index search and then reuse the results to look up data in multiple tables. Also, the Interpolation Using Prelookup block can perform sub-table selection, in which it interpolates a portion of the table data instead of the entire table. For example, if your 3-D table data constitutes a stack of 2-D tables to be interpolated, the block allows you to specify an input used to select and interpolate the 2-D tables. These features make table lookup operations more efficient, reducing computational effort and thus simulation time.

Summary of Lookup Table Block Features

The following table summarizes many features of lookup table blocks in Simulink. Use the table to identify features that correspond to particular lookup table blocks, then select the block that best meets your requirements.

Feature	Lookup Table	Lookup Table (2-D)	Lookup Table Dynamic	Lookup Table (n-D)	Direct Lookup Table (n-D)	Prelookup	Interp. Using Prelookup
Interpolation Methods							
Flat (none)	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline				•			
Extrapolation Methods							
Clipping	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline				•			
Numeric & Data Type Support							
Complex					•		
Double, Single	•	•	•	•	•	•	•
Integer	•	•	•		•	•	•
Fixed-point	•	•	•			•	•
Index Search Methods							
Binary	•	•	•	•		•	
Linear				•		•	
Evenly spaced points				•	•	•	
Start at previous index				•		•	
Miscellaneous							
Sub-table selection					•		•
Dynamic table data			•		•		
Input range checking				•	•	•	•

Entering Breakpoints and Table Data

Simulink provides different ways to enter breakpoints and table data for lookup table blocks. The following sections present examples that demonstrate data entry methods:

- "Entering Data in a Lookup Table Block's Parameter Dialog Box" on page 8-12 — Specify breakpoints using valid MATLAB expressions in the fields of a lookup table block's parameter dialog box. You can access the parameter dialog box by double-clicking a lookup table block in the Simulink model window.
- "Entering Data in the Lookup Table Editor" on page 8-14 — Specify breakpoints using the Lookup Table Editor GUI, accessed from the Simulink **Tools** menu by selecting the **Lookup Table Editor** item. Refer to "Lookup Table Editor" on page 8-26 for more information about this GUI.
- "Entering Data Using the Lookup Table Dynamic Block's Inports" on page 8-16 — Specify breakpoints as signals connected to the inports of the Lookup Table Dynamic block or the Direct Lookup Table (n-D) block. These blocks enable you to alter the breakpoint data set and table data while the simulation executes.

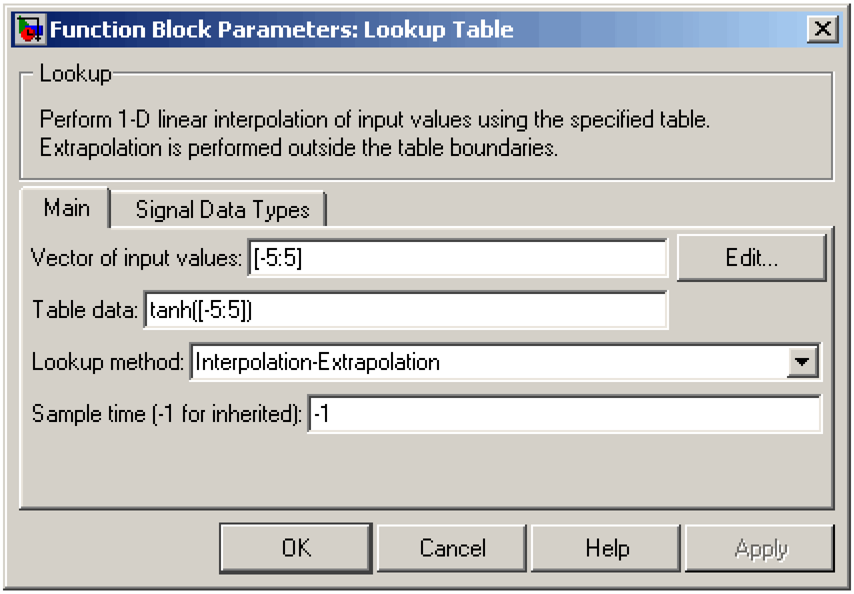
Entering Data in a Lookup Table Block's Parameter Dialog Box

Use the following procedure to populate a Lookup Table block using that block's parameter dialog box. In this example, the lookup table approximates the function $y = x^3$ over the range `[-3, 3]`.

1. Copy a Lookup Table block from the Lookup Tables block library to a Simulink model window.

2. In the Simulink model window, double-click the Lookup Table block to access its parameter dialog box.

The Lookup Table block's parameter dialog box appears.



Lookup

Perform 1-D linear interpolation of input values using the specified table. Extrapolation is performed outside the table boundaries.

Vector of input values: [-5:5]

Table data: tanh([-5:5])

Lookup method: Interpolation-Extrapolation

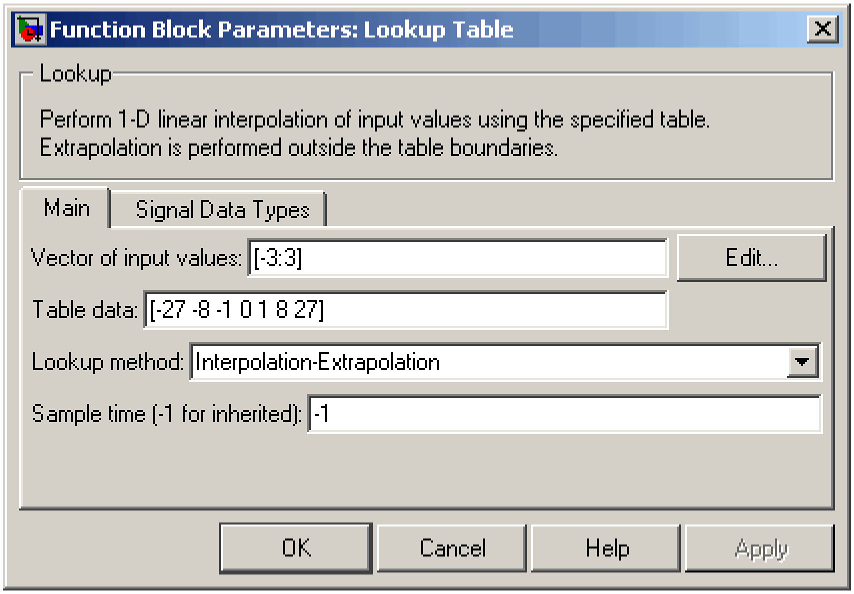
Sample time (-1 for inherited): -1

The dialog box displays the default data associated with the block.

3 Enter the breakpoint data set and table data in the specified fields of the dialog box:

- In the **Vector of input values** field, enter `[-3:3]`.
- In the **Table data** field, enter `[-27 -8 -1 0 1 8 27]`.

The Lookup Table block's dialog box should look similar to the following:



Function Block Parameters: Lookup Table

Lookup

Perform 1-D linear interpolation of input values using the specified table. Extrapolation is performed outside the table boundaries.

Vector of input values: [-3:3]

Table data: [-27 -8 -1 0 1 8 27]

Lookup method: Interpolation-Extrapolation

Sample time (-1 for inherited): -1

4 Click the **Apply** button to apply the changes or the **OK** button to apply the changes and close the dialog box.

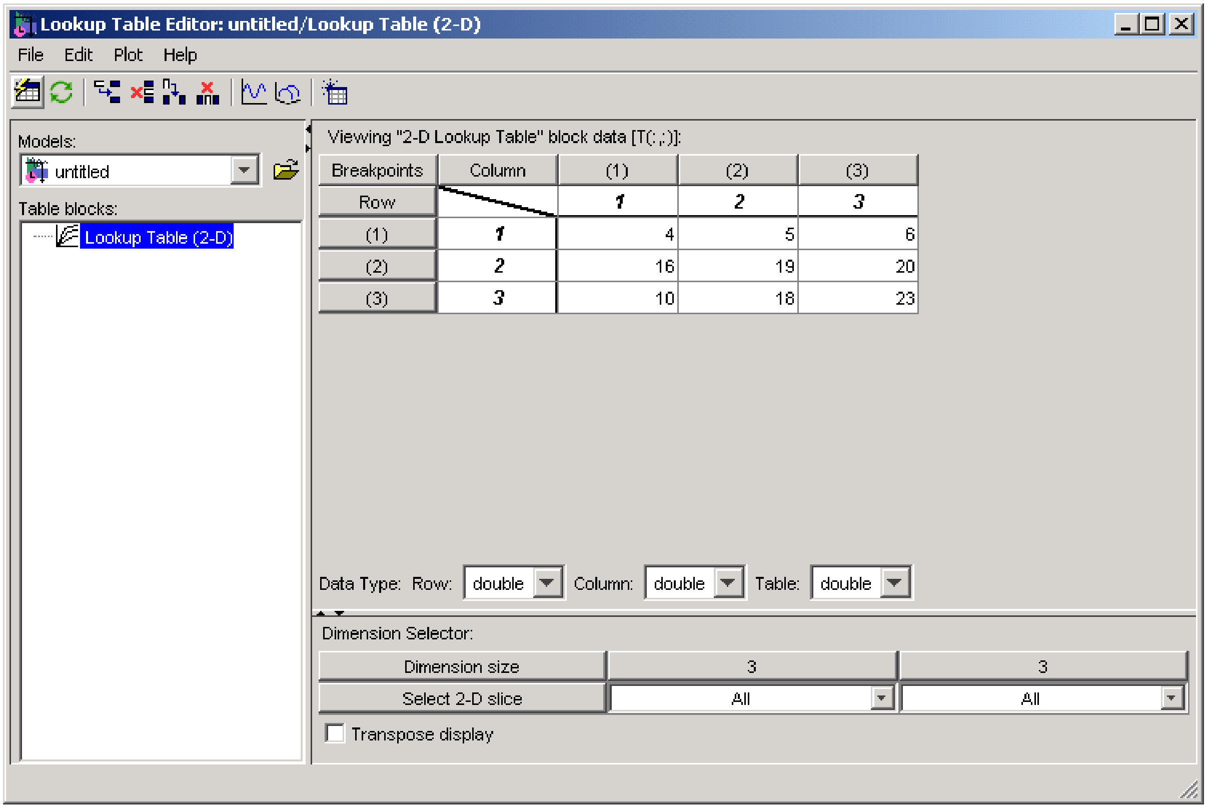
Entering Data in the Lookup Table Editor

Use the following procedure to populate a Lookup Table (2-D) block using the Lookup Table Editor. In this example, the lookup table approximates the function $z = x^2 + y^2$ over the input ranges `x = [0, 2]` and `y = [0, 2]`.

1 Copy a Lookup Table (2-D) block from the Lookup Tables block library to a Simulink model window.

2 Open the Lookup Table Editor by selecting **Lookup Table Editor** from the Simulink **Tools** menu or by clicking the **Edit** button on the parameter dialog box of the Lookup Table (2-D) block.

The Lookup Table Editor appears.



Viewing "2-D Lookup Table" block data [T(:,:)]:

Breakpoints	Column	(1)	(2)	(3)
Row		1	2	3
(1)	1	4	5	6
(2)	2	16	19	20
(3)	3	10	18	23

Data Type: Row: double Column: double Table: double

Dimension Selector:

Dimension size	3	3
Select 2-D slice	All	All

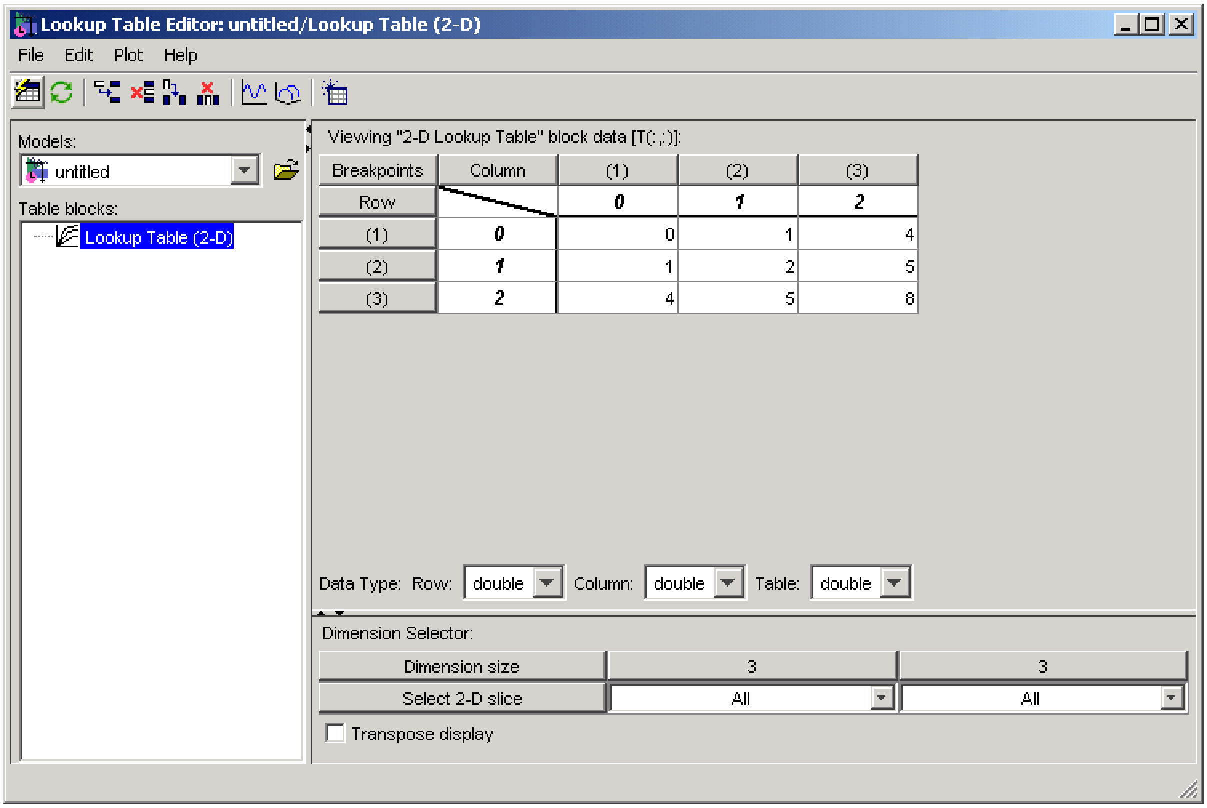
□ Transpose display

It displays the default data associated with the Lookup Table (2-D) block.

3 Under **Viewing "2–D lookup Table" block data**, enter the breakpoint data sets and table data in the appropriate cells. To change the default data, double-click a cell, enter the new value, and then press **Enter** or click outside the field to confirm the change:

- In the cells associated with the **Row Breakpoints**, enter each of the values `[0 1 2]`.
- In the cells associated with the **Column Breakpoints**, enter each of the values `[0 1 2]`.
- In the table data cells, enter the values in the array `[0 1 4; 1 2 5; 4 5 8]`.

The Lookup Table Editor should look similar to the following:



Viewing "2-D Lookup Table" block data [T(:,:)]:

Breakpoints	Column	(1)	(2)	(3)
Row		0	1	2
(1)	0	0	1	4
(2)	1	1	2	5
(3)	2	4	5	8

Data Type: Row: double Column: double Table: double

Dimension Selector:

Dimension size	3	3
Select 2-D slice	All	All

□ Transpose display

4 From the Lookup Table Editor's **File** menu, select **Update Block Data** to update the Lookup Table (2-D) block's data. Close the Lookup Table Editor.

Entering Data Using the Lookup Table Dynamic Block's Inports

Use the following procedure to populate a Lookup Table Dynamic block using that block's inports. In this example, the lookup table approximates the function $y = 3x^2$ over the range `[0, 10]`.

1 Copy a Lookup Table Dynamic block from the Lookup Tables block library to a Simulink model window.

2 Copy the blocks needed to implement the equation $y = 3x^2$ to the Simulink model window:

- A Constant block to define the input range, from the Sources library
- A Math Function block to square the input range, from the Math Operations library
- A Gain block to multiply the signal by 3, also from the Math Operations library

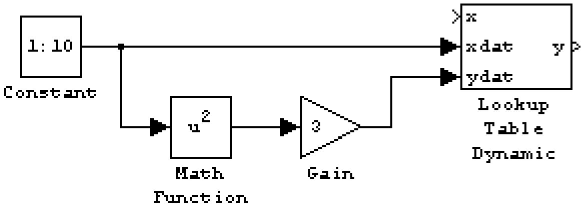
3 Assign the following parameter values to the Constant, Math Function, and Gain blocks using their parameter dialog boxes:

- Specify `1:10` as the Constant block's **Constant value** parameter.
- Specify `square` as the Math Function block's **Function** parameter.
- Specify `3` as the Gain block's **Gain** parameter.

4 Input the breakpoint data set to the Lookup Table Dynamic block by connecting the outport of the Constant block to the inport of the Lookup Table Dynamic block labeled **xdat**. This signal constitutes the input breakpoint data set represented by x.

5 Input the table data to the Lookup Table Dynamic block by branching the output signal from the Constant block and connecting it to the Math Function block. Then connect the Math Function block to the Gain block. Finally, connect the Gain block to the inport of the Lookup Table Dynamic block labeled **ydat**. This signal constitutes the table data represented by y.

The model should look similar to the following:



Characteristics of Lookup Table Data

This section outlines requirements related to the size and monotonicity of a lookup table's breakpoints, and presents a method for representing discontinuities in breakpoint data sets.

- "Sizes of Breakpoint Data Sets and Table Data" on page 8-18
- "Monotonicity of Breakpoint Data Sets" on page 8-19
- "Representing Discontinuities" on page 8-20

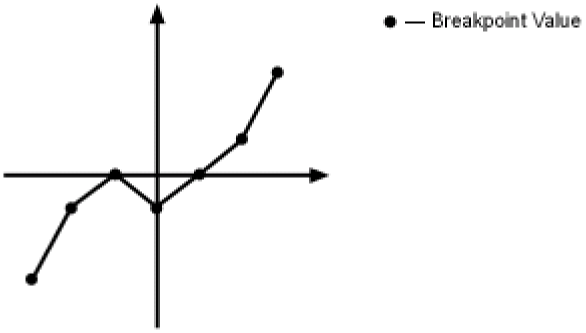
Sizes of Breakpoint Data Sets and Table Data

Simulink imposes the following constraints on the sizes of breakpoint data sets and table data associated with lookup table blocks:

- Your system's memory limitations constrain the overall size of a lookup table.
- Simulink requires correctly dimensioned lookup tables such that the overall size of the table data reflects the size of each breakpoint data set.

To illustrate the second constraint, consider the following vectors of input and output values that create the relationship depicted in the plot:

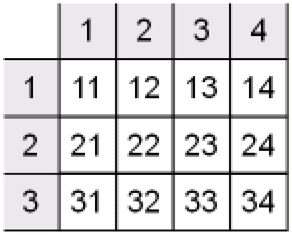
```
Vector of input values:   [-3 -2 -1  0 1 2 3]
Vector of output values:  [-3 -1  0 -1 0 1 3]
```



Here, the input and output data are the same size (1-by-7), making the data consistently dimensioned for a 1-D lookup table.

The following input and output values define the 2-D lookup table that is graphically shown:

```
Row index input values:    [1 2 3]
Column index input values: [1 2 3 4]
Table data:                [11 12 13 14; 21 22 23 24; 31 32 33 34]
```



	1	2	3	4
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34

In this example, the sizes of the vectors representing the row and column indices are 1-by-3 and 1-by-4, respectively. Consequently, the output table must be of size 3-by-4 for consistent dimensioning.

Monotonicity of Breakpoint Data Sets

The first stage of a table lookup operation involves relating inputs to the breakpoint data sets. The search algorithm requires that input breakpoint sets be *monotonically increasing*, that is, each successive element is equal to or greater than its preceding element. For example, the vector

```
A = [0  0.5  1  1.9  2  2  2  2.1  3]
```

repeats the value 2 while all other elements are increasingly larger than their predecessors; hence, A is monotonically increasing.

But for lookup tables populated with data types other than double or single, the search algorithm requires an additional constraint due to quantization effects. In such cases, the input breakpoint data sets must be *strictly monotonically increasing*, i.e., each successive element must be greater than its preceding element. Consider the vector

```
B = [0  0.5  1  1.9  2  2.1  2.17  3]
```

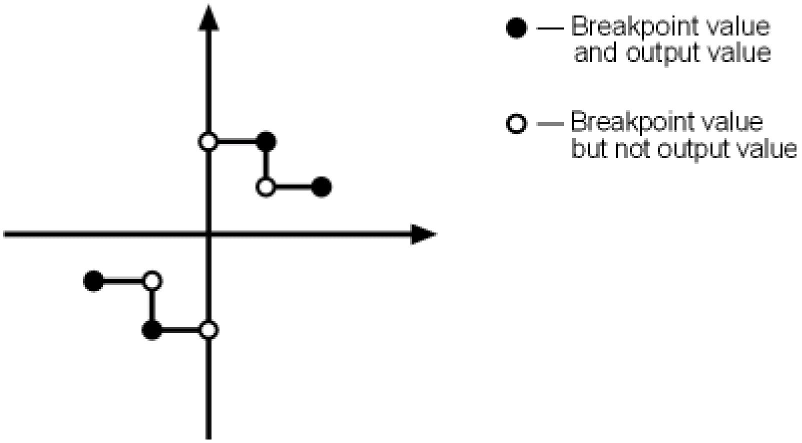
in which each successive element is greater than its preceding element, making B strictly monotonically increasing.

Note that although a breakpoint data set might be strictly monotonic in doubles format, it might not be so after conversion to a fixed-point data type.

Representing Discontinuities

You can represent discontinuities in lookup tables that have monotonically increasing breakpoint data sets. To create a discontinuity, simply repeat an input value in the breakpoint data set with different output values in the table data. For example, these vectors of input (*x*) and output (*y*) values associated with a 1-D lookup table create the step transitions depicted in the plot that follows:

```
Vector of input values:   [-2 -1 -1  0 0 1 1 2]
Vector of output values:  [-1 -1 -2 -2 2 2 1 1]
```



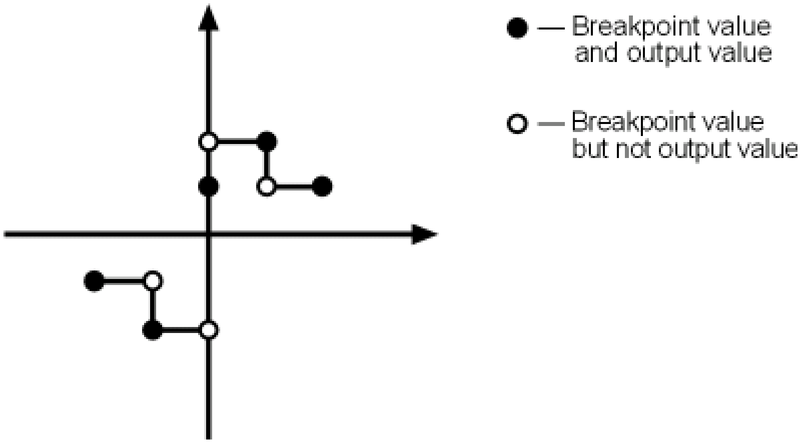
This example has discontinuities at *x* = -1, 0, and +1.

When there are two output values for a given input value, the block chooses the output according to these rules:

- If the input signal is less than zero, the block returns the output value associated with the last occurrence of the input value in the breakpoint data set. In this example, if the input is -1, y is -2, marked with a solid circle.
- If the input signal is greater than zero, the block returns the output value associated with the first occurrence of the input value in the breakpoint data set. In this example, if the input is 1, y is 2, marked with a solid circle.
- If the input signal is zero and there are two output values specified at the origin, the block returns the average of those output values. In this example, if the input is 0, y is 0, the average of the two output values -2 and 2 specified at $x = 0$.

When there are three points specified at the origin, the block generates the output associated with the middle point. The following example demonstrates this special rule:

```
Vector of input values:   [-2 -1 -1  0 0 0 1 1 2]
Vector of output values:  [-1 -1 -2 -2 1 2 2 1 1]
```



In this example, three points define the discontinuity at the origin. When the input is 0, y is 1, the value of the middle point.

You can apply this same method to create discontinuities in breakpoint data sets associated with multidimensional lookup tables.

Estimating Missing Points

The second stage of a table lookup operation involves generating outputs that correspond to the supplied inputs. If the inputs match the values of indices specified in breakpoint data sets, the block outputs the corresponding values. However, if the inputs fail to match index values in the breakpoint data sets, Simulink estimates the output. In the block's parameter dialog box, you can specify how to compute the output in this situation. The available lookup methods are described in the following sections:

- "Interpolation Methods" on page 8-22 — Estimate a value that lies between known data points.
- "Extrapolation Methods" on page 8-23 — Estimate a value that lies beyond the range of known data points.
- "Rounding Methods" on page 8-24 — Approximate a value by altering its digits according to a rule.
- "Example Output" on page 8-25 — Elucidate the differences among the preceding look-up methods.

Interpolation Methods

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. Most lookup table blocks let you select one of the following interpolation methods:

- `None (Flat)` — Disables interpolation and uses the rounding operation titled `Use Input Below`. For more information, see "Rounding Methods" on page 8-24.
- `Linear interpolation` — Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input.
- `Cubic spline interpolation` — Fits a cubic spline to the adjacent breakpoints, and returns the point on that spline corresponding to the input.

Note Blocks such as the Lookup Table Dynamic block do not allow you to choose a particular interpolation method. The `Interpolation-Extrapolation` option in the **Lookup Method** field of its block parameter dialog box performs linear interpolation.

Each of these methods involves a tradeoff between computation time and the smoothness of the result. Although rounding is quickest, it is the least smooth. Linear interpolation is slower than rounding but generates smoother results, except at breakpoints where the slope changes. Cubic spline interpolation is the slowest but produces the smoothest results.

Extrapolation Methods

When an input falls outside the breakpoint data set's range, the block extrapolates the output value from a pair of values at the end of the breakpoint data set. Most lookup table blocks let you select one of the following extrapolation methods:

- `None (Clip to Range)` or `(Use End Values)` — Disables extrapolation and returns the table data corresponding to the end of the breakpoint data set range.
- `Linear extrapolation` — Fits a line between the first or last pair of breakpoints, depending if the input is less than the first or greater than the last breakpoint, respectively. It returns the point on that line corresponding to the input.
- `Cubic spline extrapolation` — Fits a cubic spline to the first or last pair of breakpoints, depending if the input is less than the first or greater than the last breakpoint, respectively. It returns the point on that spline corresponding to the input.

Note Blocks such as the Lookup Table Dynamic block do not allow you to choose a particular extrapolation method. The `Interpolation-Extrapolation` option in the **Lookup Method** field of its block parameter dialog box performs linear extrapolation.

In addition to these methods, some lookup table blocks, such as the Lookup Table (n-D) block, allow you to select an action to perform when encountering situations that require extrapolation. For instance, you can specify that Simulink generate either a warning or an error when the lookup table's inputs are outside the ranges of its breakpoint data sets. To specify such an action, select it from the **Action for out of range input** list on the block's parameter dialog box.

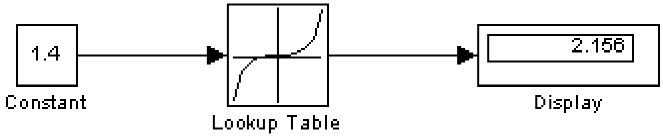
Rounding Methods

If an input falls between breakpoint values or outside the range of a breakpoint data set and you have not specified interpolation or extrapolation, Simulink rounds the value to that of an adjacent breakpoint and returns the corresponding output value. Most lookup table blocks let you select one of the following rounding methods:

- `Use Input Nearest` — Returns the output value corresponding to the nearest input value.
- `Use Input Below` — Returns the output value corresponding to the breakpoint value that is immediately less than the input value. If no breakpoint value exists below the input value, it returns the breakpoint value nearest the input value.
- `Use Input Above` — Returns the output value corresponding to the breakpoint value that is immediately greater than the input value. If no breakpoint value exists above the input value, it returns the breakpoint value nearest the input value.

Example Output

Suppose the Lookup Table block in the following model is configured to use a vector of input values given by `[-5:5]`, and a vector of output values given by `sinh([-5:5])`.



The following outputs are generated when using the specified lookup methods and inputs:

Lookup Method	Input	Output	Comment
`Interpolation-Extrapolation`	`1.4`	`2.156`	N/A
	`5.2`	`83.59`	N/A
`Interpolation-Use End Values`	`1.4`	`2.156`	N/A
	`5.2`	`74.2`	The value for `sinh(5.0)` was used.
`Use Input Above`	`1.4`	`3.627`	The value for `sinh(2.0)` was used.
	`5.2`	`74.2`	The value for `sinh(5.0)` was used.
`Use Input Below`	`1.4`	`1.175`	The value for `sinh(1.0)` was used.
	`-5.2`	`-74.2`	The value for `sinh(-5.0)` was used.
`Use Input Nearest`	`1.4`	`1.175`	The value for `sinh(1.0)` was used.

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, including custom LUT blocks that you have created using the Simulink Mask Editor (see “Editing Custom LUT Blocks” on page 8-33). You can also use a block’s parameter dialog box to edit its table. However, that requires you to open the subsystem containing the block first and then its parameter dialog box. The LUT Editor allows you to skip these steps.

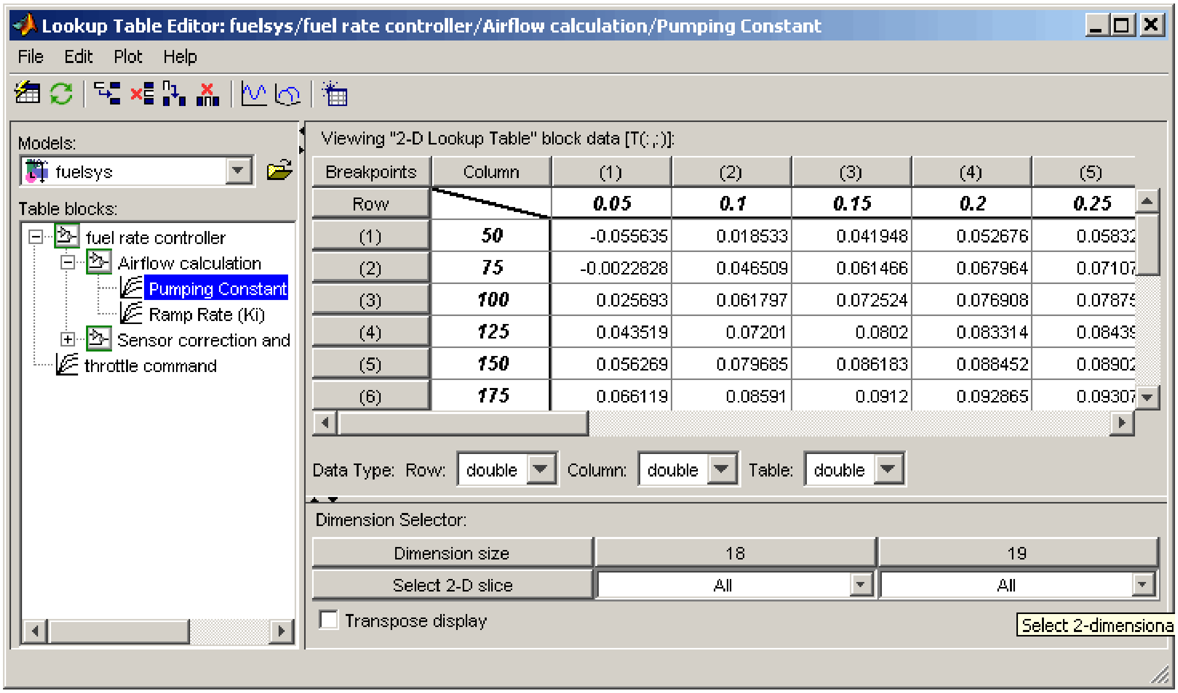
Note You cannot use the LUT Editor to change the dimensions of a lookup table. You must use the block’s parameter dialog box for this purpose.

This section explains how to open and use the LUT Editor to edit LUT blocks.

- “Opening the LUT Editor” on page 8-26
- “Browsing LUT Blocks” on page 8-27
- “Editing Table Values” on page 8-29
- “Displaying N-Dimensional Tables” on page 8-30
- “Plotting LUT Tables” on page 8-32
- “Editing Custom LUT Blocks” on page 8-33

Opening the LUT Editor

To open the editor, select **Lookup Table Editor** from the Simulink **Tools** menu. The editor appears.



Lookup Table Editor: fuelsys/fuel rate controller/Airflow calculation/Pumping Constant

Models: fuelsys

Viewing "2-D Lookup Table" block data [T(:,:)]:

Breakpoints	Column	(1)	(2)	(3)	(4)	(5)
Row		0.05	0.1	0.15	0.2	0.25
(1)	50	-0.055635	0.018533	0.041948	0.052676	0.05832
(2)	75	-0.0022828	0.046509	0.061466	0.067964	0.07107
(3)	100	0.025693	0.061797	0.072524	0.076908	0.07875
(4)	125	0.043519	0.07201	0.0802	0.083314	0.08439
(5)	150	0.056269	0.079685	0.086183	0.088452	0.08902
(6)	175	0.066119	0.08591	0.0912	0.092865	0.09307

Data Type: Row: double Column: double Table: double

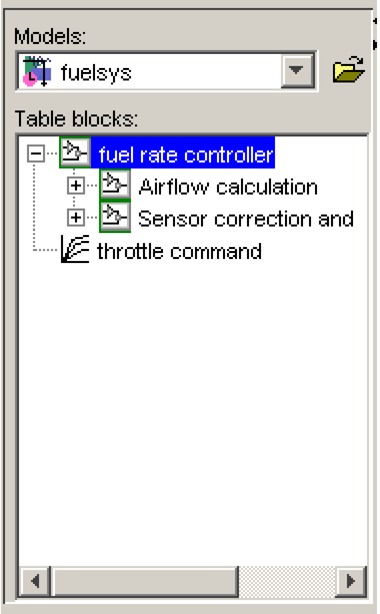
Dimension size	18	19
Select 2-D slice	All	All

☐ Transpose display

The editor contains two panes and a toolbar. 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 8-27). The pane on the right allows you to edit the selected block's lookup table ("Editing Table Values" on page 8-29). The toolbar gives you one-click access to the editor's most frequently used commands. Each toolbar button has a tooltip that explains its function.

Browsing LUT Blocks

The **Models** list in the upper-left corner of the LUT Editor lists the names of all models open in the current MATLAB session. To browse 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.



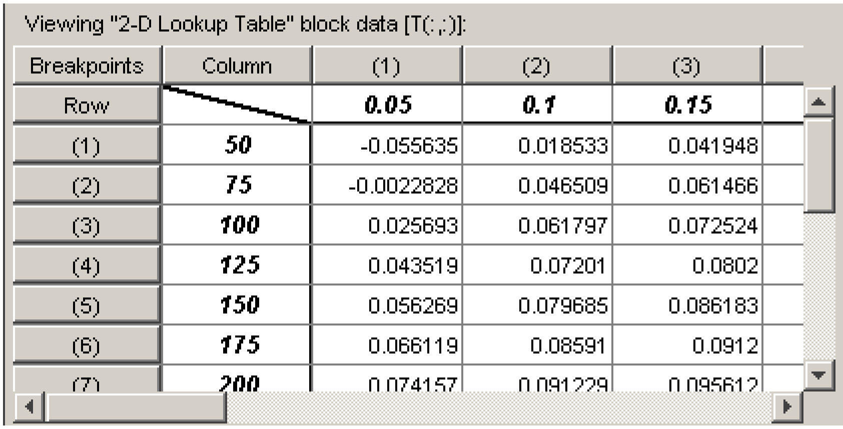
The tree view initially lists all the LUT blocks that reside at the model's root level. It also displays any subsystems that contain LUT blocks. Clicking the expand button (+) to the left of the subsystem's name expands the tree to show the LUT blocks in that subsystem. The expanded view also shows any 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 pane, allowing you to edit the table (see "Editing Table Values" on page 8-29).

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 Model** from the LUT Editor's **File** menu.

Editing Table Values

The **Viewing “2–D lookup Table” block 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.



Viewing "2-D Lookup Table" block data [T(:,:)]:

Breakpoints	Column	(1)	(2)	(3)
Row		***0.05***	***0.1***	***0.15***
(1)	***50***	-0.055635	0.018533	0.041948
(2)	***75***	-0.0022828	0.046509	0.061466
(3)	***100***	0.025693	0.061797	0.072524
(4)	***125***	0.043519	0.07201	0.0802
(5)	***150***	0.056269	0.079685	0.086183
(6)	***175***	0.066119	0.08591	0.0912
(7)	***200***	0.074157	0.091229	0.095612

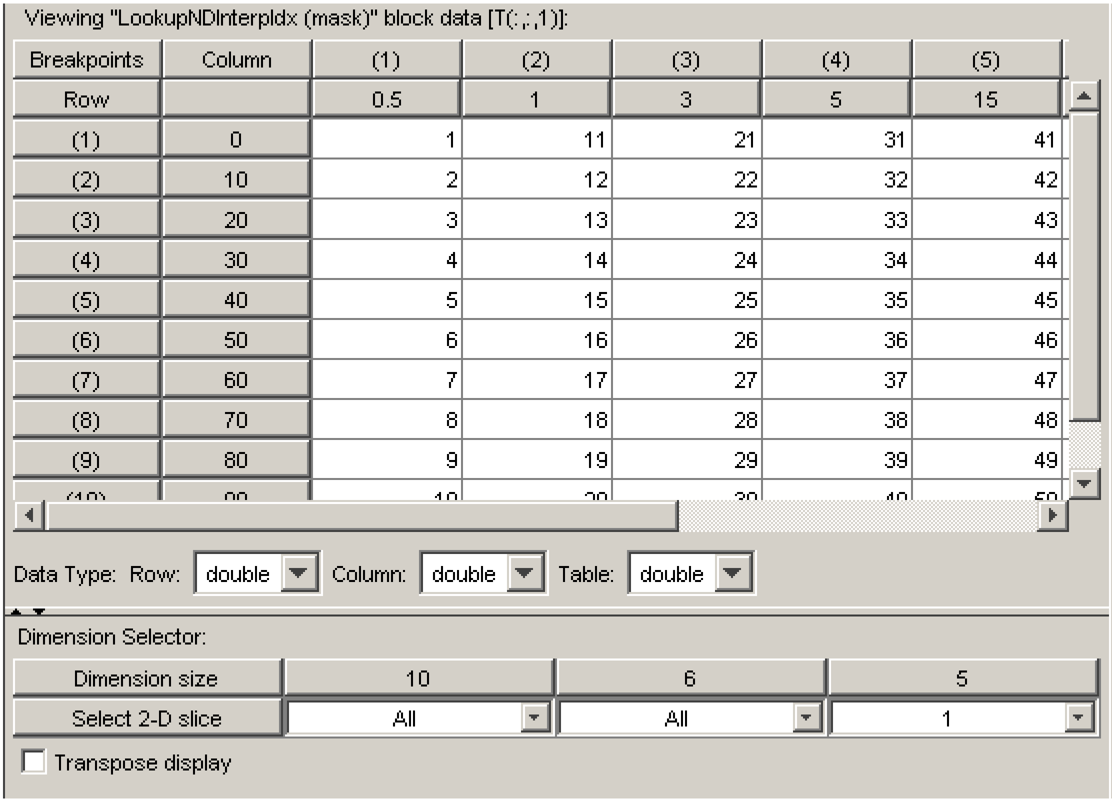
The table view displays the entire table if it is one- or two-dimensional or a two-dimensional slice of the table if the table has more than two dimensions (see “Displaying N-Dimensional Tables” on page 8-30). To change any of the displayed values, double-click the value. The LUT Editor replaces the value with an edit field containing the value. Edit the value, then press **Enter** or click outside the field to confirm the change.

The **Data Type** beneath the table allows you to specify the data type by row or column, or for the entire table. By default, the data type is `double`. To change the data type, select the pop-up index list for the table element for which you want to change the data type.

The LUT Editor records your changes 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-Dimensional 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.



Viewing "LookupNDInterpIdx (mask)" block data [T(:,:,1)]:

Breakpoints	Column	(1)	(2)	(3)	(4)	(5)
Row		0.5	1	3	5	15
(1)	0	1	11	21	31	41
(2)	10	2	12	22	32	42
(3)	20	3	13	23	33	43
(4)	30	4	14	24	34	44
(5)	40	5	15	25	35	45
(6)	50	6	16	26	36	46
(7)	60	7	17	27	37	47
(8)	70	8	18	28	38	48
(9)	80	9	19	29	39	49
(10)	90	10	20	30	40	50

Data Type: Row: double Column: double Table: double

Dimension Selector:

Dimension size	10	6	5
Select 2-D slice	All	All	1

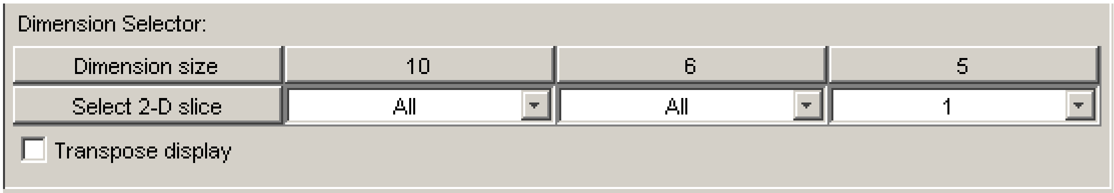
□ Transpose display

The **Dimension Selector** specifies which slice currently appears and allows you to select another slice. The selector consists of a 2-by-N array of controls where N is the number of dimensions in the lookup table. Each column corresponds to a dimension of the lookup table. The third column corresponds to the first dimension of the table, the second column to the second dimension of the table, and so on. The top row of the selector array displays the size of each dimension. The remaining rows specify which dimensions of the table correspond to the row and column axes of the slice and the indices that select the slice from the remaining dimensions.

To select another slice of the table, click the **Select row axis** and **Select column axis** radio buttons in the columns that correspond to the dimensions that you want to view. Then select the indexes of the slice from the pop-up index lists in the remaining columns.

To transpose the table display, click the **Transpose display** check box.

For example, the following selector displays slice (;,;,1) of a 3-D table.



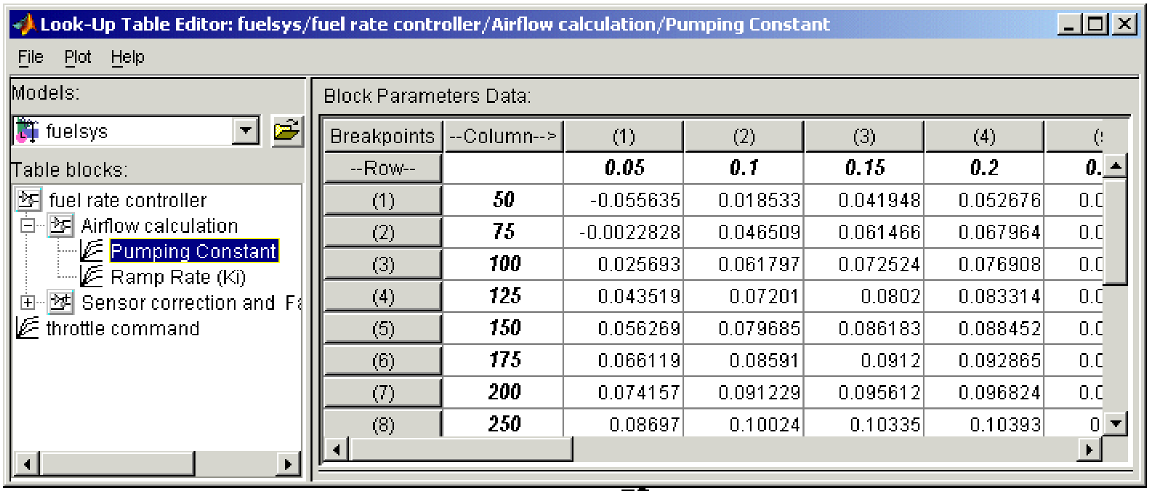
Dimension Selector:

Dimension size	10	6	5
Select 2-D slice	All	All	1

□ Transpose display

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

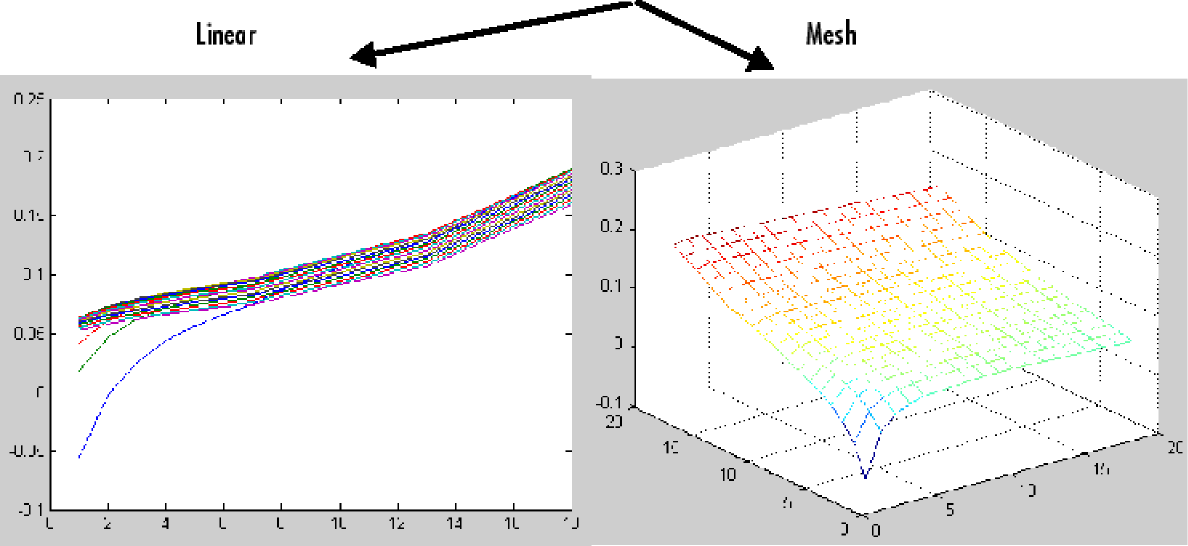
Models: fuelsys

Table blocks:

- fuel rate controller
  - Airflow calculation
    - Pumping Constant
    - Ramp Rate (Ki)
  - Sensor correction and Fa
- throttle command

Block Parameters Data:

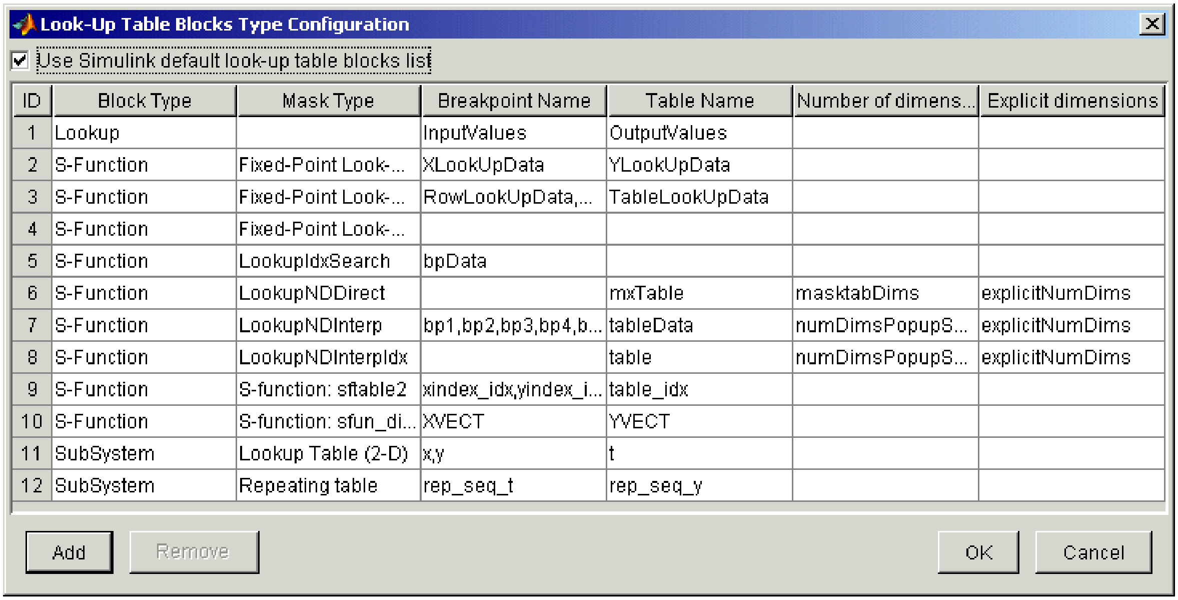
Breakpoints	--Column-->	(1)	(2)	(3)	(4)	(
--Row--		***0.05***	***0.1***	***0.15***	***0.2***	***0.***
(1)	***50***	-0.055635	0.018533	0.041948	0.052676	0.0
(2)	***75***	-0.0022828	0.046509	0.061466	0.067964	0.0
(3)	***100***	0.025693	0.061797	0.072524	0.076908	0.0
(4)	***125***	0.043519	0.07201	0.0802	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.



Look-Up Table Blocks Type Configuration

☑ 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. These include 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 Click **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, select the check box labeled **Use Simulink default look-up table blocks list** at the top of the dialog box.

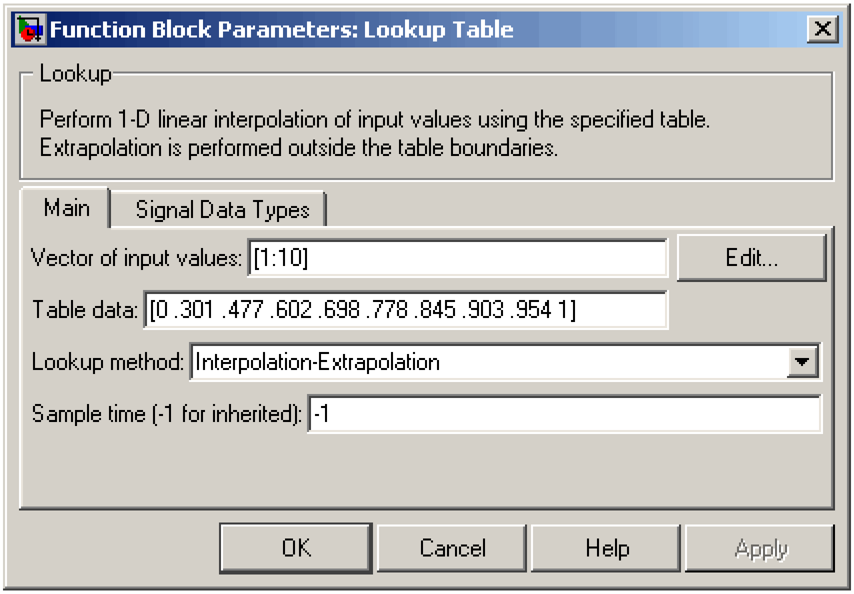
Example of a Logarithm Lookup Table

Suppose you wish to approximate the common logarithm (base 10) over the input range [1, 10] without performing an expensive computation. This can be accomplished using a lookup table block in Simulink, described in the following procedure.

1. Copy the following blocks to the Simulink model window:
   - A Constant block to input the signal, from the Sources library
   - A Lookup Table block to approximate the common logarithm, from the Lookup Tables library
   - A Display block to display the output, from the Sinks library
2. Assign the breakpoint data set and table data to the Lookup Table block by double-clicking it and entering the following values:
   - In the **Vector of input values** field, enter `[1:10]`.
   - In the **Table data** field, enter `[0 .301 .477 .602 .698 .778 .845 .903 .954 1]`.

     Alternatively, you can enter the MATLAB expression `log10(1:10)` in this field, which evaluates to the equivalent vector of output values.

   Note that the **Lookup method** is set to `Interpolation-Extrapolation` by default. The dialog box should now appear as follows:



Function Block Parameters: Lookup Table

Lookup

Perform 1-D linear interpolation of input values using the specified table. Extrapolation is performed outside the table boundaries.

Main | Signal Data Types

Vector of input values: [1:10]

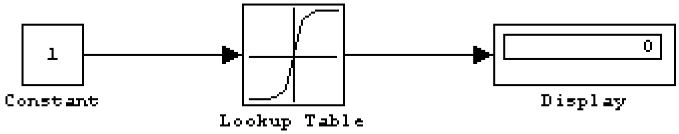
Table data: [0 .301 .477 .602 .698 .778 .845 .903 .954 1]

Lookup method: Interpolation-Extrapolation

Sample time (-1 for inherited): -1

Click the **OK** button to apply the changes and close the dialog box.

3 Connect the blocks such that you input the Constant to the Lookup Table and display its output:



4 Double-click the Constant block to open its parameter dialog box, and change the **Constant value** parameter to, e.g., `5`. Click the **OK** button to apply the changes and close the dialog box.

5 Choose **Start** from the **Simulation** menu to run the simulation.

When the value of the Constant block equals a breakpoint, the Lookup Table block returns the corresponding output value. For instance, when the input value is `5`, the output value is `0.698`.

When the value of the Constant block falls between breakpoints, the Lookup Table block linearly interpolates the output value using neighboring breakpoints. For instance, when the input value is `7.5`, the output value is `0.874`.

When the value of the Constant block falls outside the breakpoint data set's range, the Lookup Table block linearly extrapolates the output value from a pair of values at the end of the breakpoint data set. For instance, when the input value is `10.5`, the output value is `1.023`.

Lookup Table Glossary

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

Term	Meaning
breakpoint	A single element of a breakpoint data set. A breakpoint represents a particular input value to which a corresponding output value in the table data is mapped.
breakpoint data set	A vector of input values that indexes a particular dimension of a lookup table. A lookup table uses breakpoint data sets to relate its input values to the output values that it returns.
extrapolation	A process for estimating values that lie beyond the range of known data points.
interpolation	A process for estimating values that lie between known data points.
lookup table	An array of data that maps input values to output values, thereby approximating a mathematical function. Given the necessary input values, a simple "lookup" operation is used to retrieve the corresponding output values from the table. If the lookup table does not explicitly define the input values, Simulink can estimate an output value using interpolation, extrapolation, or rounding.

Term	Meaning
monotonically increasing	The elements of a set are ordered such that each successive element is greater than or equal to its preceding element.
rounding	A process for approximating a value by altering its digits according to a known rule.
strictly monotonically increasing	The elements of a set are ordered such that each successive element is greater than its preceding element.
table data	An array that serves as a sampled representation of a function evaluated at a lookup table's breakpoint values. A lookup table uses breakpoint data sets to index the table data, ultimately returning an output value.

9

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" on page 9-2
- "Modeling a Continuous System" on page 9-3
- "Choosing Best-Form Mathematical Models" on page 9-5

Converting Celsius to Fahrenheit

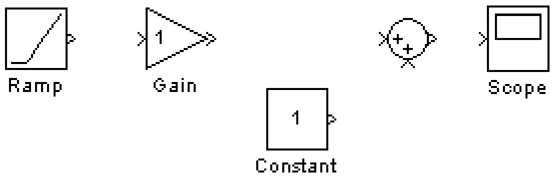
To model the equation that converts Celsius temperature to Fahrenheit

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

First, consider the blocks needed to build the model:

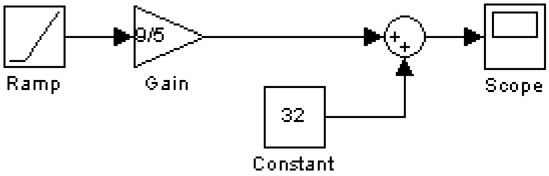
- A Ramp block to input the temperature signal, from the Sources library
- A Constant block to define a constant of 32, also from the Sources library
- A Gain block to multiply the input signal by 9/5, from the Math Operations library
- A Sum block to add the two quantities, also from the Math Operations library
- A Scope block to display the output, from the Sinks library

Next, gather the blocks into your model window.



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

Now, connect the blocks.



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

Open the Scope block to view the output. Now, choose **Start** from the **Simulation** menu to run the simulation. The simulation runs for 10 seconds.

Modeling a Continuous System

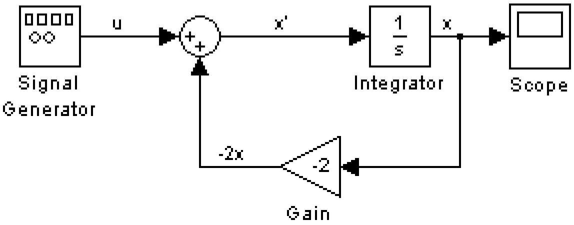
To model the differential equation

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

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.

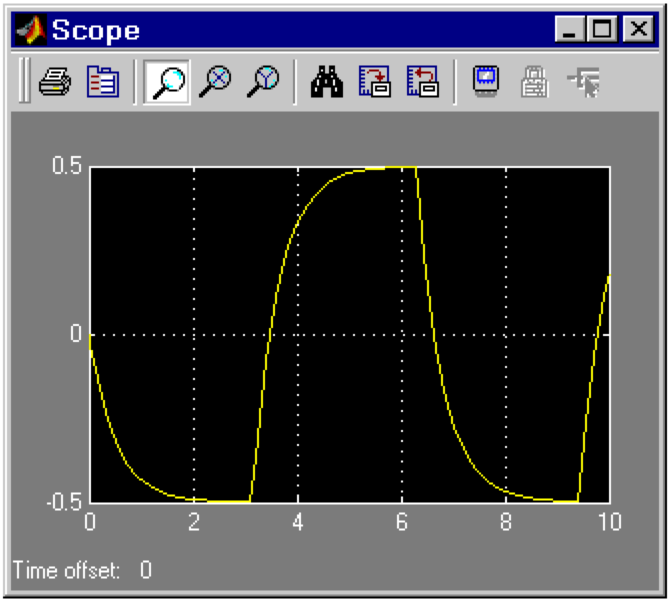
In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see “Drawing a Branch Line” on page 3-17.

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

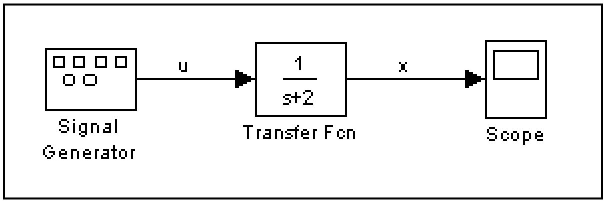
$$x = u/(s+2)$$

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.

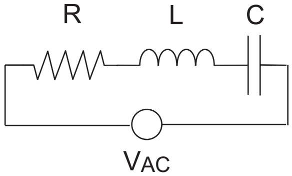
In this case the numerator is [1] (or just 1) and the denominator is [1 2].



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

Choosing Best-Form Mathematical Models

You can often formulate the mathematical system you are modeling in several ways. Choosing the best-form mathematical model allows Simulink to execute faster and more accurately. For example, consider a simple series RLC circuit.



According to Kirkoff’s voltage law, the voltage drop across this circuit is equal to the sum of the voltage drop across each element of the circuit.

$$V_{AC} = V_R + V_L + V_C$$

Using Ohm’s law to solve for the voltage across each element of the circuit, the equation for this circuit can be written as

$$V_{AC} = Ri + L\frac{di}{dt} + \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

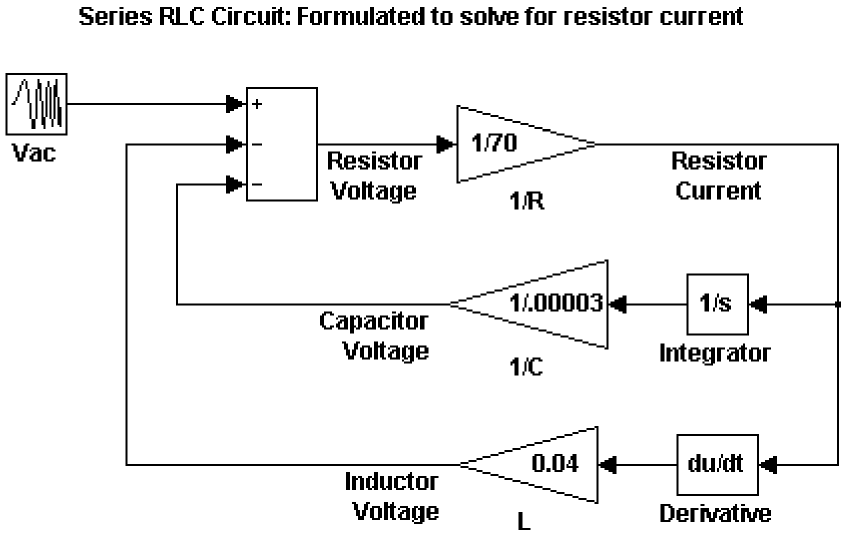
You can model this system in Simulink by solving for either the resistor voltage or inductor voltage. Which you choose to solve for affects the structure of the Simulink model and its performance.

Solving the RLC circuit for the resistor voltage yields

$$V_R = Ri$$

$$Ri = V_{AC} - L\frac{di}{dt} - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

The following diagram shows this equation modeled in Simulink where *R* is `70`, *C* is `0.00003`, and *L* is `0.04`. The resistor voltage is the sum of the voltage source, the capacitor voltage, and the inductor voltage. You need the current in the circuit to calculate the capacitor and inductor voltages. To calculate the current, multiply the resistor voltage by a gain of *1/R*. Calculate the capacitor voltage by integrating the current and multiplying by a gain of *1/C*. Calculate the inductor voltage by taking the derivative of the current and multiplying by a gain of *L*.



This formulation contains a Derivative block associated with the inductor. Whenever possible, you should avoid mathematical formulations that require Derivative blocks as they introduce discontinuities into your system. Simulink uses numerical integration to solve the model dynamics though time. These integration solvers take small steps through time to satisfy an accuracy constraint on the solution. If the discontinuity introduced by the Derivative block is too large, it is not possible for the solver to step across it.

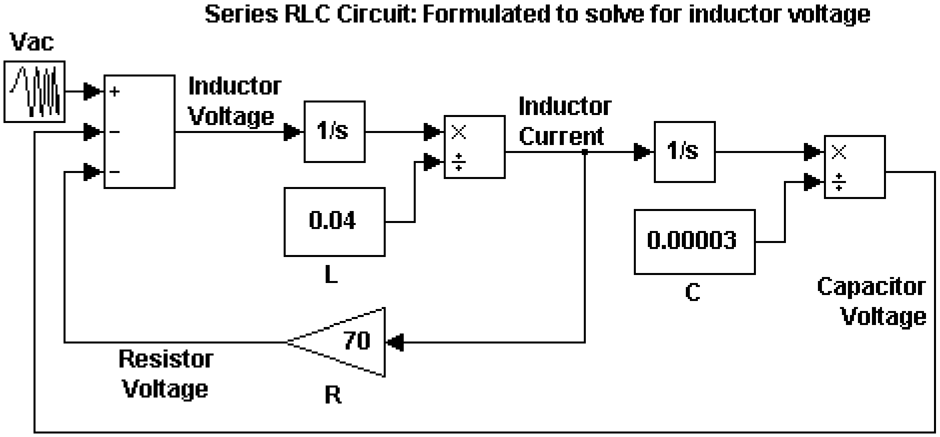
In addition, in this model the Derivative, Sum, and two Gain blocks create an algebraic loop. Algebraic loops slow down the model's execution and can produce less accurate simulation results. See "Algebraic Loops" on page 1-26 for more information.

To avoid using a Derivative block, reformulate the equation to solve for the inductor voltage.

$$V_L = L\frac{di}{dt}$$

$$L\frac{di}{dt} = V_{AC} - Ri - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

The following diagram shows this equation modeled in Simulink. The inductor voltage is the sum of the voltage source, the resistor voltage, and the capacitor voltage. You need the current in the circuit to calculate the resistor and capacitor voltages. To calculate the current, integrate the inductor voltage and divide by *L*. Calculate the capacitor voltage by integrating the current and dividing by *C*. Calculate the resistor voltage by multiplying the current by a gain of *R*.



This model contains only integrator blocks and no algebraic loops. As a result, the model simulates faster and more accurately.

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 diagrammatically (see "Modeling a Continuous System" on page 9-3) 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 (see "Function-Call Subsystems" on page 3-56 for a description of function-call subsystems)
- Self-triggering subsystems and loops containing non-latched triggered subsystems (see "Triggered Subsystems" on page 3-49 in the Using Simulink documentation for a description of triggered subsystems and `Inport` in the Simulink reference documentation for a description of latched input)
- Loops containing action subsystems

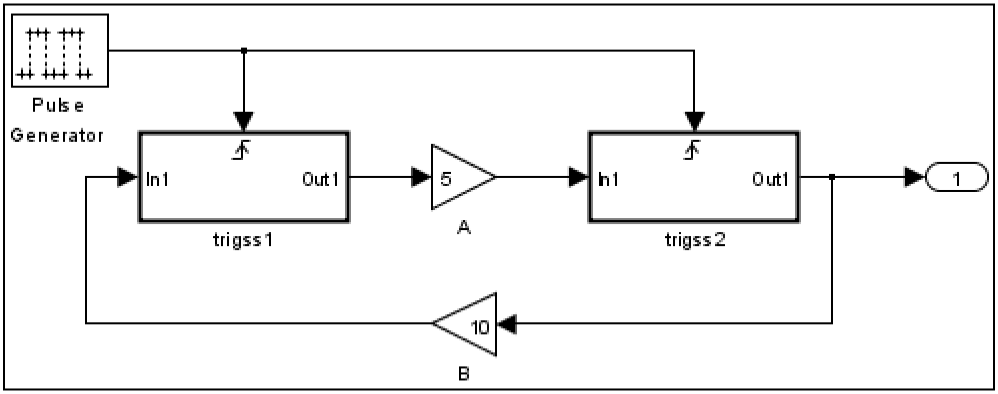
The Subsystem Examples block library in the Ports & Subsystems library contains models that illustrates examples of valid and invalid loops involving triggered and function-call subsystems. Examples of invalid loops include the following models:

- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1` (`sl_subsys_trigerr1`)
- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2` (`sl_subsys_trigerr2`)
- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call systems/sl_subsys_fcncallerr3` (`sl_subsys_fcncallerr3`)

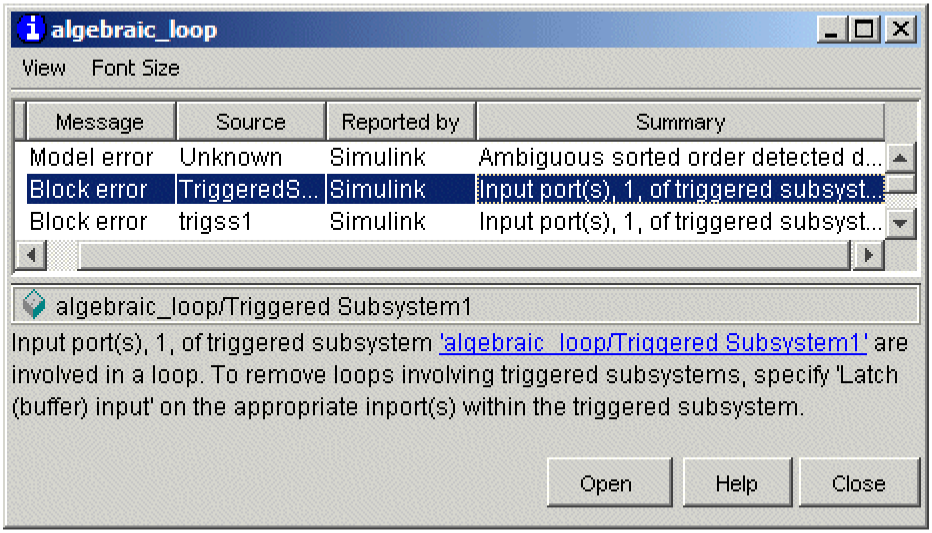
You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update Diagram** from the model's **Edit** menu. If the model contains invalid loops, Simulink highlights the invalid loops as illustrated in the following model .



and displays an error message in the Simulation Diagnostics Viewer.



Message	Source	Reported by	Summary
Model error	Unknown	Simulink	Ambiguous sorted order detected d...
Block error	TriggeredS...	Simulink	Input port(s), 1, of triggered subsyst...
Block error	trigss1	Simulink	Input port(s), 1, of triggered subsyst...

algebraic_loop/Triggered Subsystem1

Input port(s), 1, of triggered subsystem 'algebraic_loop/Triggered Subsystem1' are involved in a loop. To remove loops involving triggered subsystems, specify 'Latch (buffer) input' on the appropriate inport(s) within the triggered subsystem.

Tips for Building Models

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

- Memory issues

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

- Using hierarchy

  More complex models often benefit from adding the hierarchy of subsystems to the model. Grouping blocks simplifies the top level of the model and can make it easier to read and understand the model. For more information, see “Creating Subsystems” on page 3-33. The Model Browser provides useful information about complex models (see “The Model Browser” on page 10-29).

- 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 5-61 and “Annotating Diagrams” on page 3-22.

- Modeling strategies

  If several of your models tend to use the same blocks, you might find it easier to save these blocks in a model. Then, when you build new models, just open this model and copy the commonly used blocks from it. You can create a block library by placing a collection of blocks into a system and saving the system. You can then access the system by typing its name in the MATLAB Command Window.

  Generally, when building a model, design it first on paper, then build it using the computer. Then, when you start putting the blocks together into a model, add the blocks to the model window before adding the lines that connect them. This way, you can reduce how often you need to open block libraries.

Avoiding Problems with Shadowed Files

This section includes the following topics:

- "What Are Shadowed Files?" on page 9-12
- "Making Sure the Correct Block Diagram Is Loaded" on page 9-12
- "Detecting and Fixing Problems" on page 9-14

What Are Shadowed Files?

The rules which Simulink uses to find Model files are similar to those used by MATLAB. See "How the Search Path Determines Which Function to Use" in the MATLAB documentation.

If there are two Model files with the same name (e.g. `mylibrary.mdl`) on the MATLAB path, the one higher on the path is loaded, and the one lower on the path is said to be "shadowed".

However, there is an important difference between the way Simulink handles block diagrams and the way MATLAB handles functions: a loaded block diagram takes precedence over any unloaded ones, regardless of its position on the MATLAB path. This is done for performance reasons, as part of Simulink's incremental loading methodology.

The precedence of a loaded block diagram over any others can have important implications, particularly since a block diagram can be loaded without its Simulink window being visible.

Making Sure the Correct Block Diagram Is Loaded

When using libraries and referenced models, Simulink may load a block diagram without showing its window. If the MATLAB path or the current MATLAB directory changes while block diagrams are in memory, these block diagrams may interfere with the use of other files of the same name. For example, after a change of directory, a loaded but invisible library may be used instead of the one the user expects.

To see an example:

1. Enter sldemo_hydcyl4 to open the Simulink demo model sldemo_hydcyl4.

2. Use the find_system command to see which block diagrams are in memory:

   ```
   find_system('type','block_diagram')

   ans =

       'hydlib'
       'sldemo_hydcyl4'
   ```

   Note that a Simulink library, hydlib, has been loaded, but is currently invisible.

3. Now close sldemo_hydcyl4. Run the find_system command again, and you will see that the library is still loaded.

If you change to another directory which contains a different library called hydlib, and try to run a model in that directory, the library in that directory would not be loaded because the block diagram of the same name in memory takes precedence. This can lead to problems including:

- Simulation errors
- "Bad Link" icons on blocks which are library links
- Wrong results

To prevent these conditions, it is necessary to close the library explicitly as follows:

```
close_system('hydlib')
```

Then, when Simulink next needs to use a block in a library called hydlib it will use the file called hydlib.mdl which is highest on the MATLAB path at the time. Alternatively, to make the library visible, enter:

```
open_system('hydlib')
```

Detecting and Fixing Problems

When updating a block diagram, Simulink checks the position of its file on the MATLAB path and will issue a warning if it detects that another file of the same name exists and is higher on the MATLAB path. The warning reads:

```
The file containing block diagram 'mylibrary' is shadowed
by a file of the same name higher on the MATLAB path.
```

This may indicate that the wrong file called `mylibrary.mdl` is being used. To see which file called `mylibrary.mdl` is loaded into memory, enter:

```
 which mylibrary

C:\work\Model1\mylibrary.mdl
```

To see all the files called `mylibrary` which are on the MATLAB path (note that this can include M-files), enter:

```
which -all mylibrary

C:\work\Model1\mylibrary.mdl
C:\work\Model2\mylibrary.mdl  % Shadowed
```

To close the block diagram called `mylibrary` and let Simulink load the file which is highest on the MATLAB type, enter:

```
close_system('mylibrary')
```

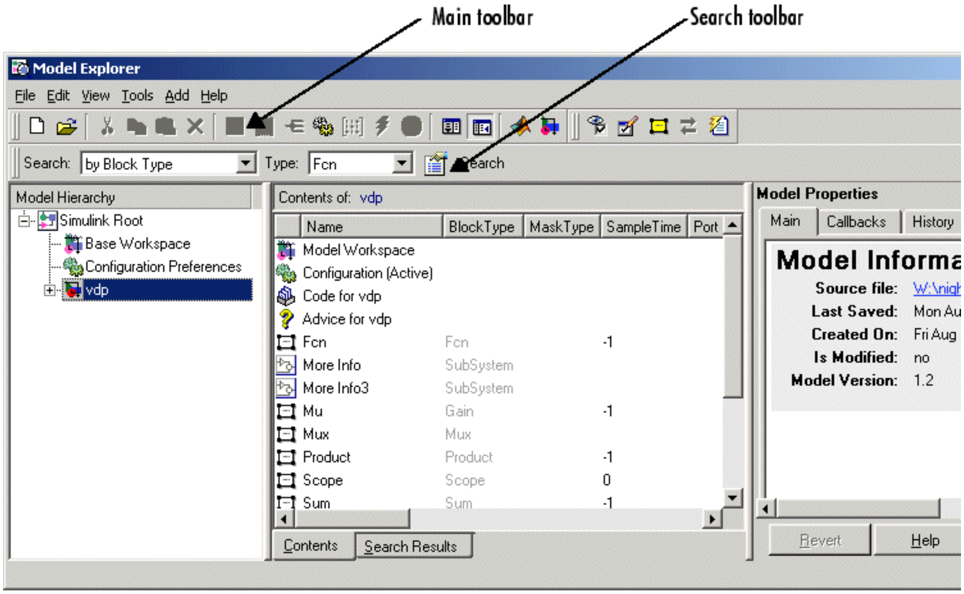
10

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.



Main toolbar

Search toolbar

Search: by Block Type Type: Fcn Search

Model Hierarchy
- Simulink Root
  - Base Workspace
  - Configuration Preferences
  - vdp

Contents of: vdp

Name	BlockType	MaskType	SampleTime	Port
Model Workspace				
Configuration (Active)				
Code for vdp				
Advice for vdp				
Fcn	Fcn		-1	
More Info	SubSystem			
More Info3	SubSystem			
Mu	Gain		-1	
Mux	Mux			
Product	Product		-1	
Scope	Scope		0	
Sum	Sum		-1	

Model Properties

Model Informa

Source file: W:\nigh
Last Saved: Mon Au
Created On: Fri Aug
Is Modified: no
Model Version: 1.2

Model Hierarchy pane Contents pane Dialog pane

The Model Explorer includes the following components:

- **Model Hierarchy** pane (see "Model Hierarchy Pane" on page 10-4)
- **Contents** pane (see "Contents Pane" on page 10-6)
- **Dialog** pane (see "Dialog Pane" on page 10-12)
- **Main** toolbar (see "Main Toolbar" on page 10-14)
- **Search** bar (see "Search Bar" on page 10-16)

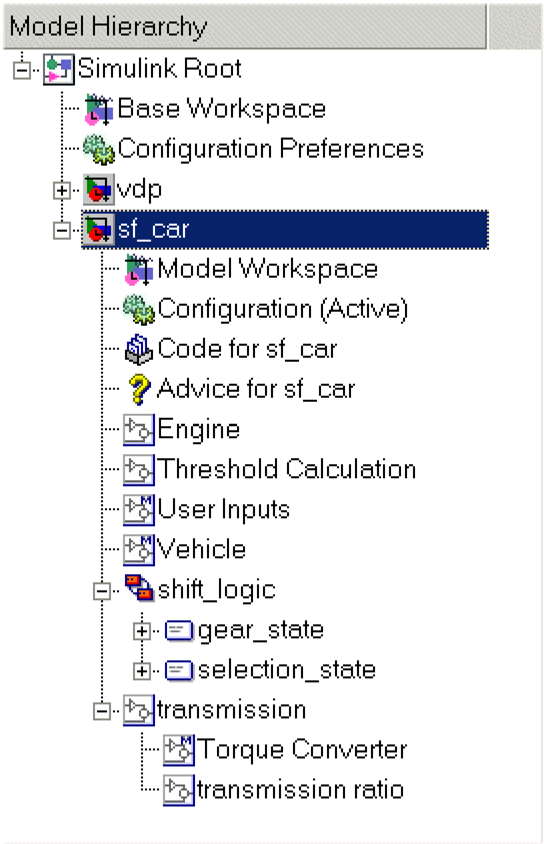
You can use the Model Explorer's **View** menu to hide the **Dialog** pane and the toolbars, thereby making more room for the other panes.

See the following topics for discussion on:

- "Model Hierarchy Pane" on page 10-4
- "Contents Pane" on page 10-6
- "Dialog Pane" on page 10-12
- "Main Toolbar" on page 10-14
- "Search Bar" on page 10-16
- "Setting the Model Explorer's Font Size" on page 10-20

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy.



Model Hierarchy

- Simulink Root
  - Base Workspace
  - Configuration Preferences
  - vdp
  - sf_car
    - Model Workspace
    - Configuration (Active)
    - Code for sf_car
    - Advice for sf_car
    - Engine
    - Threshold Calculation
    - User Inputs
    - Vehicle
    - shift_logic
      - gear_state
      - selection_state
    - transmission
      - Torque Converter
      - transmission ratio

Simulink Root

The first node in the view represents the Simulink root. Expanding the root node displays nodes representing the MATLAB workspace (the Simulink base workspace) and each model and library loaded in the current session.

Base Workspace

This node represents the MATLAB workspace. The MATLAB workspace is the base workspace for Simulink models. Variables defined in this workspace are visible to all open Simulink models, i.e., to all models whose nodes appear beneath the **Base Workspace** node in the **Model Hierarchy** pane.

Configuration Preferences

If you check the **Show Configuration Preferences** option on the Model Explorer's **View** menu, the expanded Simulink Root node also displays a Configuration Preferences node. Selecting this node displays the preferred model configuration (see "Configuration Sets" on page 11-37) for new models in the adjacent panes. You can change the preferred configuration by editing the displayed settings and using the **Model Configuration Preferences** dialog box to save the settings (see "Model Configuration Preferences Dialog" on page 11-46).

Model Nodes

Expanding a model node displays nodes representing the model's configuration sets (see "Configuration Sets" on page 11-37), 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

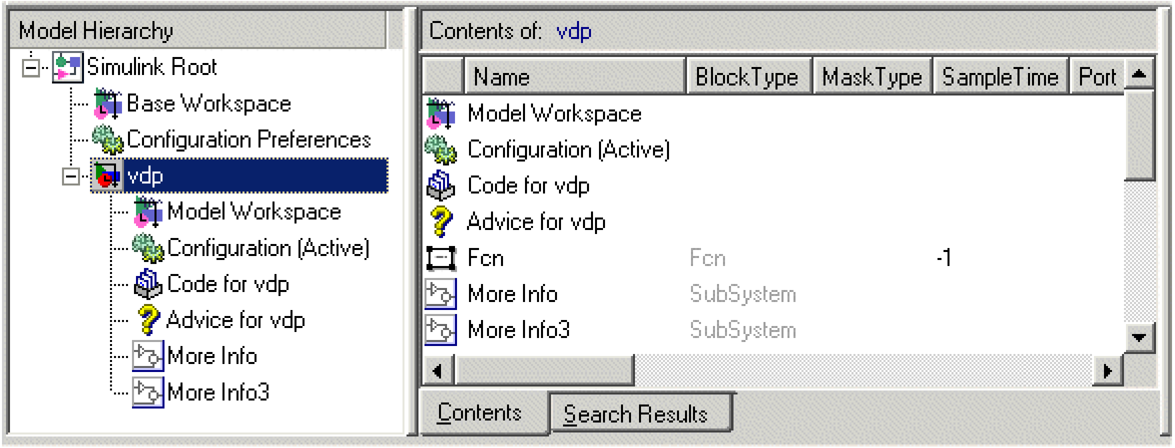
To display the contents of an object displayed in the **Model Hierarchy** pane (e.g., a model or configuration set) in the adjacent **Contents** pane, select the object. To open a graphical object (e.g., a model, subsystem, or chart) in an editor window, right-click the object. A context menu appears. 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 11-37 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 3-63), 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 either of two tabular views selectable by tabs. The **Contents** tab displays the contents of the object selected in the **Model Hierarchy** pane. The **Search Results** tab displays the results of a search operation (see “Search Bar” on page 10-16) .



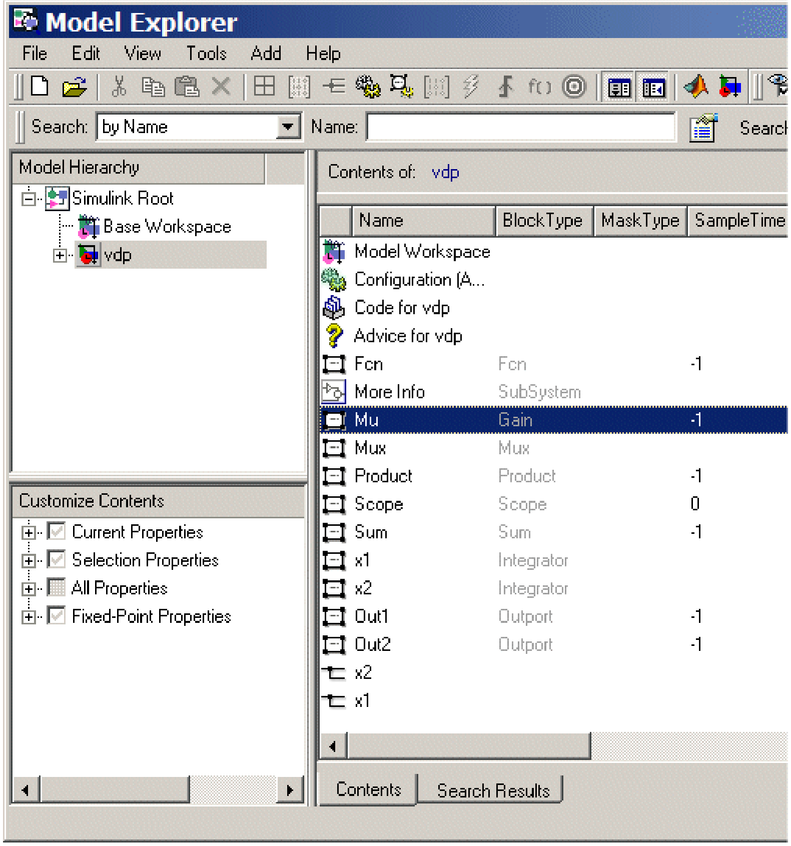
Name	BlockType	MaskType	SampleTime	Port
Model Workspace				
Configuration (Active)				
Code for vdp				
Advice for vdp				
Fcn	Fcn		-1	
More Info	SubSystem			
More Info3	SubSystem			

In both views, the table rows correspond to objects (e.g., blocks or states); the table columns, to object properties (e.g., name and type). The table cells display the values of the properties of the objects contained by the object selected in the **Model Hierarchy** pane or found by a search operation.

The objects and properties displayed in the **Contents** pane depend on the type of object (e.g., subsystem, chart, or configuration set) selected in the **Model Hierarchy** pane. For example, if the object selected in the **Model Hierarchy** pane is a model or subsystem, the **Contents** pane by default displays the name and type of the top-level blocks contained by that model or subsystem. If the selected object is a Stateflow chart or state, the **Contents** pane by default shows the name, scope, and other properties of the events and data that make up the chart or state.

Customize Contents Pane

The **Customize Contents** pane allows you to select the properties that the **Contents** pane displays for the object selected in the **Model Hierarchy** pane. When visible, the pane appears in the lower-left corner of the Model Explorer window.



Contents of: vdp

Name	BlockType	MaskType	SampleTime
Model Workspace			
Configuration (A...			
Code for vdp			
Advice for vdp			
Fcn	Fcn		-1
More Info	SubSystem		
Mu	Gain		-1
Mux	Mux		
Product	Product		-1
Scope	Scope		0
Sum	Sum		-1
x1	Integrator		
x2	Integrator		
Out1	Outport		-1
Out2	Outport		-1
x2			
x1			

A splitter divides the **Customize Contents** pane from the **Model Hierarchy** pane above it. Drag the splitter up or down to adjust the relative size of the two panes.

The **Customize Contents** pane contains a tree-structured property list. The list's top-level nodes group object properties into the following categories:

- `Current Properties`

  Properties that the **Contents** pane currently displays.

- `Selection Properties`

  Properties of the object currently selected in the **Contents** pane.

- `All Properties`

  Properties of the contents of all models displayed in the Model Explorer thus far in this session.

- `Fixed Point Properties`

  Fixed-point properties of blocks.

By default, the **Contents** pane displays a standard subset of properties for the currently selected model. The **Customize Contents** pane allows you to perform the following customizations:

- To display additional properties of the selected model, expand the `All Properties` node, if necessary, and check the desired properties.
- To delete some but not all properties from the **Contents** pane, expand the `Current Properties` node, if necessary, and uncheck the properties that you do not want to appear in the **Contents** pane.
- To delete all properties from the **Contents** pane (except the selected object's name), uncheck `Current Properties`.
- To display only the properties of the currently selected object, uncheck `Current Properties` to clear the properties display, then check `Selection Properties`.
- To add or remove fixed-point block properties from the **Contents** pane, check or uncheck `Fixed Point Properties`.

Customizing the Contents Pane

The Model Explorer's **View** menu allows you to control the type of objects and properties displayed in the **Contents** pane.

- To display only object names in the **Contents** pane, uncheck the **Show Properties** item on the **View** menu.

- To customize the set of properties displayed in the **Contents** pane, select **Customize Contents** from the **View** menu or click the **Customize Contents** button on the Model Explorer's main toolbar (see "Main Toolbar" on page 10-14). The **Customize Contents** pane appears. Use the pane to select the properties you want the **Contents** pane to display.
- To specify the types of subsystem or chart contents displayed in the **Contents** pane, select **List View Options** from the **View** menu. A menu of object types appears. Check the types that you want to be displayed (e.g., **Blocks** and **Named Signals/Connections** or **All Simulink Objects** for models and subsystems).

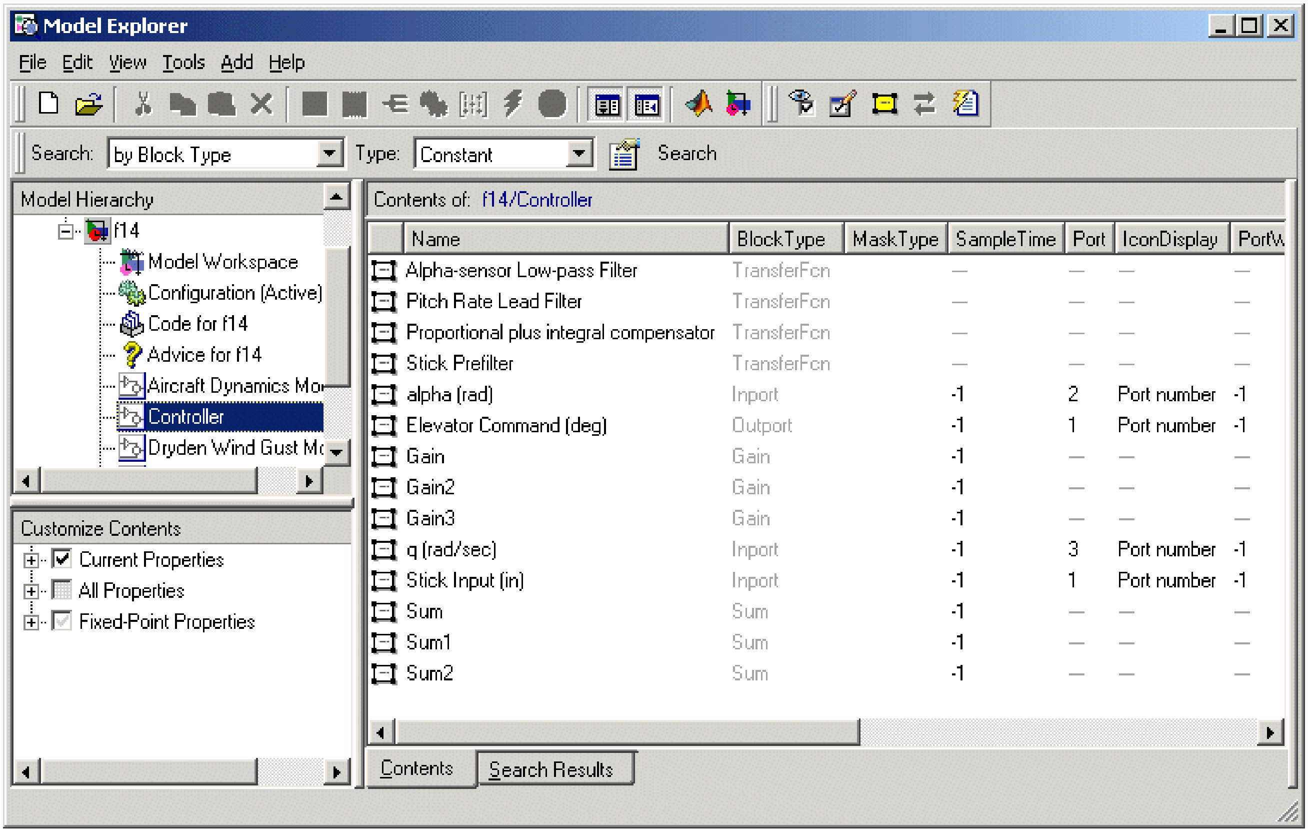
Reordering the Contents Pane

The **Contents** pane by default displays its contents in ascending order by name. To order the contents in ascending order by any other displayed property, click the head of the column that displays the property. To change the order from ascending to descending, or vice versa, click the head of the property column that determines the current order.

Marking Nonexistent Properties

Some of the properties that the Contents pane is configured to display may not apply to all the objects currently listed in the **Contents** pane. You can configure the Model Explorer to indicate the inapplicable properties.

To do this, select **Mark Nonexistent Properties** from the Model Explorer's **View** menu. The Model Explorer now displays dashes for the values of properties that do not apply to the objects displayed in the **Contents** pane.



Search: by Block Type Type: Constant Search

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.

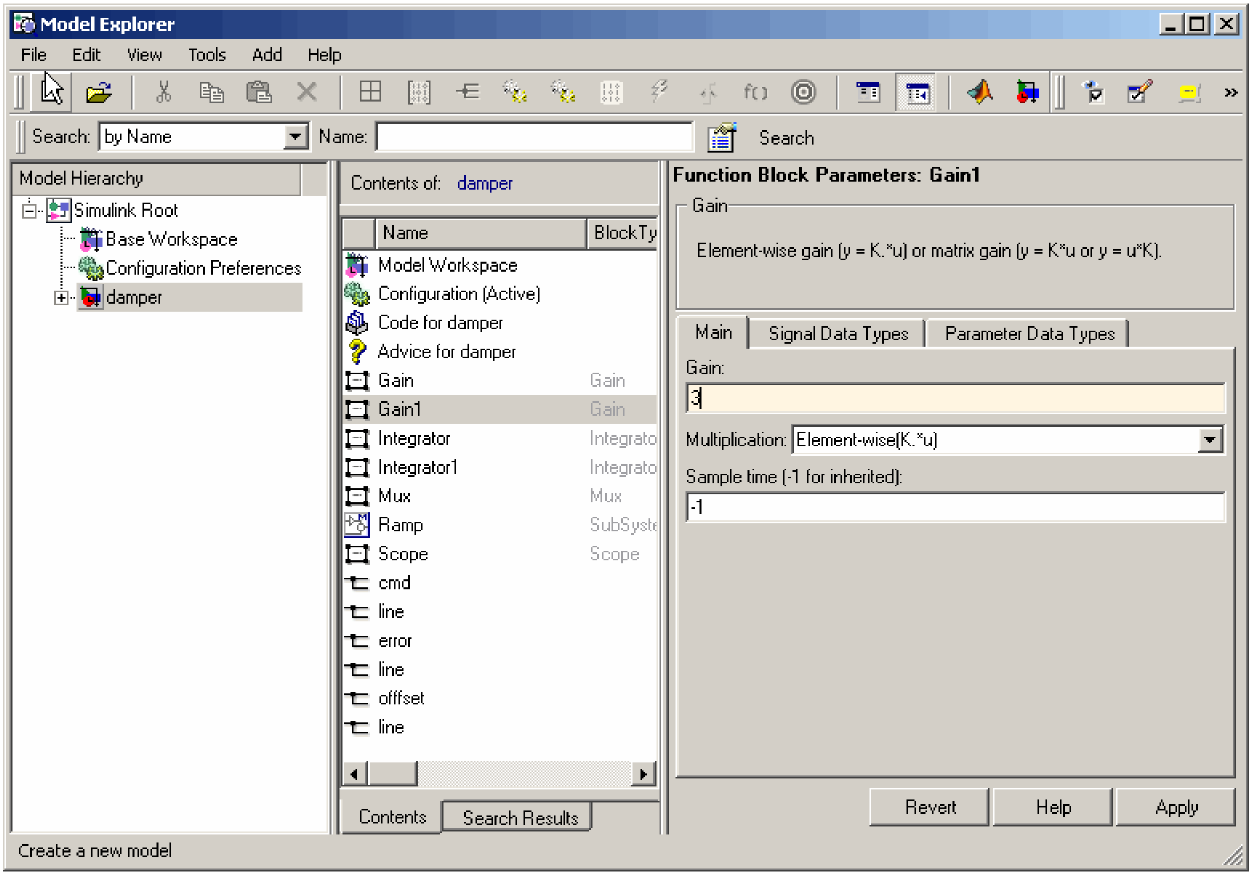
To assign the same property value to multiple objects displayed in the **Contents** pane, select the objects and then change one of the selected objects to have the new property value. The Model Explorer assigns the new property value to the other selected objects as well.

Dialog Pane

Use the **Dialog** pane to view and change properties of the blocks or signals in your model.

1. To show the **Dialog** pane, select **Dialog View** from the **View** menu, located on the Model Explorer's main toolbar.

2. In the **Contents** pane, select an object (such as a block or signal). The properties are displayed in the **Dialog** pane.



Name	BlockTy
Model Workspace	
Configuration (Active)	
Code for damper	
Advice for damper	
Gain	Gain
Gain1	Gain
Integrator	Integrato
Integrator1	Integrato
Mux	Mux
Ramp	SubSyste
Scope	Scope
cmd	
line	
error	
line	
offfset	
line	

Function Block Parameters: Gain1

Gain

Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).

Gain: 3

Multiplication: Element-wise(K.*u)

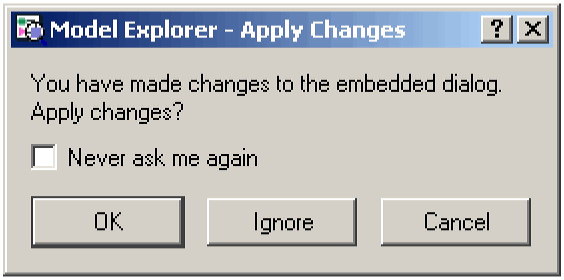
Sample time (-1 for inherited): -1

Create a new model

3 Change a property (for example, the gain of a gain block) in the **Dialog** pane.

4 Click **Apply** to accept the change, or click **Revert** to return to the original value.

Clicking outside a dialog with unapplied changes causes the **Model Explorer – Apply Changes** dialog box to appear.



You have made changes to the embedded dialog. Apply changes?

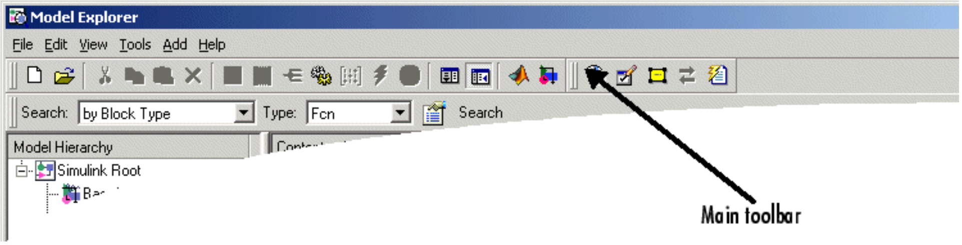
□ Never ask me again

Click **OK** to accept the changes. Click **Ignore** to revert to the original settings. Click **Cancel** to dismiss the dialog box without making any changes.

To suppress appearance of this dialog box in the future, you can check the **Never ask me again** box on this instance of the dialog box or check **Auto Apply/Ignore Dialog Changes** on the Model Explorer **Tools** menu.

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.

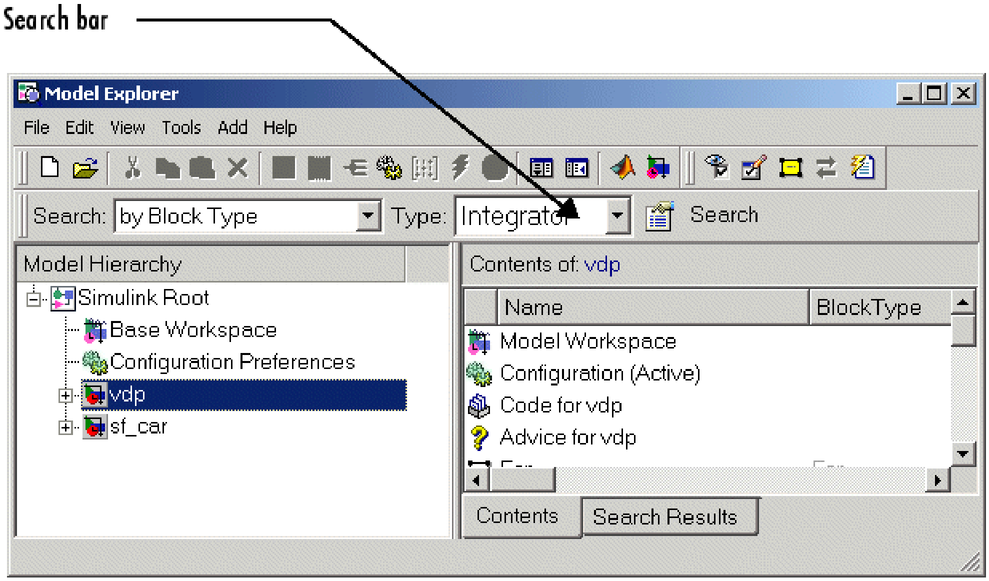
Button	Usage
✂	Cut the objects (e.g., variables) selected in the **Contents** pane from the object (e.g., a workspace) selected in the **Model Hierarchy** pane. Save a copy of the object on the system clipboard.
⎘	Copy the objects selected in the **Contents** pane to the system clipboard.
📋	Paste objects from the clipboard into the object selected in the Model Explorer's **Model Hierarchy** pane.
✕	Delete the objects selected in the **Contents** pane from the object selected in the **Model Hierarchy** pane.
▦	Add a MATLAB variable to the workspace selected in the **Model Hierarchy** pane.
▨	Add a `Simulink.Parameter` object to the workspace selected in the **Model Hierarchy** pane.
⊣	Add a `Simulink.Signal` object to the workspace selected in the **Model Hierarchy** pane.
⚙	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.

Button	Usage
	Customize the Model Explorer’s **Contents** pane.
	Bring the MATLAB desktop to the front.
	Display the Simulink Library Browser.

To show or hide the main toolbar, select **Main Toolbar** from the Model Explorer’s **View** menu.

Search Bar

The Model Explorer’s search bar allows you to select, configure, and initiate searches of the object selected in the **Model Hierarchy** pane. It appears at the top of the Model Explorer window.



Search: by Block Type
Type: Integrator

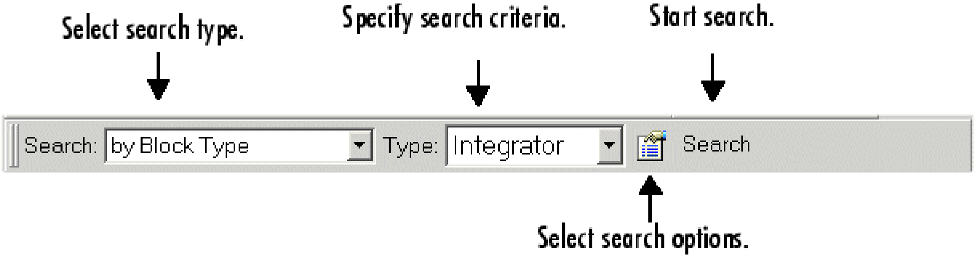
Simulink Root
Base Workspace
Configuration Preferences
vdp
sf_car

Contents of: vdp

Name	BlockType
Model Workspace	
Configuration (Active)	
Code for vdp	
Advice for vdp	

To show or hide the search bar, check or uncheck **Search Bar** in the Model Explorer's **View > Toolbars** menu.

The search bar includes the following controls:



Search Type

Specifies the type of search to be performed. Options include:

- `by Block Type`

  Search for blocks of a specified block type. Selecting this search type causes the search bar to display a block type list control that allows you to select the target block type from the types contained by the currently selected model.

- `by Property Name`

  Searches for objects that have a specified property. Selecting this search type causes the search bar to display a control that allows you to specify the target property's name by selecting from a list of properties that objects in the search domain can have.

- `by Property Value`

  Searches for objects whose property matches a specified value. Selecting this search type causes the search bar to display controls that allow you to specify the name of the property, the value to be matched, and the type of match (equals, less than, greater than, etc.).

- `for Fixed Point`

  Searches a model for all blocks that support fixed-point computations.

- `by Name`

Searches a model for all objects that have the specified string in the name of the object.

- `by Stateflow Type`

  Searches for Stateflow objects of a specified type.

- `for Library Links`

  Searches for library links in the current model.

- `by Class`

  Searches for Simulink objects of a specified class.

- `for Model References`

  Searches a model for references to other models.

- `by Dialog Prompt`

  Searches a model for all objects whose dialogs contain a specified prompt.

- `by String`

  Searches a model for all objects in which a specified string occurs.

Search Options

Specifies options that apply to the current search. The options include:

- `Search Current System and Below`

  Search the current system and the subsystems that it includes directly or indirectly.

- `Look Inside Masked Subsystems`

  Search includes masked subsystems.

- `Look Inside Linked Subsystems`

  Search includes linked subsystems.

- `Match Whole String`

  Do not allow partial string matches, e.g., do not allow `sub` to match `substring`.

- `Match Case`

Consider case when matching strings, e.g., `Gain` does not match `gain`.

- `Regular Expression`

  The Model Explorer considers a string to be matched as a regular expression.

- `Evaluate Property Values During Search`

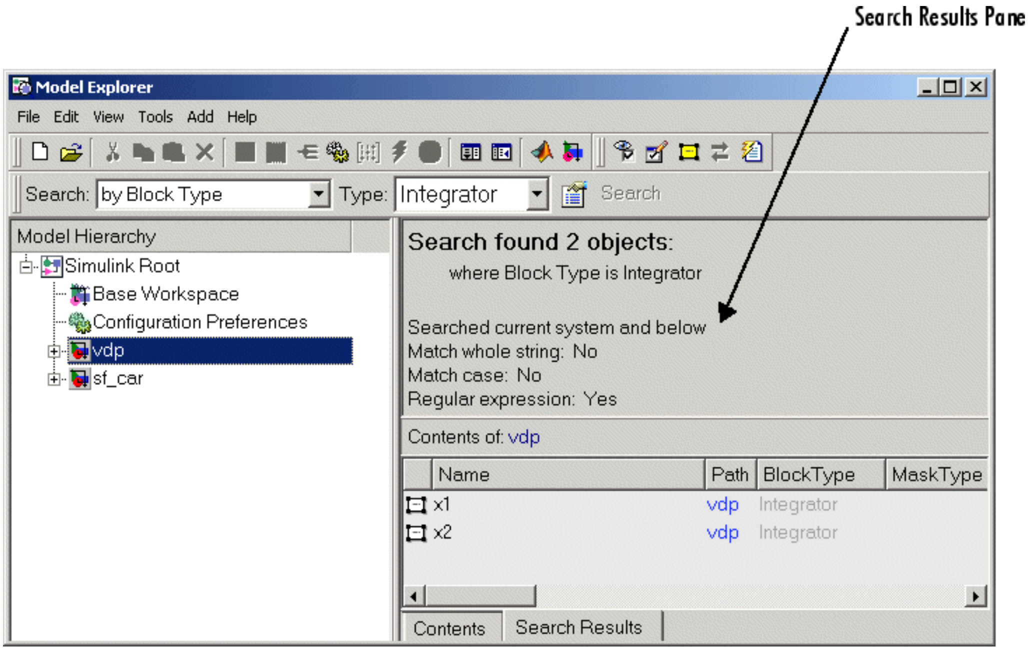
  This option applies only for searches by property value. If enabled, the option causes the Model Explorer to evaluate the value of each property as a MATLAB expression and compare the result to the search value. If disabled (the default), the Model Explorer compares the unevaluated property value to the search value.

- `Refine Search`

  Causes the next search operation to search for objects that meet both the original and new search criteria (see "Refining a Search" on page 10-20).

Search Button

Initiates the search specified by the current settings of the search bar on the object selected in the Model Explorer's **Model Hierarchy** pane. The Model Explorer displays the results of the search in the tabbed **Search Result**s pane.


Search Results Pane

Search: by Block Type Type: Integrator

Search found 2 objects:
where Block Type is Integrator

Searched current system and below
Match whole string: No
Match case: No
Regular expression: Yes

Contents of: vdp

Name	Path	BlockType	MaskType
x1	vdp	Integrator	
x2	vdp	Integrator	

You can edit the results displayed in the **Search Results** pane. For example, to change all objects found by a search to have the same property value, select the objects in the **Search Results** pane and change one of them to have the new property value.

Refining a Search

To refine the previous search, check the **Refine Search** option on the search bar's **Search Options** menu. A **Refine** button replaces the **Search** button on the search bar. Use the search bar to define new search criteria and then click the **Refine** button. The Model Explorer searches for objects that match the previous search criteria and the new criteria.

Setting the Model Explorer's Font Size

You use Model Explorer to change the font size used in the Simulink dialog boxes and in Model Explorer.

To change the font size used in Model Explorer and in the Simulink Dialog boxes, first open Model Explorer (see “The Model Explorer” on page 10-2).

- Press the **Ctrl+** keys to increase the font size

  Alternatively, from the Model Explorer’s **View** menu, select **Increase Font Size**

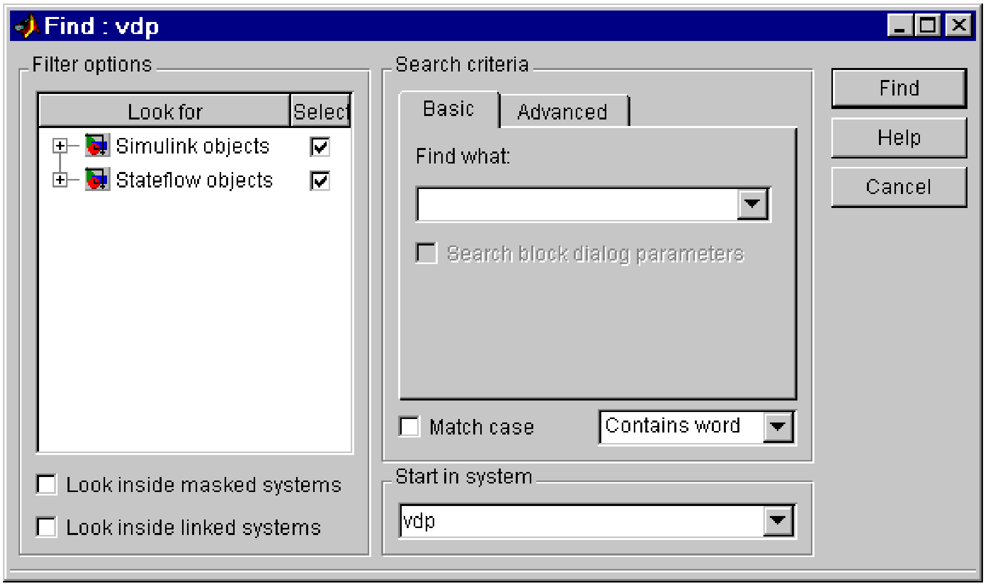
- Press the **Ctrl-** keys to decrease the font size

  Alternatively, from the Model Explorer’s View menu, select **Decrease Font Size**

Note These changes simultaneously alter the font size used by Model Explorer and in the Simulink dialog boxes. The changes remain in effect across Simulink sessions.

The Finder

The Finder locates blocks, signals, states, or other objects in a model. To display the Finder, select **Find** from the Simulink Model Editor's **Edit** menu. The **Find** dialog box appears.


Find : vdp

Filter options

Look for	Select
Simulink objects	☑
Stateflow objects	☑

□ Look inside masked systems

□ Look inside linked systems

Search criteria

Basic | Advanced

Find what:

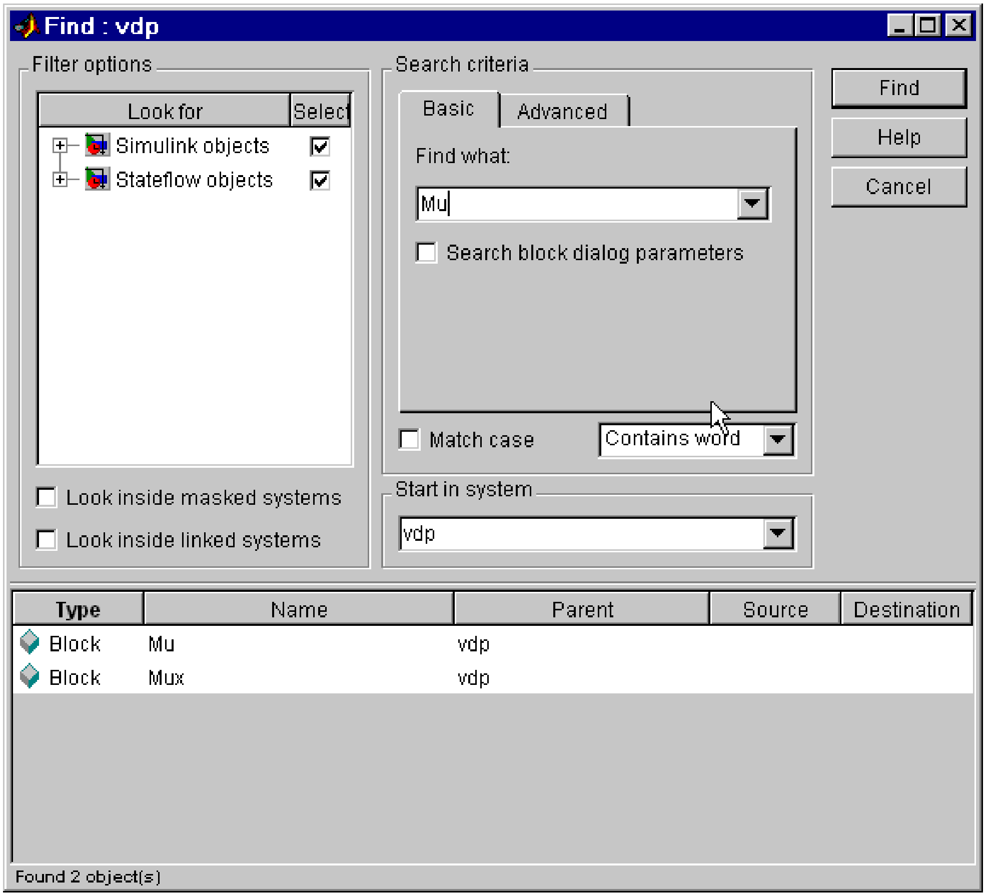
□ Search block dialog parameters

□ Match case — Contains word

Start in system: vdp

Use the **Filter options** (see "Filter Options" on page 10-24) and **Search criteria** (see "Search Criteria" on page 10-25) 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	☑

☐ Look inside masked systems

☐ Look inside linked systems

Find what: Mu

☐ Search block dialog parameters

☐ Match case   Contains word

Start in system: vdp

Type	Name	Parent	Source	Destination
Block	Mu	vdp		
Block	Mux	vdp		

Found 2 object(s)

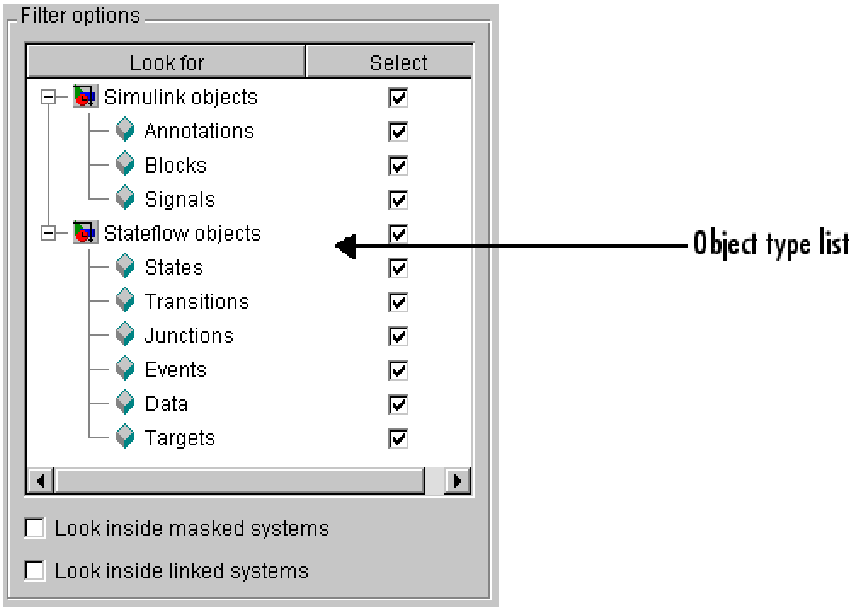
You can display an object by double-clicking its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object. To sort the results list, click any of the buttons at the top of each column. For example, to sort the 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.

See the following topics for details on:

- “Filter Options” on page 10-24
- “Search Criteria” on page 10-25

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

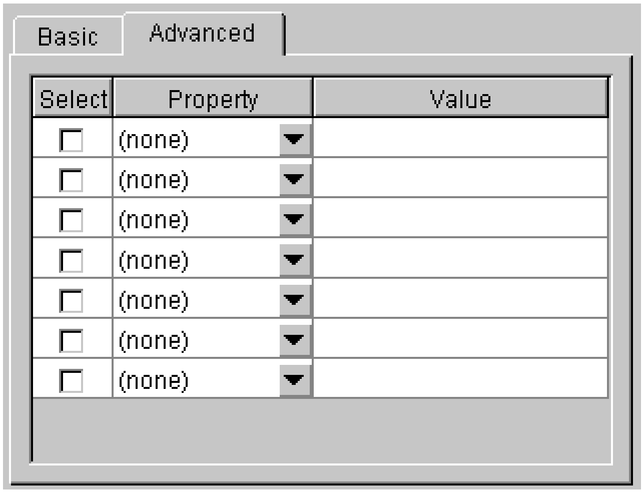
The **Search criteria** panel allows you to specify the criteria that objects must meet to satisfy your search request.

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]`).
`\d`	Matches a digit (same as `[0-9]`).
`\D`	Matches a nondigit (same as `[^0-9]`).

Character	Meaning
`\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.

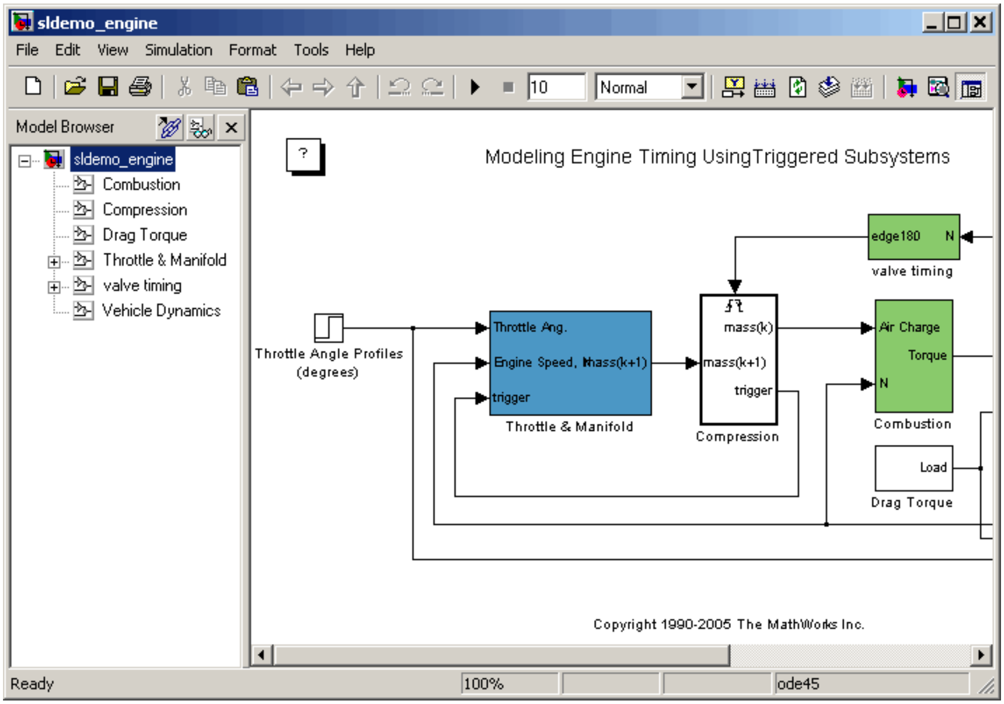
The Model Browser

The Model Browser enables you to

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

Note The browser is available only on Microsoft Windows platforms.

To display the Model Browser, select **Model Browser Options > Model Browser** from the Simulink **View** menu.



The model window splits into two panes. The left pane displays the browser, a tree-structured view of the block diagram displayed in the right pane.

Note The **Browser initially visible** preference causes Simulink to open models by default in the Model Browser. To set this preference, select **Preferences** from the Simulink **File** menu.

The top entry in the tree view corresponds to your model. A button next to the model name allows you to expand or contract the tree view. The expanded view shows the model's subsystems. A button next to a subsystem indicates that the subsystem itself contains subsystems. You can use the button to list the subsystem's children. To view the block diagram of the model or any subsystem displayed in the tree view, select the subsystem. You can use either the mouse or the keyboard to navigate quickly to any subsystem in the tree view.

The following sections discuss:

- "Navigating with the Mouse" on page 10-30
- "Navigating with the Keyboard" on page 10-30
- "Showing Library Links" on page 10-31
- "Showing Masked Subsystems" on page 10-31

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.

Model Dependencies

Use the model dependencies manifest tools to

- List files required by your model in a “manifest” file
- Package the model with required files into a zip file
- Compare older and newer manifests for the same model

See “Using File Manifests” on page 10-32.

Use the Model Dependencies Viewer to view libraries and models referenced by your model. You can explore a graph of all the models and libraries.

See “Model Dependency Viewer” on page 10-39.

Using File Manifests

The model dependencies tools identify files required by your model. The files are listed in a “manifest” file. The manifest is an XML file with the extension `.smf`. The default location for the manifest file is the same directory as the model itself. Any HTML reports generated from manifest files will be created as separate files in your temporary directory. To locate your temporary directory, see the MATLAB function `tempdir`.

The following sections describe using manifests:

- “Generating Manifests” on page 10-33
- “Editing Manifests” on page 10-33
- “Comparing Manifests” on page 10-34
- “Exporting Files in Manifest” on page 10-35
- “Scope of Analysis” on page 10-35
- “M-Code Analysis” on page 10-37
- “Best Practices for Using Manifests” on page 10-38

Note To quickly package a model with required files, you can go straight to the Export Files in Manifest dialog, where you can also generate and edit the manifest.

Generating Manifests

To generate a list of required files in a manifest file:

1. Select **Tools > Model Dependencies > Generate Manifest**.

   The Generate Manifest dialog appears.

2. Select the **Analysis scope**: `Full Analysis`, or `MDL Files Only` (see "Scope of Analysis" on page 10-35).

3. Select the check box **View HTML report on completion** to display a report when you generate the manifest.

   You can choose `Plain HTML` or `HTML with Hyperlinks` from the **Report style** list.

4. Click **OK**.

The report lists required files and toolboxes, and details of references to other files so you can identify where dependencies arise.

Editing Manifests

To edit the list of required files in a manifest file:

1. Select **Tools > Model Dependencies > Edit Manifest**.

   The Choose Manifest dialog appears.

   If you do not already have a manifest, you can click **Generate Manifest** to open the Generate Manifest dialog (see "Generating Manifests" on page 10-33). When you have generated the manifest, you return to the Choose Manifest dialog.

2. Select the manifest file to edit, and click **OK**.

The Edit Manifest Contents dialog appears.

3 To add or remove a file:

- Click **Add** to open a file browser and locate a file to add to the list.
- To remove a file, click to select the file in the list (you can multiselect), and click **Remove**.

4 Click **Show Report** to display the HTML report listing required files and toolboxes, and details of references to other files so you can identify where dependencies arise.

5 When you are satisfied with the manifest list, click **OK**.

Comparing Manifests

You can compare any two manifests.

To compare manifest files:

1 Select **Tools > Model Dependencies > Compare Manifests**.

The Compare Manifests dialog appears.

2 Select a newer manifest file.

You can click **Generate Manifest** to create a new one. This opens the Generate Manifest dialog. See "Generating Manifests" on page 10-33. When you have generated the manifest, you return to the Compare Manifests dialog.

3 Select an older manifest file.

4 Choose a report name and location.

5 Click **OK**.

When you click **OK**, the report appears comparing the file lists in the two manifests.

Exporting Files in Manifest

You can export copies of the files listed in the manifest, to package the model with required files into a zip file.

To export your model with required files:

1. Select **Tools > Model Dependencies > Export Files in Manifest**.

   The Export Files in Manifest dialog appears.

2. Select a manifest, or click **Generate Manifest** to create one. The Generate Manifest dialog appears. See "Generating Manifests" on page 10-33. When you have generated the manifest, you return to the Export Files in Manifest dialog.

3. Click **Edit Manifest** to view or change the list of required files. See "Editing Manifests" on page 10-33.

   When you close the Edit Manifest Contents dialog, you return to the Export Files in Manifest dialog.

4. Select the check box **Preserve directory hierarchy** if you want to keep directory structure for your exported model and files. You can then select the root directory to use for this structure.

5. Choose a zip file name to export to, and click **OK**.

Scope of Analysis

When Simulink generates a manifest file, it performs a static analysis on your model, which means that the model does not need to be capable of performing an "update diagram" operation (see "Updating a Block Diagram" on page 2-14).

The two scope of analysis settings are:

1. `MDL Files Only`

   Select `MDL Files Only` if you want to see only the full hierarchy of the Simulink model files (library links and model reference), including all the model files that are inside MathWorks toolboxes and blocksets.

The analysis scope of `MDL Files Only` looks for file dependencies introduced by:

- Model reference
- Library links

2 `Full Analysis`

Select `Full Analysis` if you want to see all the files that the model depends upon. With this option, further analysis is not performed on any MDL files that form part of MathWorks toolboxes and blocksets.

The analysis scope of `Full Analysis` looks for file dependencies introduced by:

- Model reference
- Library links
- Block and model callbacks
- Custom S-functions (M-code and C)
- From File source blocks
- MATLAB Fcn blocks
- Real-Time Workshop custom code
- Real-Time Workshop Embedded Coder templates

The HTML report shows details of the analysis scope under the heading **Dependency analysis settings**. The values of true or false reported depend on the selected Scope of Analysis, as shown in the following table.

Analysis Type	Full Analysis	MDL Files Only
Analyze files in MathWorks toolboxes	False	True
Analyze files in user-defined toolboxes	True	True
Find model references	True	True
Find library links	True	True

Analysis Type	Full Analysis	MDL Files Only
Find S-functions	True	False
Find callback files	True	False
Find code generation files	True	False

The analysis may not find all files required by your model (for examples, see "M-Code Analysis" on page 10-37).

To include dependencies that the analysis was not able to detect, you can add additional file dependencies to a manifest file using the Edit Manifest option (see "Editing Manifests" on page 10-33).

The analysis may not report certain blocksets or toolboxes required by a model. You should be aware of this when sending a model to another user. Blocksets that do not introduce dependence on any files (such as Simulink Fixed Point) cannot be detected.

If your model uses classes created using the Data Class Designer, you need to manually add some files to the manifest in order to ensure that the model works properly when exported. If the model uses a class in package "`packagename`", the generated manifest will contain only the file `@packagename/schema.p`. You need to manually add all the files in the `@packagename` directory and its subdirectories to the manifest.

M-Code Analysis

When the dependency analysis tool encounters M-code, for example in a model, block callback, or in an M-file S-Function, it attempts to identify the files it refers to. If those files contain M-code, they are also analyzed. This is similar to the functionality of `depfun` but with some enhancements:

- Files that are in MathWorks toolboxes are not analyzed.
- Strings passed into calls to `eval`, `evalc`, and `evalin` are analyzed.
- File names passed to `load`, `fopen`, and `imread` are identified.

File names passed to load, etc., are identified only if they are literal strings. For example:

```
load('mydatafile')
load mydatafile
```

The following example, and anything more complicated, will not be identified as a file dependency:

```
str = 'mydatafile';
load(str);
```

Similarly, arguments to eval, etc., are analyzed only if they are literal strings.

The dependency analysis tool looks inside MAT-files to find the names of variables which will be loaded. This enables it to distinguish reliably between variable names and function names in block callbacks.

Best Practices for Using Manifests

The starting point for dependency analysis is the model itself. Make sure that the model refers to any data files it needs, even if you would normally load these manually. For example, add code to the model's PreLoadFcn to load them automatically, e.g.,

```
load mydatafile
load('my_other_data_file.mat')
```

This way, the dependency analysis tools can add them to the manifest. For more detail on how callbacks are analyzed, see the notes on M-code analysis (see "M-Code Analysis" on page 10-37).

More generally, it is important to make sure that any variables used by the model are created or loaded by the model itself, either in model callbacks or in scripts called from model callbacks. This reduces the possibility of the dependency analysis tool confusing variable names with function names when analyzing block callbacks.

If you plan to export the manifest after creating it, try to ensure that the model does not refer to any files by their absolute paths, e.g.,

```
load C:\mymodel\mydata\mydatafile.mat
```

because these may become invalid when the model is exported to another machine. If referring to files in other directories, do it by relative path, e.g.,

```
load mydata\mydatafile.mat
```

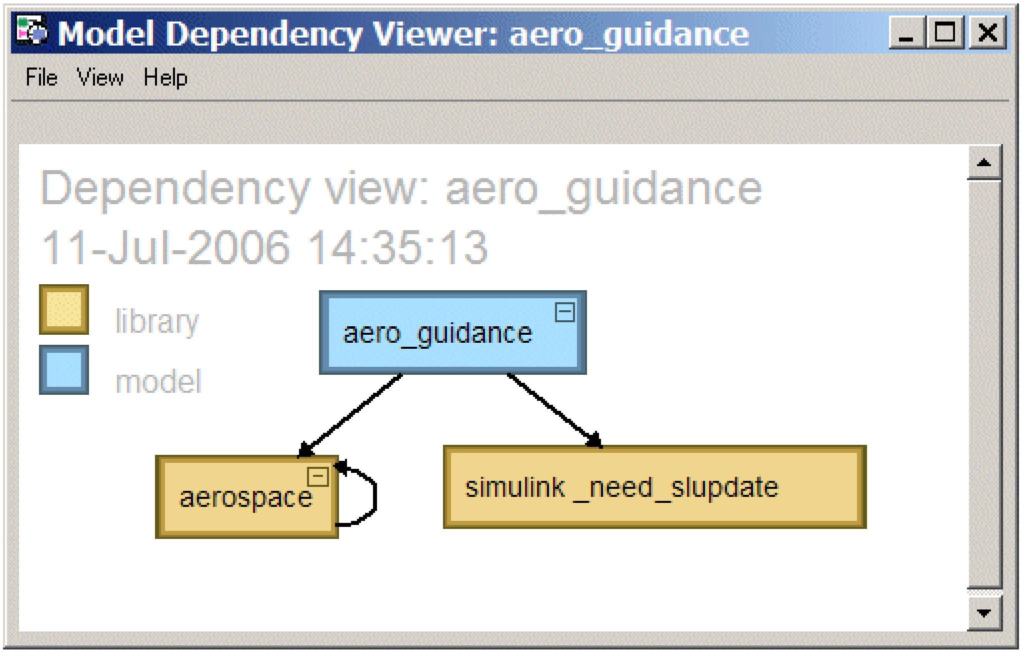
and then select the check box **Preserve directory hierarchy** when exporting, so that the exported files are in the same locations relative to each other. Also, choose a root directory so that all the files listed in the manifest are inside it. Otherwise, any files outside the root will be copied into a new directory called `external` underneath the root, but relative paths to those files will no longer be valid.

Always test exported zip files by extracting the contents to a new location on your computer and testing the model. Be aware that in some cases required files may be on your path but not in the zip file, if your path contains references to directories other than MathWorks toolboxes.

Model Dependency Viewer

The Model Dependency Viewer displays a dependency view of a model. The dependency view is a graph of all the models and libraries referenced directly or indirectly by the model. You can use the dependency view to quickly find and open referenced libraries and models.

To display a dependency view for a model, open the model and select **Tools > Model Dependencies > Model Dependency Viewer** from the model editor's menu bar. An instance of the Model Dependency Viewer appears, displaying a dependency view of the model.

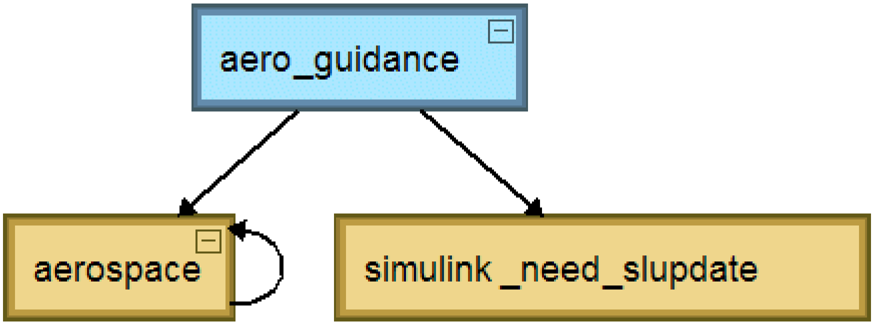


See the following topics for information on using the viewer:

- "About Model Dependency Views" on page 10-40
- "Manipulating a Dependency View" on page 10-42
- "Browsing Dependencies" on page 10-44
- "Saving a Dependency View" on page 10-45
- "Printing a Dependency View" on page 10-45

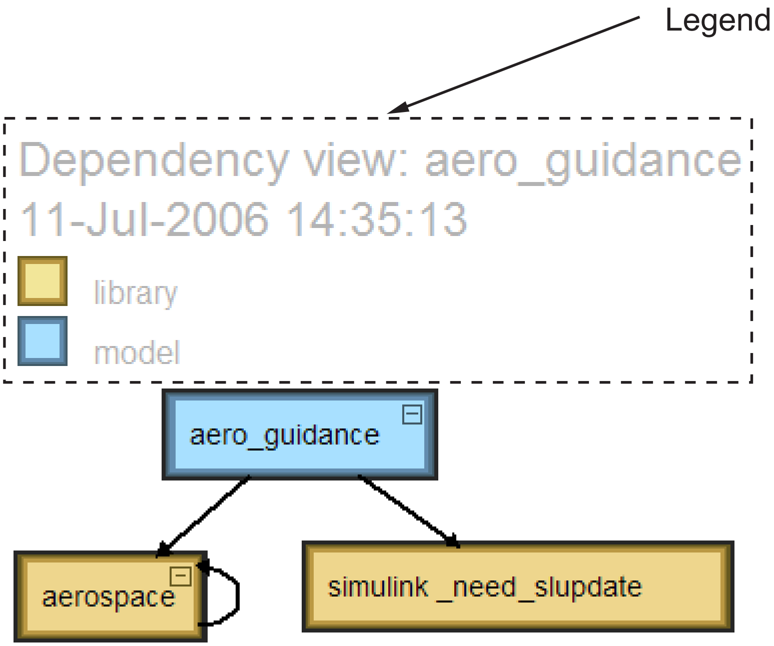
About Model Dependency Views

A model dependency view consists of a set of blocks connected by arrows. Blue blocks represent models; brown boxes, libraries. Arrows represent dependencies. For example, the arrows in the following view indicate that the `aero_guidance` model references two libraries: `aerospace` and `simulink_need_slupdate`.



An arrow that points to the library from which it emerges indicates that the library references itself, i.e., blocks in the library reference other blocks in that same library. For example, the preceding view indicates that the `aerospace` library references itself.

A model dependency view optionally includes a legend that identifies the model in the view and the date and time the view was created.



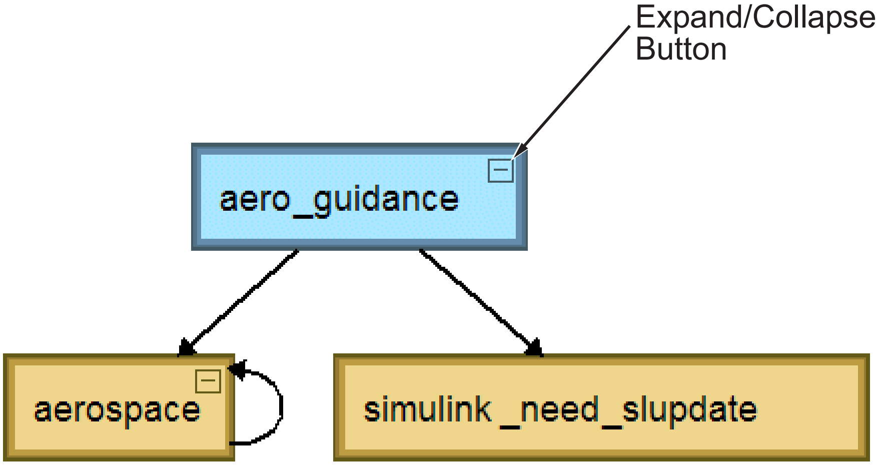
Manipulating a Dependency View

The Model Dependency Viewer allows you to manipulate dependency views in various ways. See the following topics for more information:

- "Displaying Simulink Blocksets" on page 10-43
- "Expanding or Collapsing Views" on page 10-43
- "Zooming a Dependency View" on page 10-43
- "Moving a Dependency View" on page 10-44
- "Rearranging a Dependency View" on page 10-44
- "Displaying and Hiding a Dependency View's Legend" on page 10-44
- "Refreshing a Dependency View" on page 10-44

Displaying Simulink Blocksets. A dependency view initially displays dependencies only on custom block libraries. It does not display, for example, dependencies on standard Simulink block libraries or optional blocksets. To display dependencies on optional Simulink blocksets, select **View > Simulink Blocksets** from the dependency viewer's menu bar.

Expanding or Collapsing Views. To hide or display a particular model or library's dependencies, click the expand(+)/collapse(-) button displayed in the upper, right corner of the block representing the model or library.



To expand all dependencies in a view, select **View > Expand All** from the dependency viewer's menu bar.

Zooming a Dependency View. To zoom a dependency view:

1 Move the cursor over the view.

2 Press your keyboard's space bar and your mouse's right button simultaneously.

   The cursor shape changes to a cross-arrow.

3 Move the cursor down or up to zoom the view in or out, respectively.

Moving a Dependency View. To move a dependency view to another location in the viewer window:

1. Move the cursor over the view.
2. Press your keyboard's space bar and your mouse's left button simultaneously.

   The cursor shape changes to a cross-arrow.
3. Move the cursor to drag the view to another location.

Rearranging a Dependency View. You can rearrange a dependency view by moving the blocks representing models and libraries. This can make a complex view easier to read. To move a block to another location:

1. Select the block you want to move by clicking it.
2. Drag and drop the block in the new location.

Displaying and Hiding a Dependency View's Legend. To display or hide a dependency view's legend, select **View > Show Legend** from the viewer's menu bar.

Refreshing a Dependency View. After changing a model displayed in a dependency view or any of its dependencies, you must update the view to reflect any dependency changes. To update the view, select **View > Refresh** from the dependency viewer's menu bar.

Browsing Dependencies

You can use a dependency view to browse a model's dependency.

- To open a model or library displayed in the view, click its block.
- To open all models displayed in the view, select **File > Open All Models** from the viewer's menu bar.
- To save all models displayed in the view, select **File > Save All Models**.
- To close all models displayed in the view, select **File > Close All Models**.

Saving a Dependency View

You can save a dependency view for later viewing. To save the current view, select **File > Save As** from the viewer's menu bar and use the resulting dialog box to save the view under a name of your choosing. To reopen the view, select **File > Open** from any viewer's menu bar and use the resulting file open dialog box to open the view.

Printing a Dependency View

To print a dependency view, select **File > Print** from the dependency viewer's menu bar.

11

Running Simulations

The following sections explain how to use Simulink to simulate a dynamic system.

Simulation Basics

You can simulate a Simulink model at any time simply by clicking the **Start** button on the Model Editor displaying the model (see “Starting a Simulation” on page 11-5). 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 the model. Simulink enables you to create multiple model configurations, called configuration sets, modify existing configuration sets, and switch configuration sets with a click of a mouse button (see “Configuration Sets” on page 11-37 for information on creating and selecting configuration sets).

Once you have defined or selected a model configuration set that meets your needs, you can start the simulation. 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 11-6), 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.

Controlling Execution of a Simulation

The Simulink graphical interface includes menu commands, toolbar buttons, and blocks that enable you to start, stop, and pause a simulation.

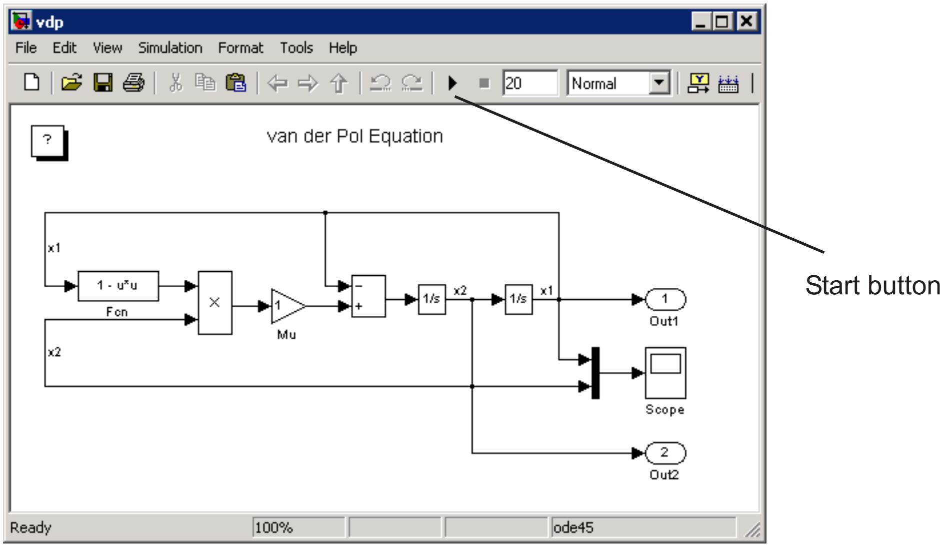
Note This sections explains how to run a simulation interactively. See “Running a Simulation Programmatically” on page 11-152 for information on running a simulation from a program, S-function, or the MATLAB command line.

The following topics show you haw to start and stop a simulation, and to use blocks:

- “Starting a Simulation” on page 11-5
- “Pausing or Stopping a Simulation” on page 11-6
- “Using Blocks to Stop or Pause a Simulation” on page 11-7

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.

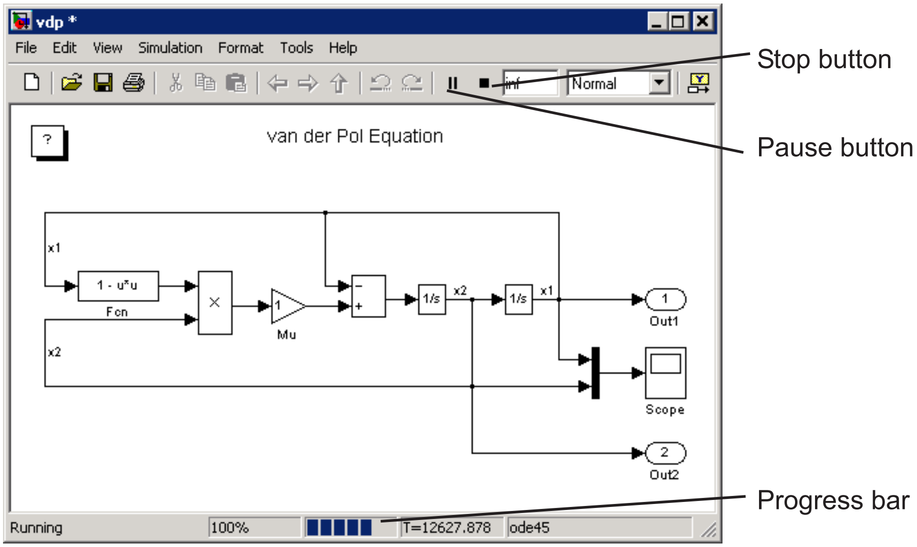


You can also use the keyboard shortcut, **Ctrl+T**, to start the simulation.

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 "Configuration Parameters Dialog Box" on page 11-66).

While the simulation is running, a progress bar at the bottom of the model window shows how far the simulation has progressed. A **Stop** command replaces the **Start** command on the **Simulation** menu. A **Pause** command appears on the menu and replaces the **Start** button on the model toolbar.



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.

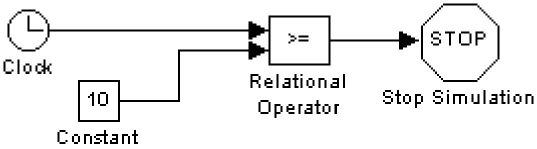
Using Blocks to Stop or Pause a Simulation

Using Stop Blocks

You can use the Stop Simulation block to terminate a simulation when the block's input is nonzero. To use the Stop Simulation block:

1. Drag a copy of the Stop Simulation block from the Sinks library and drop it into your model.
2. Connect the Stop Simulation block to a signal whose value becomes nonzero at the time when the simulation should be terminated.

For example, this model stops the simulation when the input signal reaches 10.



If the block input is a vector, any nonzero element causes the simulation to terminate.

Creating Pause Blocks

You can use an Assertion block to pause the simulation when the block's input signal is zero. To create a pause block:

1. Drag a copy of the Assertion block from the Model Verification library and drop it into your model.
2. Connect the Assertion block to a signal whose value becomes zero at the time when the simulation should be paused.
3. Open the Assertion block's **Block Parameters** dialog box.

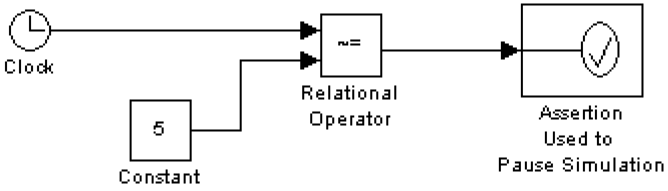
- Enter the following commands into the **Simulation callback when assertion fails** field:

  ```
  set_param(bdroot,'SimulationCommand','pause'),
  disp(sprintf('\nSimulation paused.'))
  ```

- Uncheck the **Stop simulation when assertion fails** option.

4 Click **OK** to apply the changes and close this dialog box.

The following model uses an Assertion block configured as described above, in conjunction with the Relational Operator block, to pause the simulation when the simulation time is equal to 5.



When the simulation pauses, the Assertion block displays the following message at the MATLAB command line.

```
Simulation paused
Warning: Assertion detected in 'assertion_as_pause/
   Assertion Used to Pause Simulation' at time 5.000000
```

You can resume the suspended simulation by choosing **Continue** from the **Simulation** menu on the model editor, or by selecting the **Continue** button in the toolbar.

Note The Assertion block uses the `set_param` command to pause the simulation. See “Running a Simulation Programmatically” on page 11-152 for more information on using the `set_param` command to control the execution of a Simulink model.

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
  - Dimension of any signals
- Display block data tips on a Microsoft Windows installation (see “Block Data Tips” on page 4-2).

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 "Solver Pane" on page 11-68 for more information. On Microsoft Windows, you can also specify the simulation stop time in the Model Editor's toolbar.

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.

- "Choosing a Solver Type" on page 11-11
- "Choosing a Fixed-Step Solver" on page 11-12
- "Choosing a Variable-Step Solver" on page 11-17

For information on tailoring the selected solver to your model, see "Improving Simulation Accuracy" on page 11-150.

Choosing a Solver Type

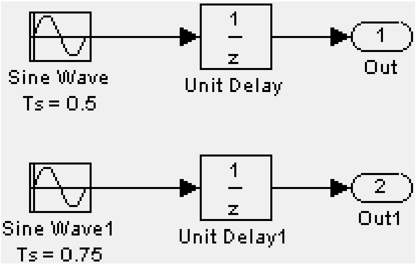
Simulink divides solvers into two types: fixed-step and variable-step. Both types of solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the step size. With a fixed-step solver, the step size remains constant throughout the simulation. By contrast, 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 "Solver Pane" on page 11-68).

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 1-20) 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 ensures 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 11-14) 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 11-16) 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

Solver	Integration Technique
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 11-76). The more iterations and the higher the extrapolation 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

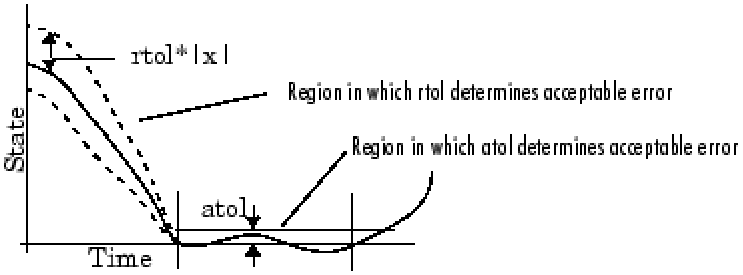
The solvers use standard local error control techniques to monitor the error at each time step. During each time step, the solvers compute the state values at the end of the step and also determine the *local error*, the estimated error of these state values. They then compare the local error to the *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 ith 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
State
Region in which atol determines acceptable error
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 Data from the MATLAB Workspace" on page 11-21
- "Exporting Data to the MATLAB Workspace" on page 11-27
- "Importing and Exporting States" on page 11-31
- "Limiting Output" on page 11-33
- "Specifying Output Options" on page 11-34

Importing Data from the MATLAB Workspace

Simulink can apply input from the MATLAB workspace to the model's top-level input ports 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 11-81). Then, enter an external input specification in the adjacent edit box and click **Apply**.

Note The use of the **Input** box is independent of the setting of the **Format** list on the **Data Import/Export** pane.

The input data can take any of the following forms:

- time series — see "Importing Time-Series Data" on page 11-22
- array — see "Importing Data Arrays" on page 11-23
- time expression — see "Using a MATLAB Time Expression to Import Data" on page 11-24
- structure — see "Importing Data Structures" on page 11-24

Simulink linearly interpolates or extrapolates input values as necessary if the **Interpolate data** option is selected for the corresponding Inport.

Importing Time-Series Data

Any root-level Inport block can import data specified by a time-series object (see `Simulink.Timeseries` in the online Simulink reference) residing in the MATLAB workspace. In addition, any root-level input port defined by a bus object (see `Simulink.Bus` in the online Simulink reference) can import data from a time-series array object (see `Simulink.TSArray` in the online Simulink reference) that has the same structure as the bus object. Simulink time-series objects are a derivation of standard MATLAB time-series objects and, therefore, can be manipulated using the MATLAB Time Series Tools. See “Using Time Series Tools” in the MATLAB Data Analysis documentation for further details.

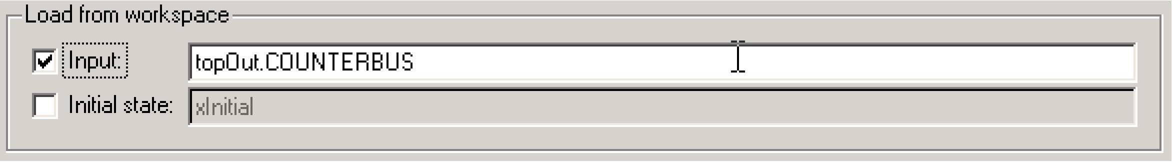
Importing time-series objects allows you to import data logged by a previous simulation run (see “Logging Signals” on page 5-34). For example, suppose that you have a model that references several other models. You could use data logged from the inputs of the referenced models when simulating the top model as inputs for the referenced models simulated by themselves. This allows you to test the referenced models independently of the top model and each other.

To import data from time-series objects and time-series array objects, enter a comma-separated list of variables or expressions into the **Input** edit field on the **Data Import/Export** pane of the Configuration Parameters dialog box (see “Configuration Parameters Dialog Box” on page 11-66). Each variable or expression in the **Input** list should evaluate to the appropriate time-series object or time-series array object that corresponds to one of the model’s root-level input ports, with the first item corresponding to the first root-level input port, the second to the second root-level input port, and so on. The model `sldemo_mdlref_counter_bus`, referenced by the top model `sldemo_mdlref_bus`, contains an example of importing time-series objects.

To use this demo, open `sldemo_mdlref_bus` and run the simulation. The top model is configured to store logged signals into a variable named `topOut`. Currently, two signals are being logged: `COUNTERBUS` and `OUTPUTBUS`. After running the simulation, you can view the logged signals by typing `topOut` at the MATLAB prompt.

```
topOut =

Simulink.ModelDataLogs (sldemo_mdlref_bus):
  Name                   Elements  Simulink Class

  COUNTERBUS                 2      TsArray
  OUTPUTBUS                  2      TsArray
```

The variable `topOut` is a `Simulink.ModelDataLogs` object that contains, in this case, two `Simulink.TsArray` objects corresponding to the two logged bus signals. The `Simulink.TsArray` object `COUNTERBUS` can be used as the input to the submodel `sldemo_mdlref_counter_bus` to run this model independently of the top model. This is accomplished by entering `topOut.COUNTERBUS` into the **Input** edit field on the **Data Import/Export** pane of the Configuration Parameters dialog box, as shown below.


Load from workspace

☑ Input: topOut.COUNTERBUS

□ Initial state: xInitial

By using the time-series array object as the input to `sldemo_mdlref_counter_bus`, independently running this model produces the same output as when run within the top model `sldemo_mdlref_bus`.

Importing Data Arrays

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

The total number of columns of the input matrix must equal `n + 1`, where `n` is the total number of signals entering the model's input ports.

The default input expression for a model is `[t,u]` and the default input format is `Array`. So if you define `t` and `u` in the MATLAB workspace, you need only select the **Input** option to input data from the model workspace.

For example, suppose that a model has two input ports, one of which accepts two signals and the other of which accepts one signal. Also, suppose that the MATLAB 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 input ports. For example, suppose that a model has one vector Inport that accepts two signals. Furthermore, suppose that timefcn is a user-defined function that returns a row vector two elements long. The following are valid input time expressions for such a model:

```
'[3*sin(t), cos(2*t)]'

'4*timefcn(w*t)+7'
```

Simulink evaluates the expression at each step of the simulation, applying the resulting values to the model's input ports. 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. Simulink evaluates the type of data structure based on the structure itself.

Importing Signal-and-Time Data Structures. To import structures that include both signal and time data, the input structure must have two top-level fields: time and signals. The time field contains a column vector of the simulation times. The signals field contains an array of substructures, each of which corresponds to a model input port.

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

Note You must set the **Port dimensions** parameter of the Inport to be the same value as the dimensions field of the corresponding input structure. If the values differ, Simulink stops and displays an error message when you try to simulate the model.

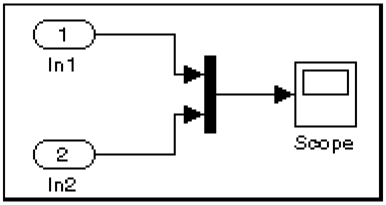
If the inputs for a port are scalar or vector values, the values field must be an M-by-N array where M is the number of time points specified by the time field and N is the length of each vector value. For example, the following code creates an input structure for loading 11 time samples of a two-element 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 input port, you would select the **Input** option on the **Data Import/Export** pane and enter a in the input expression field.

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

As another example, consider the following model, which has two inputs.



Suppose that you want to input a sine wave into the first port and a cosine wave into the second port. To do this, define a vector, a, as follows, in the MATLAB workspace:

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

Select the **Input** box for this model, 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 input port's value array, the value for the second time step from the second element of the value array, etc. If you enter the structure without time, the Inport block must have a discrete sample time.

Per-Port Structures. This format consists of a separate structure-with-time or structure-without-time for each port. Each port's input data structure has only one `signals` field. To specify this option, enter the names of the structures in the **Input** text field as a comma-separated list, `in1, in2,...,` `inN`, where `in1` is the data for your model's first port, `in2` for the second input port, and so on.

Specifying Time Vectors for Discrete Systems

In some cases, Simulink calculates block sample hits at sample times different from those specified by a time vector generated in MATLAB. Typically, these are small floating point inaccuracies that can cause Simulink to apparently miss a specified time step in leu of a different sample point. In order to avoid these numerical inaccuracies, generate the time vector either in MATLAB or in Simulink based on the fundamental sample time of the model.

For example, if the model has a fundamental sample time `Ts` (see "Determining Step Size for Discrete Systems") of 0.001 then the time vector `Tvector` should be calculated with the command

```
  Tvector = Ts*[n1, n2, n3...];
```

where *n1*, *n2*, *n3*, etc. are integers that, when multiplied by the fundamental sample time, yield the desired time vector.

Exporting Data to the MATLAB Workspace

Simulink allows you to export a model's states and root-level outputs to the MATLAB workspace during simulation of the model. To do this, select the type of data that you want to export on the **Save to workspace** area of the **Data Import/Export** pane (see "Data Import/Export Pane" on page 11-81) of the Configuration Parameters dialog box. The field adjacent to each type specifies the name of a workspace variable to be used by Simulink to store the exported data. Each field initially specifies a default variable. You can edit the fields to specify names of your own choosing.

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.

Format Options

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 output port, each row to the outputs at a specific time.

Note You can use array format to save your model's outputs and states only if the outputs are either all scalars or all vectors (or all matrices for states), are either all real or all complex, and are all of the same data type. Use the `Structure` or `Structure with time` output formats (see "Structure with time" on page 11-28) 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`

  Contains a vector of the simulation times.

- signals

  Contains an array of substructures, each of which corresponds to a model output port.

Each substructure has four fields:

- values

  Contains the outputs for the corresponding output port. If the outputs are scalars or vectors, the values field is a matrix each of whose rows represents an output at the time specified by the corresponding element of the time vector. If the outputs are matrix (2-D) values, the values field is a 3-D array of dimensions M-by-N-by-T where M-by-N is the dimensions of the output signal and T is the number of output samples. If T = 1, MATLAB drops the last dimension. Therefore, the values field is an M-by-N matrix.
- dimensions

  Specifies the dimensions of the output signal.
- label

  Specifies the label of the signal connected to the output port or the type of state (continuous or discrete).
- blockName

  Specifies the name of the corresponding output port or block with states.
- inReferencedModel

  Contains a value of 1 if the signals field records the final state of a block that resides in the submodel. Otherwise, the value is false (0).

The following is an example of the structure-with-time format for a nonreferenced model.

```
>> xout.signals(1)

ans =

                 values: [296206x1 double]
             dimensions: 1
```

```
               label: 'CSTATE'
           blockName: 'vdp/x1'
   inReferencedModel: 0
```

The structure used to save states has a similar organization. The states structure has two top-level fields:

- `time`

  The `time` field contains a vector of the simulation times.
- `signals`

  The field contains an array of substructures, each of which corresponds to one of the model's states.

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

The time samples of a state are stored in the `values` field as a matrix of values. Each row corresponds to a time sample. Each element of a row corresponds to an element of the state. If the state is a matrix, the matrix is stored in the `values` array in column-major order. For example, suppose that the model includes a 2-by-2 matrix state and that 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.

Note 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 11-25 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 input port, and so on.

Importing and Exporting States

Simulink allows you to import the initial values of a system's states, i.e., its initial conditions, at the beginning of a simulation and save the final values of the states at the end of the simulation. This feature allows you to save a steady-state solution and restart the simulation at that known state.

Saving Final States

To save the final values of a model's states, check **Final states** in the **Save to workspace** area of the **Data Import/Export** pane and enter a name in the adjacent edit field. Simulink saves the states in a workspace variable having the specified name. The saved data has the format that you specify in the **Save options** area of the **Data Import/Export** pane.

When saving states from a referenced model in the structure-with-time format, Simulink adds a boolean subfield named `inReferencedModel` to the `signals` field of the saved data structure. This field's value is true (`1`) if the `signals` field records the final state of a block that resides in the submodel, and a `0` otherwise. For example,

```
>> xout.signals(1)

ans =

               values: [101x1 double]
           dimensions: 1
                label: 'DSTATE'
            blockName: [1x66 char]
    inReferencedModel: 1
```

If the signals field records a submodel state, its blockName subfield contains a compound path comprising a top model path and a submodel path. The top model path is the path from the model root to the Model block that references the submodel. The submodel path is the path from the submodel root to the block whose state the signals field records. The compound path uses a | character to separate the top and submodel paths, e.g.,

```
>> xout.signals(1).blockName

ans =

sldemo_mdlref_basic/CounterA|sldemo_mdlref_counter/Previous Output
```

Loading Initial States

To load states, check **Initial state** in the **Load from workspace** area of the **Data Import/Export** pane and specify the name of a variable that contains the initial state values, for example, a variable containing states saved from a previous simulation. The initial values specified by the workspace variable override the initial values specified by the model itself, i.e., the values specified by the initial condition parameters of those blocks in the model that have states.

Use the structure or structure-with-time option to specify initial states if you want to accomplish any of the following.

- Associate initial state values directly with the full path name to the states. This eliminates errors that could occur if Simulink reorders the states, but the initial state array is not correspondingly reordered.
- Assign a different data type to each state's initial value.
- Initialize only a subset of the states.

For example, the following commands create an initial state structure that can be used to initialize the x2 state of the vdp model. The x1 state is not initialized in the structure and, therefore, the value entered into the state's associated Integrator block is used during the simulation.

```
% Open the vdp demo model
vdp

% Use getInitialState to obtain an initial state structure
states = Simulink.BlockDiagram.getInitialState('vdp');

% Set the initial value of the signals structure element
% associated with x2 to 2.
states.signals(2).values = 2;

% Remove the signals structure element associated with x1
states.signals(1) = [];
```

To use this `states` variable, open the **Configuration Parameters** dialog box for the vdp model. Check **Initial state** in the **Load from workspace** area of the **Data Import/Export** pane and type `states` into the associated edit field. When you run the model, note that both states have the initial value of 2. The initial value of the `x2` state is assigned in the `states` structure, while the initial value of the `x1` state is assigned in its Integrator block.

Note You must use the structure or structure-with-time format to initialize the states of a top model and the models that it references.

Limiting Output

Saving data to 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 11-81) enables you to control how much output the simulation generates. You can choose from three options:

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

Refining Output

The `Refine output` choice provides additional output points when the simulation output is too coarse. This parameter provides an integer number of output points between time steps; for example, a refine factor of 2 provides output midway between the time steps as well as at the steps. The default refine factor is 1.

To get smoother output, it is much faster to change the refine factor instead of reducing the step size. When the refine factor is changed, the solvers generate additional points by evaluating a continuous extension formula at those points. This option changes the simulation step size so that time steps coincide with the times that you have specified for additional output.

The refine factor applies to variable-step solvers and is most useful when you are using `ode45`. The `ode45` solver is capable of taking large steps; when graphing simulation output, you might find that output from this solver is not sufficiently smooth. If this is the case, run the simulation again with a larger refine factor. A value of 4 should provide much smoother results.

Note This option helps the solver locate zero crossings (see "Zero-Crossing Detection" on page 1-20). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Producing Additional Output

The `Produce additional output` choice enables you to specify directly those additional times at which the solver generates output. When you select this option, 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. This option causes the solver to produce hit times at the output times you have specified, in addition to the times it needs to accurately simulate the model.

Note This option helps the solver locate zero crossings (see "Zero-Crossing Detection" on page 1-20). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Producing Specified Output Only

The `Produce specified output only` choice provides simulation output *only* at the simulation start time, simulation stop time, and the specified output times. For example, if the simulation start time is set to `0` and the simulation stop time is set to `60`, entering `[10: 10: 50]` in the `Output times` field results in simulation output at these times:

```
0, 10, 20, 30, 40, 50, 60
```

This option changes the simulation step size so that time steps coincide with the times that you have specified for producing output. The solver may hit other time steps to accurately simulate the model, however the output will not include these points. This choice is useful when you are comparing different simulations to ensure that the simulations produce output at the same times.

Note This option helps the solver locate zero crossings (see "Zero-Crossing Detection" on page 1-20). In particular, it helps reduce the chance of missing a zero crossing. It does not help locate the missed zero crossings.

Comparing Output Options

A sample simulation generates output at these times:

```
0, 2.5, 5, 8.5, 10
```

Choosing `Refine output` and specifying a refine factor of 2 generates output at these times:

```
0, 1.25, 2.5, 3.75, 5, 6.75, 8.5, 9.25, 10
```

Choosing the `Produce additional output` option and specifying `[0:10]` generates output at these times

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

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

Choosing the `Produce specified output only` option and specifying `[0:10]` generates output at these times:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Configuration Sets

A *configuration set* is a named set of values for a model’s parameters, such as solver type and simulation start or stop time. Every new model is created with a default configuration set, called Configuration, that initially specifies default values for the model’s parameters. You can subsequently create and modify additional configuration sets and associate them with the model. The sets associated with a model can specify different values for any or all configuration parameters.

This section describes techniques for defining and using configuration sets that are stored in individual models. Such configuration sets are available only to the model that contains them. The next section, “Referencing Configuration Sets” on page 11-49, describes techniques for storing configuration sets in the base workspace, independently of any model. Such configuration sets can be shared by any number of models.

Configuration Set Components

A configuration set comprises groups of related parameters called components. Every configuration set includes the following components:

- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing

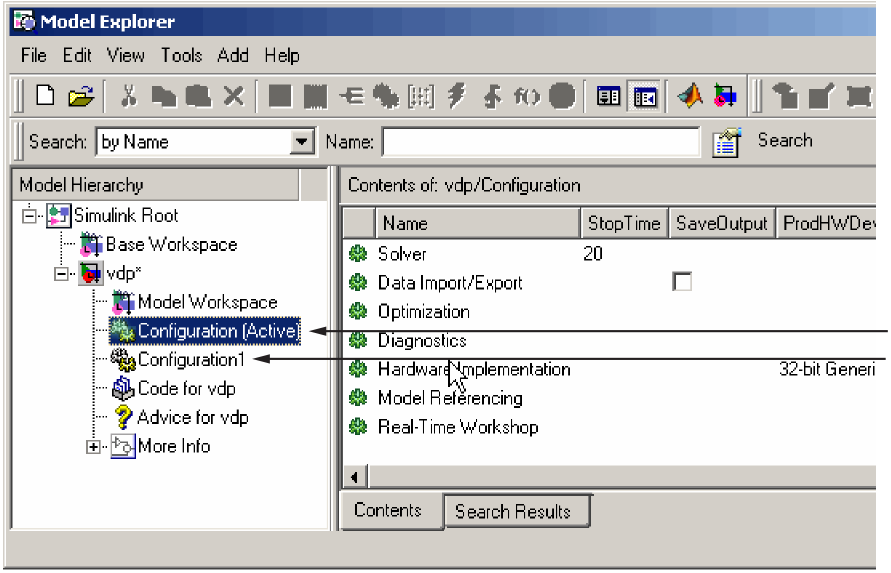
Some Simulink-based products, such as Real-Time Workshop, define additional components. If such a product is installed on your system, the configuration set also contains the components that it defines.

The Active Set

Only one of the configuration sets associated with a model is active at any given time. The active set determines the current values of the model's parameters. Changing the value of a parameter in the Model Explorer changes its value in the associated configuration set. Simulink allows you to change the active or inactive set at any time (except when executing the model). In this way, you can quickly reconfigure a model for different purposes, e.g., testing and production, or apply standard configuration settings to new models.

Displaying Configuration Sets

To display the configuration sets associated with a model, open the Model Explorer (see "The Model Explorer" on page 10-2). The configuration sets associated with the model appear as gear-shaped nodes in the Model Explorer's **Model Hierarchy** pane.



Model Hierarchy

- Simulink Root
  - Base Workspace
  - vdp*
    - Model Workspace
    - Configuration (Active)
    - Configuration1
    - Code for vdp
    - Advice for vdp
    - More Info

Contents of: vdp/Configuration

Name	StopTime	SaveOutput	ProdHWDev
Solver	20		
Data Import/Export		□	
Optimization			
Diagnostics			
Hardware Implementation			32-bit Generi
Model Referencing			
Real-Time Workshop			

Configuration sets

The Model Explorer's **Contents** pane displays the components of the selected configuration set. The Model Explorer's Dialog pane display a dialog for setting the parameters of the selected group (see "Configuration Parameters Dialog Box" on page 11-66).

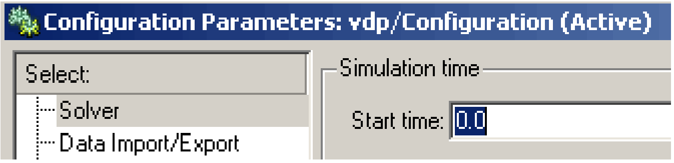
Activating a Configuration Set

To activate a configuration set, right-click the configuration set's node to display the node's context menu, then select **Activate** from the context menu.

Opening Configuration Sets

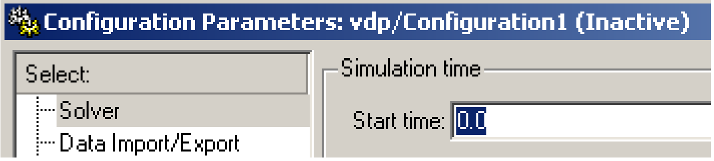
In Model Explorer, to open the configuration parameter dialog for a configuration set, right-click the configuration set's node to display the node's context menu, then select **Open** from the context menu. You can open the configuration parameter dialog for any configuration set, whether it is active or not. You might want to open a configuration set to inspect or edit the parameter settings.

The title bar of the dialog indicates if the configuration set is active or inactive.



Configuration Parameters: vdp/Configuration (Active)

Start time: 0.0



Configuration Parameters: vdp/Configuration1 (Inactive)

Start time: 0.0

Note Every configuration set has its own configuration parameter dialog. As you change the state of a configuration set, the top-left corner label changes to reflect the state.

Copying, Deleting, and Moving Configuration Sets

You can use edit commands on the Model Explorer's **Edit** or context menus or object drag-and-drop operations to delete, copy, and move configuration sets among models displayed in the Model Explorer's **Model Hierarchy** pane.

For example, to copy a configuration set, using edit commands:

1. Select the model with a configuration set that you want to copy in the **Model Hierarchy** pane.

2. Select the configuration set that you want to copy in the **Contents** pane.

3. Select **Copy** from the Model Explorer's **Edit** menu or the configuration set's context menu or press **Ctrl+C**.

4. Select the model in which you want to create the copy.

   Note You can create a copy in the same model as the original.

   ---

5. Select **Paste** from the Model Explorer's **Edit** menu or from the model's context menu or press **Ctrl+V**.

To copy the configuration set, using object drag-and-drop, hold the right mouse button down and drag the configuration set's node to the node of the model in which you want to create the copy. To move a configuration set from one model to another, using drag-and-drop, hold the left mouse button down and drag the configuration set's node to the node of the destination model.

Note You cannot move or delete a model's active configuration set.

Copying Configuration Set Components

To copy a configuration set component from one configuration set to another:

1. Select the component in the Model Explorer's **Contents** pane.
2. Select **Copy** from the Model Explorer's **Edit** menu or the component's context menu or press **Ctrl+C**.
3. Select the configuration set into which you want to copy the component.
4. Select **Paste** from the Model Explorer's **Edit** menu or the component's context menu or press **Ctrl+C**.

   Note The copy replaces the component of the same name in the destination configuration set. For example, if you copy the Solver component of configuration set A and paste it into configuration set B, the copy replaces B's existing Solver component.

Creating Configuration Sets

To create a new configuration set, select **Configuration Set** from the Model Explorer's **Add** menu or press the **Add Configuration** button in the Model Explorer's toolbar. You can also create a new configuration set by copying an existing configuration set.

Setting Values in Configuration Sets

To set the value of a parameter in a configuration set, select the configuration set in the Model Explorer and then edit the value of the parameter on the corresponding dialog 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. The API includes the `Simulink.ConfigSet` data object class and the following model construction commands:

- `attachConfigSet`
- `attachConfigSetCopy`
- `detachConfigSet`
- `getConfigSet`
- `getConfigSets`
- `setActiveConfigSet`
- `getActiveConfigSet`
- `openDialog`
- `closeDialog`

These commands, along with the methods and properties of `Simulink.ConfigSet` class, allow an M-file program to create and modify configuration sets, attach configuration sets to a model, set a model's active configuration set, open and close configuration sets, and detach configuration sets from a model. For example, to create a configuration set from scratch at the command line, enter

```
cfg_set = Simulink.ConfigSet
```

The default name of the new configuration set is `Configuration`. To change the name, execute

```
cfg_set.Name = 'name'
```

where *name* is the set's new name.

Use `get_param` and `set_param` to get and set the value of a parameter in a configuration set. For example, to specify the Simulink fixed-step discrete solver in the configuration set, execute

```
set_param(cfg_set, 'Solver', 'FixedStepDiscrete')
```

To save the configuration set in a MAT-file, execute

```
save mat_file cfg_set
```

where *mat_file* is the name of the MAT-file. To load the configuration set, execute

```
load mat_file
```

To prevent or allow a user to change the value of a parameter in a configuration set using either the Model Explorer or `set_param` command, execute

```
setPropEnabled(cfg_set, 'param', [0 | 1])
```

where *param* is the name of the parameter. To attach a configuration set to a model, execute

```
attachConfigSet(model, cfg_set)
```

where model is the model name (in quotes) or object. To get a model's active configuration set, execute

```
cfg_set = getActiveConfigSet(model)
```

To get a configuration set's full name (e.g., `'engine/Configuration'`), execute

```
getFullName(cfg_set)
```

To set a model's active set, execute

```
setActiveConfigSet(model, 'cfg_set_name')
```

where *cfg_set_name* is the configuration set's name.

To open the configuration parameter dialog for an active configuration set, execute

```
openDialog(cfg_set)
```

To open the configuration parameter dialog for any configuration set, execute

```
cfg_set= getConfigSet(gcs,'cfg_set_name')
openDialog(cfg_set);
```

where `cfg_set_name` is the configuration set's name.

To close the configuration parameter dialog for a configuration set, execute

```
closeDialog(cfg_set);
```

To rename the active configuration set of modelA, copy it, and attach a copy of that configuration to modelB, execute

```
activeConfigA = getActiveConfigSet('modelA');
activeConfigA.Name = 'myactiveConfigA';
newConfig = attachConfigSetCopy('modelB', activeConfigA);
```

where `activeConfigA` is the active configuration set of `modelA`. `modelA` is the model whose active configuration set you want to copy. `modelB` is the model with which you want to associate the copy of the configuration set. You can also use this command to make multiple copies of one configuration set. This might be useful if you want to programmatically assign copies of the same configuration set to multiple models.

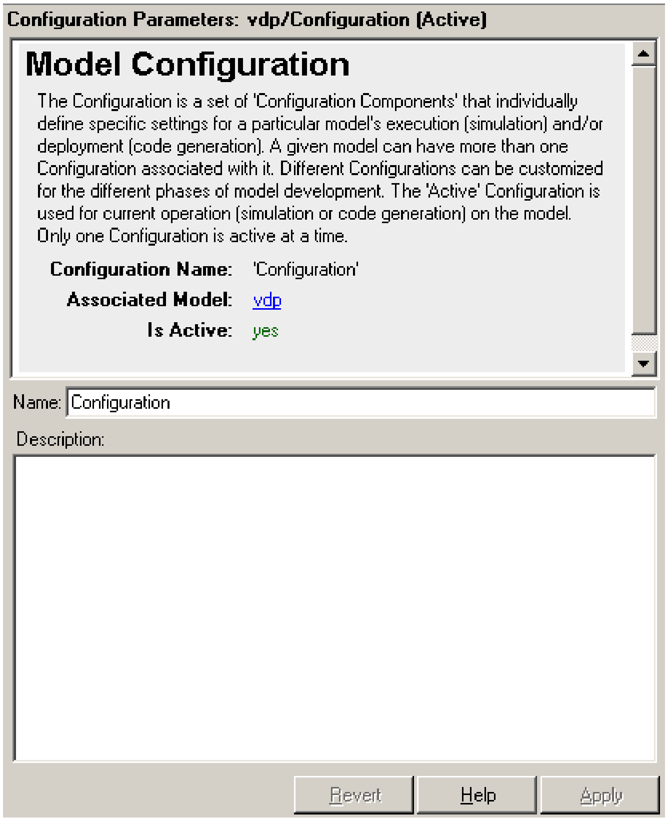
To detach a configuration set from a model, execute

```
detachConfigSet(model, cfg_set)
```

where model is the model name (in quotes) or object.

Model Configuration Dialog

The Model Configuration dialog appears in the Model Explorer dialog pane when you select any model configuration.



Configuration Parameters: vdp/Configuration (Active)

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

Description:

The dialog has the following fields:

Name

Name of the configuration. You can change the configuration name by editing this field.

Description

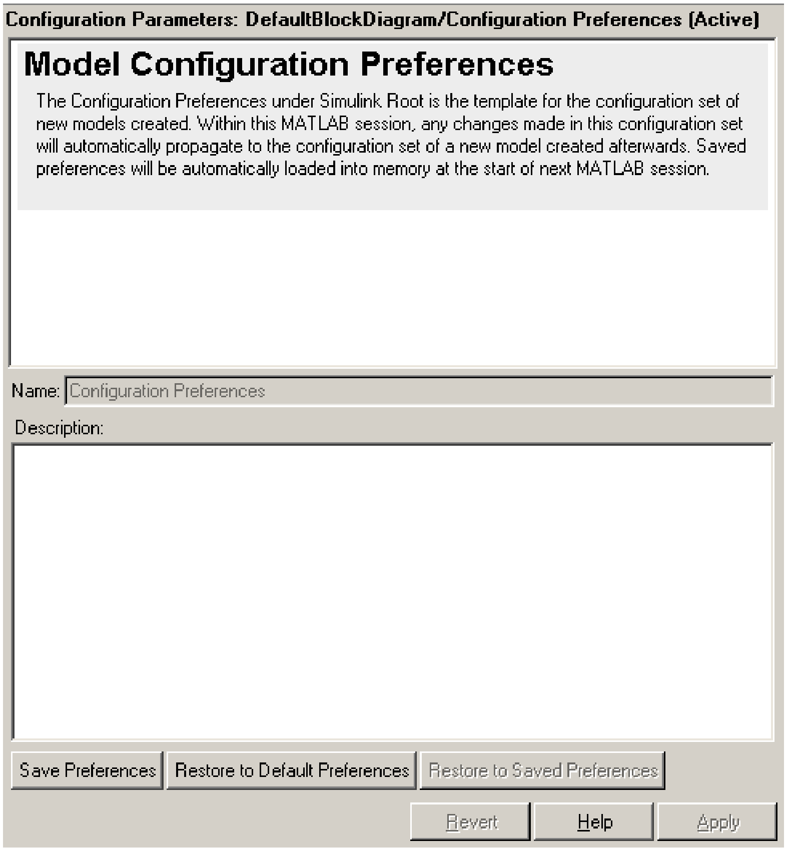
A description of the configuration. You can use this field to enter information about the configuration.

Model Configuration Preferences Dialog

The Model Configuration Preferences dialog appears in the Model Explorer dialog pane when you view the default model configuration.

1. Enable **View > Show Configuration Preferences** in the Model Explorer menu.

2. Select Configuration Preferences under the Simulink Root node in the Model Explorer **Model Hierarchy** pane.



Configuration Parameters: DefaultBlockDiagram/Configuration Preferences (Active)

Model Configuration Preferences

The Configuration Preferences under Simulink Root is the template for the configuration set of new models created. Within this MATLAB session, any changes made in this configuration set will automatically propagate to the configuration set of a new model created afterwards. Saved preferences will be automatically loaded into memory at the start of next MATLAB session.

Name: Configuration Preferences

Description:

The dialog has the following fields and buttons:

Name

Name of the default model preferences configuration. This name is always Configuration Preferences, and cannot be changed.

Description

A description of the default configuration preferences. You can use this field to enter information about the default preferences you have set.

Save Preferences

Click to save the configuration preferences for use with the current MATLAB session and subsequent sessions.

Note When you make changes to the configuration preferences and click **Apply**, they remain in effect during the current MATLAB session. To keep the changes in effect for subsequent sessions, click **Save Preferences**.

Restore to Default Preferences

Click to restore the default configuration settings for creating new models. These underlying defaults are part of Simulink and cannot be changed.

Restore to Saved Preferences

Click to restore the preferences to the settings in effect the last time the preferences were saved. This option overrides any changes that you have made to the preferences since the beginning of the session or since the last time the preferences were restored.

Referencing Configuration Sets

Overview of Configuration References

By default, a configuration set is stored within an individual model, which allows it to be used only by that model. Alternatively, a configuration set can stored independently of any model, which allows it to be used by any or all models.

A configuration set that exists outside any model is called a *freestanding configuration set*. Each model that uses a freestanding configuration set does so by defining a *configuration reference* that points to the configuration set. The result is the same as if the referenced configuration set were stored within the model.

You can use configuration references and freestanding configuration sets to:

- **Assign the same configuration set to any number of models**

  Each model that uses a given configuration set contains a configuration reference that points to a MATLAB variable. The value of that variable is a freestanding configuration set. All of those models then share that configuration set, and changing the value of any parameter in the set changes it for every model that uses the set. This capability is useful for reconfiguring large numbers of submodels quickly, and for ensuring consistent configuration of parent models and referenced models.

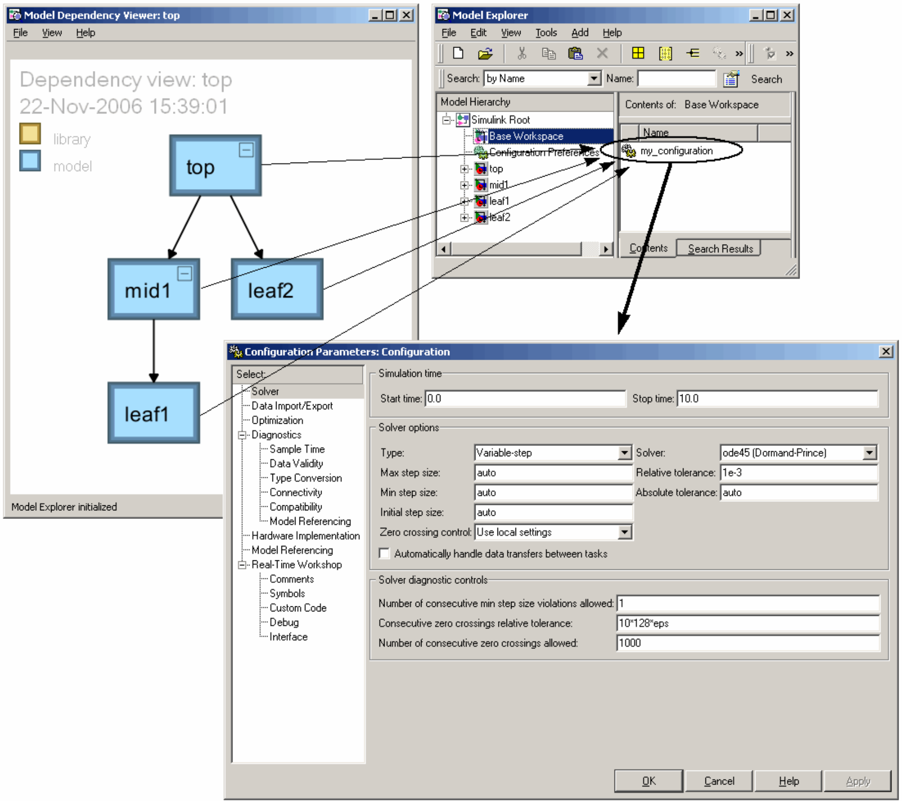
- **Replace the configuration sets of any number of models without changing the model files**

  When multiple models use configuration references to access a freestanding configuration set, assigning a different configuration set to the MATLAB variable specified by the references assigns that set to all the models. This capability allows you to maintain a library of configuration sets and assign them as needed to any number of models in a single operation.

- **Use different configuration sets for a referenced model in different contexts changing the model file**

  The submodel that uses different configuration sets in different contexts contains a configuration reference that specifies a MATLAB variable. Each context assigns an appropriate freestanding configuration set to that variable. When the submodel is used in each context, the value of the variable in that context specifies its configuration set, which can differ in different contexts without requiring changes to the submodel.

The next figure shows one way to use configuration references. Each of the four models represented in the Model Dependency Viewer specifies the configuration reference named `my_reference` as its active configuration set, and `my_reference` points to a freestanding configuration set named `Configuration`. The parameter values in `Configuration` therefore apply to all four models, and any change to any parameter in `Configuration` applies to all four models.



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
□ Automatically handle data transfers between tasks

Solver diagnostic controls

Number of consecutive min step size violations allowed: 1
Consecutive zero crossings relative tolerance: 10*128*eps
Number of consecutive zero crossings allowed: 1000

To use a configuration reference to link a freestanding configuration set to a model, you:

1. Create or obtain a configuration set and store it in the base workspace as the value of a MATLAB variable.

2. Create a configuration reference that specifies the relevant MATLAB variable.

3. Attach the configuration reference to the model just as you would attach a configuration set.

4. Activate the reference just as you would activate a configuration set stored within the model.

5. Access, set, and change configuration set parameters and the MATLAB variable as needed.

A configuration reference is implemented as an object of type `Simulink.ConfigSetRef`, and a configuration set is an object of type `Simulink.ConfigSet` The two classes are similar in many ways. Wherever the same operation is applicable to both, the relevant functions and methods are overloaded to work with either class. For example, you can `attach` or `activate` a configuration set or a configuration reference using the same GUI operations and API syntax.

You cannot nest configuration references: only one level of indirection is available. You can obtain configuration parameter values by operating on a configuration reference just as if it were the configuration set that it references. See "Getting Values from a Referenced Configuration Set" on page 11-62 for details. General information about configuration sets appears in "Configuration Sets" on page 11-37.

Creating a Freestanding Configuration Set

All freestanding configuration sets are stored in the base workspace as the values of base workspace variables. Although you can store a configuration set in a model and point to it with a base workspace variable, such a configuration set would not be freestanding; trying to use it in a configuration reference would cause an error. You can store a freestanding configuration set in the base workspace in these ways:

- Create and populate a new configuration set.
- Copy a configuration set that is stored in a model.
- Load a configuration set that was saved in a MAT-file.

You can store any number of configuration sets in the base workspace, assigning each to a different variable. You can use any technique to manipulate a freestanding configuration set and its parameter values that you could use with a configuration set stored directly in a model.

Creating and Populating a New Configuration Set

You can create a new configuration set in the base workspace as follows:

```
cset = Simulink.ConfigSet
```

where cset is a new or existing base workspace variable. The new configuration set initially has default parameter values, copied from the default configuration set.

Copying a Configuration Set Stored in a Model

You can copy an existing configuration set to the base workspace using any of the techniques described in “Configuration Sets” on page 11-37, and assign the set to a MATLAB variable. For example:

```
cset = copy (getActiveConfigSet(mdl))
cset = copy (getConfigSet(mdl, ConfigSetName))
```

where mdl is any open model, and ConfigSetName is the name of any configuration set attached to the model. The first example obtains the currently active configuration set. The second example obtains a configuration set by specifying the name under which it appears in the Model Explorer.

Be sure to copy any configuration set obtained from an existing model, as shown in the examples. Otherwise, cset will refer to the existing configuration set stored in the model, rather than a new freestanding configuration set in the base workspace, and any use of a configuration references that links to cset will cause an error.

Reading a Configuration Set from a MAT-File

To use a freestanding configuration set across multiple MATLAB sessions, you can save it into a MAT-file. To create the MAT-file, you first copy the configuration set to a base workspace variable, as previously described, then save the variable to the MAT-file:

```
save (workdir/ConfigSetName.mat, cset)
```

where workdir is a working directory, ConfigSetName.mat is the name of the MAT-file, and cset is a base workspace variable whose value is the configuration set to be saved. When you later reopen your model, you can reload the configuration set into the variable:

```
load (workdir/ConfigSetName.mat)
```

To execute code that reads configuration sets from MAT-files, you can use the pre-load function of a top model, the MATLAB startup script, and various other techniques, including entering the `load` statement(s) interactively. Any technique that executes the necessary code will work.

Creating and Attaching a Configuration Reference

Once you have stored a configuration set in the base workspace, as described in “Creating a Freestanding Configuration Set” on page 11-52, you can link to that configuration set from a configuration reference, and attach the reference to a model. The model then has the same configuration parameters that it would if the referenced configuration set were stored directly in the model. You can attach any number of configuration references to a model. Each must have a unique name.

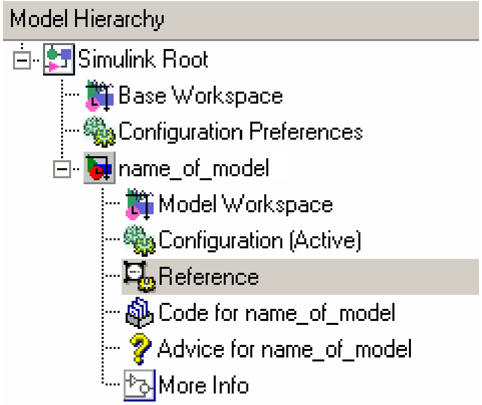
GUI Techniques

To create a configuration reference using the GUI:

1 In the Model Explorer, select the model to which the configuration reference will be attached.

2 Click the **Add Reference** tool or choose **Add > Configuration Reference**.

   A new configuration reference attaches to the selected model. The default name of the new reference is `Reference`, with a digit appended if necessary to prevent name conflict. The name of the configuration reference appears in the Model Hierarchy pane under the Model Workspace icon, below the names of any configuration sets.

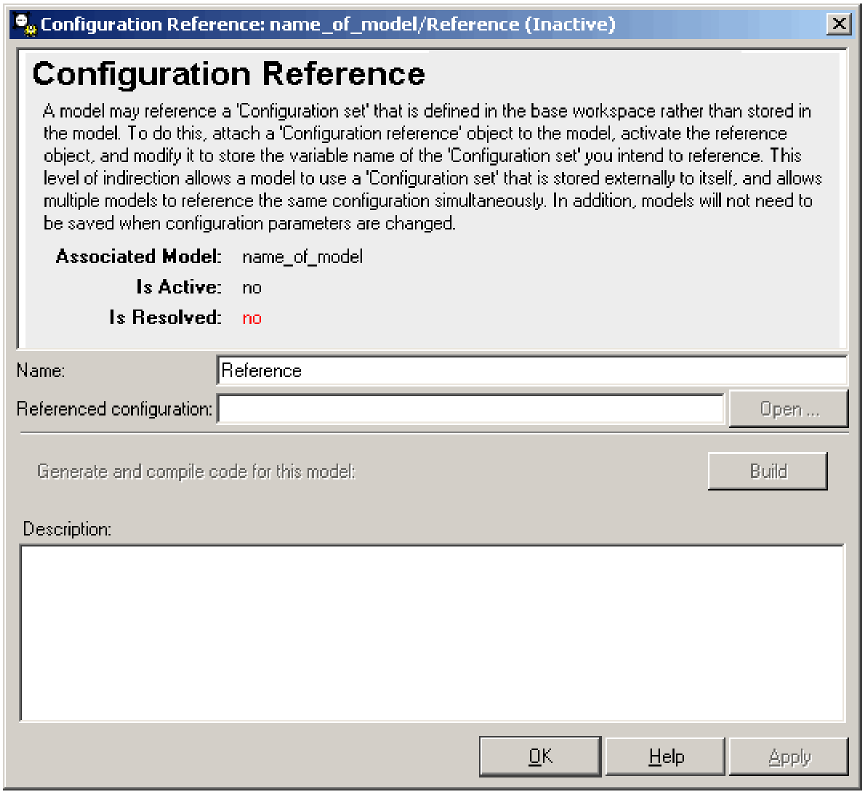


Model Hierarchy

- Simulink Root
  - Base Workspace
  - Configuration Preferences
  - name_of_model
    - Model Workspace
    - Configuration (Active)
    - Reference
    - Code for name_of_model
    - Advice for name_of_model
    - More Info

3 Select the new configuration reference in the Model Hierarchy pane, or right-click the configuration reference and choose **Open** from the context menu.

The Configuration Reference dialog appears in the Dialog pane or a separate window.



Configuration Reference: name_of_model/Reference (Inactive)

Configuration Reference

A model may reference a 'Configuration set' that is defined in the base workspace rather than stored in the model. To do this, attach a 'Configuration reference' object to the model, activate the reference object, and modify it to store the variable name of the 'Configuration set' you intend to reference. This level of indirection allows a model to use a 'Configuration set' that is stored externally to itself, and allows multiple models to reference the same configuration simultaneously. In addition, models will not need to be saved when configuration parameters are changed.

Associated Model: name_of_model

Is Active: no

Is Resolved: no

Name: Reference

Referenced configuration:

Generate and compile code for this model:

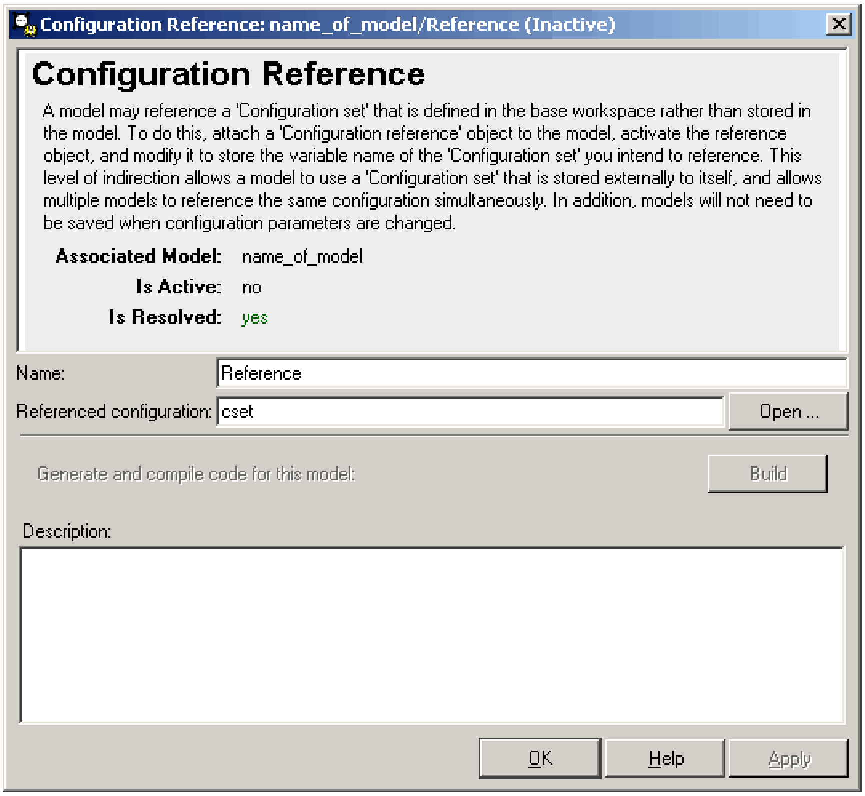
Description:

4 Change the default **Name** if desired. This name exists for human readability, and does not affect the configuration reference functionally.

5 Specify the **Referenced configuration** set to be the base workspace variable whose value is the freestanding configuration set that you want to reference. Be careful not to specify the name of a configuration reference. Configuration references cannot be nested, and an error will result.

6 Click **OK** or **Apply**.

The **Is Resolved** field in the dialog changes to yes.



Configuration Reference: name_of_model/Reference (Inactive)

Configuration Reference

A model may reference a 'Configuration set' that is defined in the base workspace rather than stored in the model. To do this, attach a 'Configuration reference' object to the model, activate the reference object, and modify it to store the variable name of the 'Configuration set' you intend to reference. This level of indirection allows a model to use a 'Configuration set' that is stored externally to itself, and allows multiple models to reference the same configuration simultaneously. In addition, models will not need to be saved when configuration parameters are changed.

Associated Model: name_of_model

Is Active: no

Is Resolved: yes

Name: Reference

Referenced configuration: cset

Generate and compile code for this model:

Description:

If you do not specify a valid **Referenced configuration**, Simulink posts a warning. Any attempt to use a configuration reference that lacks a valid **Referenced configuration** generates an error. The API equivalent of **Referenced configuration** is `WSVarName`. You can later use the GUI or API to correct the specification or provide a configuration set with the correct name. See "Unresolved Configuration References" on page 11-61 for more information.

API Techniques

To create and populate a configuration reference using the API:

1 Create the reference:

```
cref = Simulink.ConfigSetRef
```

2 Change the default name if desired:

```
cref.Name = 'ConfigSetRefName'
```

3 Specify the referenced configuration set:

```
cref.WSVarName = 'cset'
```

Be careful not to specify the name of a configuration reference. Configuration references cannot be nested, and an error will result.

4 Attach the reference to a model:

```
attachConfigSet(mdl, cref, true)
```

The third argument is optional and authorizes renaming if needed to avoid a name conflict.

Any attempt to use a configuration reference that does not specify a valid `WSVarName` generates an error. The GUI equivalent of `WSVarName` is **Referenced configuration**. You can later use the API or GUI to correct the reference or provide a configuration set that has the correct name. See “Unresolved Configuration References” on page 11-61 for more information.

Obtaining a Configuration Reference Handle

Most functions and methods that operate on a configuration reference take a handle to the reference. If you have created a configuration reference programmatically, with a statement like

```
cref = Simulink.ConfigSetRef
```

the variable *`cref`* contains a handle to the reference. If you do not already have a handle, you can obtain one by executing:

```
cref = getConfigSet(mdl, 'ConfigSetRefName')
```

where *`ConfigSetRefName`* is the name of the configuration reference as it appears in the Model Explorer, e.g., `Reference`. This is the name you specified by setting the **Name** field in the Configuration Reference dialog or executing

```
cref.Name = 'ConfigSetRefName'
```

The technique for obtaining a configuration reference handle is the same as you would use to obtain a configuration set handle. Wherever the same operation applies to both configuration sets and configuration references, applicable functions and methods are overloaded to perform correctly with either class.

Attaching a Configuration Reference to Additional Models

After you have created a configuration reference and attached it to a model, you can attach copies of the reference to any number of additional models. To create and attach a copy of a configuration reference, you can use any GUI or API technique that you could use to copy and attach a configuration set, such as dragging, pasting, or a function like `attachConfigSetCopy`. See "Copying, Deleting, and Moving Configuration Sets" on page 11-40 and "Configuration Set API" on page 11-42.

Models do not share configuration reference objects. Each model has its own copy of any configuration reference attached to it, just as each has its own copy of any attached configuration set. Configuration references in different models establish configuration set sharing by specifying the same base workspace variable, which links the various models to the same freestanding configuration set.

If you use the GUI, attaching an existing configuration reference to an additional model automatically attaches a copy, as distinct from a handle to the original. If necessary to prevent name conflict, the GUI will add or increment a digit at the end of the copied reference's name. If you use the API, be sure to explicitly copy the configuration reference before attaching it, with statements like:

```
cref = copy (getConfigSet(mdl, ConfigSetName))
attachConfigSet (cref, mdl, true)
```

If you omit the `copy` operation, *`cref`* will be a handle to the original configuration reference, rather than a copy of the reference, and any attempt to use *`cref`* will cause an error. If you omit the argument `true` to `attachConfigSet`, the operation will fail if it would cause a name conflict.

The following example shows code for obtaining a freestanding configuration set and attaching references to it to two models. After the code executes, one of the models contains both an internal configuration set and a configuration reference that points to a freestanding copy of that configuration set. If the internal copy is superfluous, it can be removed with `detachConfigSet`, as shown in the last line of the example.

```
% Get copy of original config set as a variable in the base workspace
open_system('mdl1')
cset = getConfigSet('mdl1', 'Configuration')  % Get handle to local cset
cset1 = copy(cset)  % Create freestanding copy; original remains in model

% In the original model, create a configuration reference to the cset cop
cref1 = Simulink.ConfigSetRef
cref1.WSVarName = 'cset1'
attachConfigSet('mdl1', cref1, true)   % Rename if name conflict occurs

% In a second model, create a configuration reference to the same cset
open_system('mdl2')
attachConfigSetCopy('mdl2', cref1, true)  % Rename if name conflict occur
detachConfigSet('mdl1', 'Configuration')  % Delete original cset from fir
```

Changing a Configuration Reference

You can change an existing configuration reference as needed by reopening its Configuration Reference dialog and changing its **Name** or **Referenced configuration**. Similarly, you can use the API on an existing configuration reference to change its `Name` or `WSVarName`. If you refer to a configuration set that does not yet exist, no error occurs, but the configuration reference is unusable. The configuration reference becomes usable as soon as the configuration set exists.

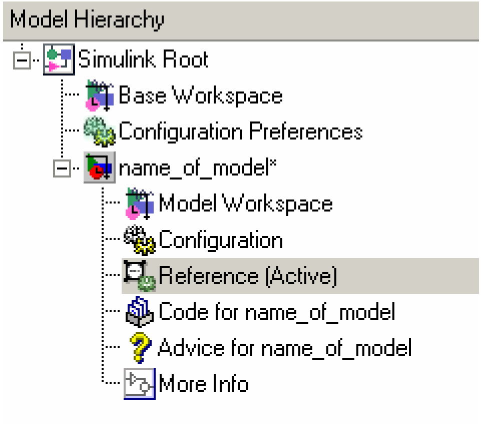
Activating a Configuration Reference

Once you have created a configuration reference and attached it to a model, you can activate it using any technique that would activate a configuration set stored in the model, such as:

- From the GUI, choose **Activate** from the configuration reference's context menu.

- From the API, execute setActiveConfigSet, specifying the configuration reference as the first argument.

When a configuration reference is active, the **Is Active** field of the Configuration Reference dialog is `yes`, and the Model Explorer shows the name of the reference suffixed with `(Active)`.



The freestanding configuration set to which the active reference points now provides the configuration parameters for the model containing the reference.

Unresolved Configuration References

When a configuration reference does not specify a valid configuration set, the configuration reference is unresolved, and the **Is Resolved** field of its Configuration Reference dialog has the value `no`. If you activate an unresolved configuration reference, no warning or error occurs. However, an unresolved configuration reference that is Active provides no configuration parameter values to the model. Therefore:

- Fields that display values that can be known only by accessing a configuration parameter, like Stop Time in the model window, are blank.
- Trying to build the model, simulate it, generate code for it, or otherwise require it to access configuration parameter values, causes an error.

"Creating and Attaching a Configuration Reference" on page 11-54 describes techniques for resolving configuration references.

Getting Values from a Referenced Configuration Set

You can use `get_param` on a configuration reference to obtain parameter values from the linked configuration set, just as if the reference object were the configuration set itself. Simulink retrieves the referenced configuration set and performs the indicated `get_param` on it.

For example, if configuration reference *`cref`* links to configuration set *`cset`*, the following operations give identical results:

```
get_param (cset, 'StopTime')
get_param (cref, 'StopTime')
```

Changing Values in a Referenced Configuration Set

You cannot change a configuration set in any way, including changing configuration parameter values, by operating on a configuration reference. Thus, with `cref` and `cset` as above, if you executed:

```
set_param (cset, 'StopTime', 300)
set_param (cref, 'StopTime', 300)          % ILLEGAL
```

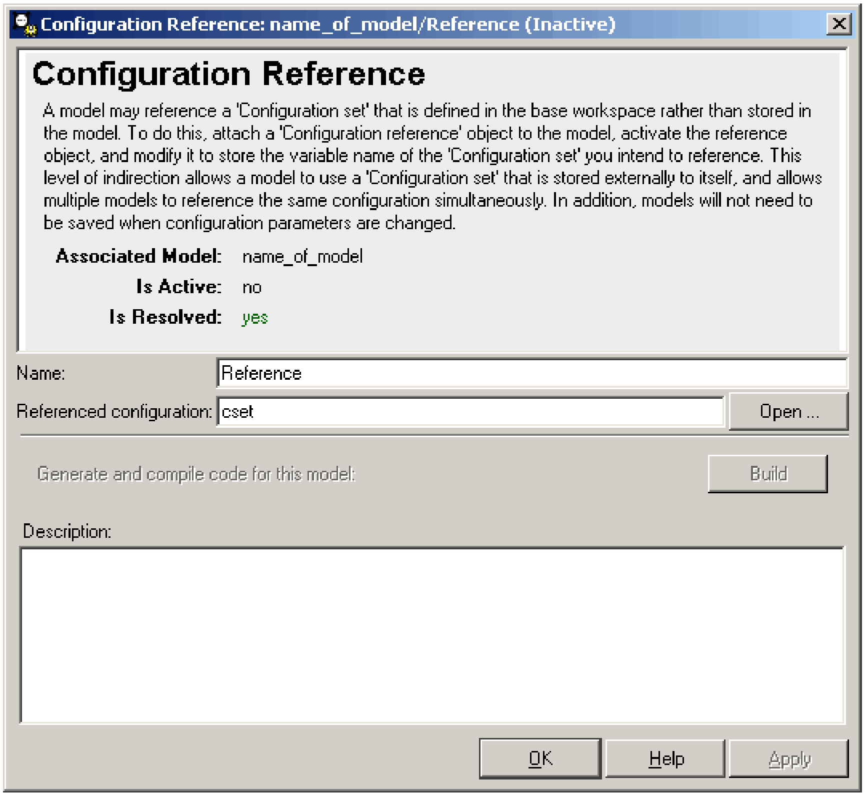
the first operation would succeed, but the second would cause an error. Instead, you must obtain the configuration set itself and change it directly, using the GUI or the API.

GUI Techniques

To obtain a referenced configuration set using the GUI:

1. Select the configuration reference in the Model Hierarchy pane, or right-click the configuration reference and choose **Open** from the context menu.

   The Configuration Reference dialog appears in the Dialog pane or a separate window.



Configuration Reference: name_of_model/Reference (Inactive)

Configuration Reference

A model may reference a 'Configuration set' that is defined in the base workspace rather than stored in the model. To do this, attach a 'Configuration reference' object to the model, activate the reference object, and modify it to store the variable name of the 'Configuration set' you intend to reference. This level of indirection allows a model to use a 'Configuration set' that is stored externally to itself, and allows multiple models to reference the same configuration simultaneously. In addition, models will not need to be saved when configuration parameters are changed.

Associated Model: name_of_model

Is Active: no

Is Resolved: yes

Name: Reference

Referenced configuration: cset

Generate and compile code for this model:

Description:

2 Click **Open** to the right of the **Referenced configuration** field.

The Configuration Parameters dialog box opens on the configuration set specified by **Referenced configuration**. You can now change and apply or save parameter values as you would for any configuration set.

API Techniques

To obtain a referenced configuration set using the API:

1 Obtain a handle to the configuration reference, as described in "Obtaining a Configuration Reference Handle" on page 11-58.

2 Obtain the configuration set `cset` from the configuration reference *`cref`*:

```
cset = cref.getRefConfigSet
```

You can now use set_param on *cset* to change parameter values. Example:

```
set_param (cset, 'StopTime', 300)
```

If you want to change parameter values through the GUI, execute:

```
cset.openDialog
```

The Configuration Parameters dialog box opens on the specified configuration set.

Replacing a Referenced Configuration Set

You can completely replace the base workspace variable and configuration set that a configuration reference uses. However, the pointer from the configuration reference to the configuration set becomes stale. You must then execute:

```
cref.refresh
```

where *cref* is the configuration reference. If you do not execute refresh, the configuration reference will continue to use the previous instance of the base workspace variable and its configuration set. This example illustrates the problem.

```
cset1=Simulink.ConfigSet;             % Create a new configuration set
set_param (cset1, 'StopTime', 500)    % Set a non-default stop time
cref1=Simulink.ConfigSetRef;          % Create a new config reference
cref1.WsVarName='cset1';              % Resolve the config ref to the set
attachConfigSet('mdl1', cref1, true)  % Attach the config ref to a model
cset1=Simulink.ConfigSet;             % Replace config set on base worksp
% cset1.refresh;                      % Call to refresh is commented out
set_param (cset, 'StopTime', 75)      % Set a different stop time
```

If you simulate the model, it will stop at 500, not at 75. Calling cset1.refresh where shown prevents the problem.

Building Models and Generating Code

The Real-Time Workshop pane of the Configuration Parameters dialog contains a **Build** button. Its availability differs depending on whether the configuration set displayed by the dialog is stored in a model or is a freestanding configuration set.

- When the pane displays a configuration set stored in a model, the **Build** button is enabled, and you can use it to generate and compile code for the model.
- When the pane displays a freestanding configuration set, the **Build** button is disabled because the configuration set does not know which (if any) models are linked to it.

To provide the same capabilities whether a configuration set is stored in a model or is freestanding, the Configuration Reference dialog contains a **Build** button. This button has the same effect as its equivalent in the Configuration Parameters dialog, and operates on the model that contains the configuration reference.

Configuration Reference Limitations

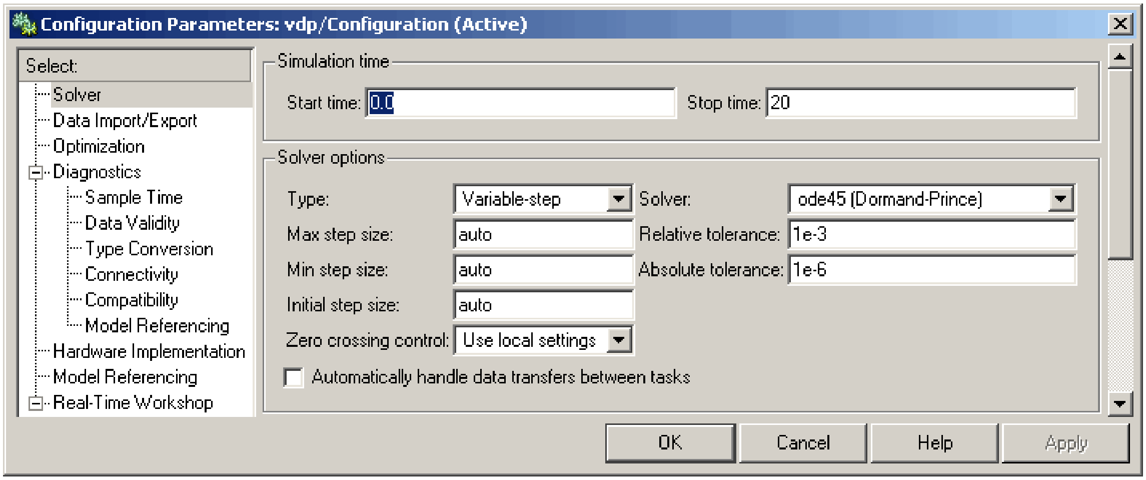
- You cannot nest configuration references: only one level of indirection is available, so a configuration reference cannot link to another configuration reference; it must specify a freestanding configuration set.
- If you replace the base workspace variable and configuration set that a configuration reference uses, you must then execute `refresh` for every configuration reference that uses the replaced variable and set. See “Replacing a Referenced Configuration Set” on page 11-64.
- If you activate a configuration reference when using a custom target, the `ActivateCallback` function does not get triggered to notify the corresponding freestanding configuration set. Likewise, if a freestanding configuration set switches from one target to another, the `ActivateCallback` does not get triggered to notify the new target, even if an active configuration reference points to that target. For more information about `ActivateCallback` functions, see “rtwgensettings Structure” in the Real-Time Workshop Embedded Coder Developing Embedded Targets documentation.

Configuration 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 11-37).

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 10-2 for more information.

To display the dialog box, select **Configuration Parameters** from the model editor's **Simulation** or context menu, or press **Ctrl+E**. The dialog box appears.



Configuration Parameters: vdp/Configuration (Active)

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

□ Automatically handle data transfers between tasks

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 set specific configuration parameters:

- "Solver Pane" on page 11-68
- "Data Import/Export Pane" on page 11-81

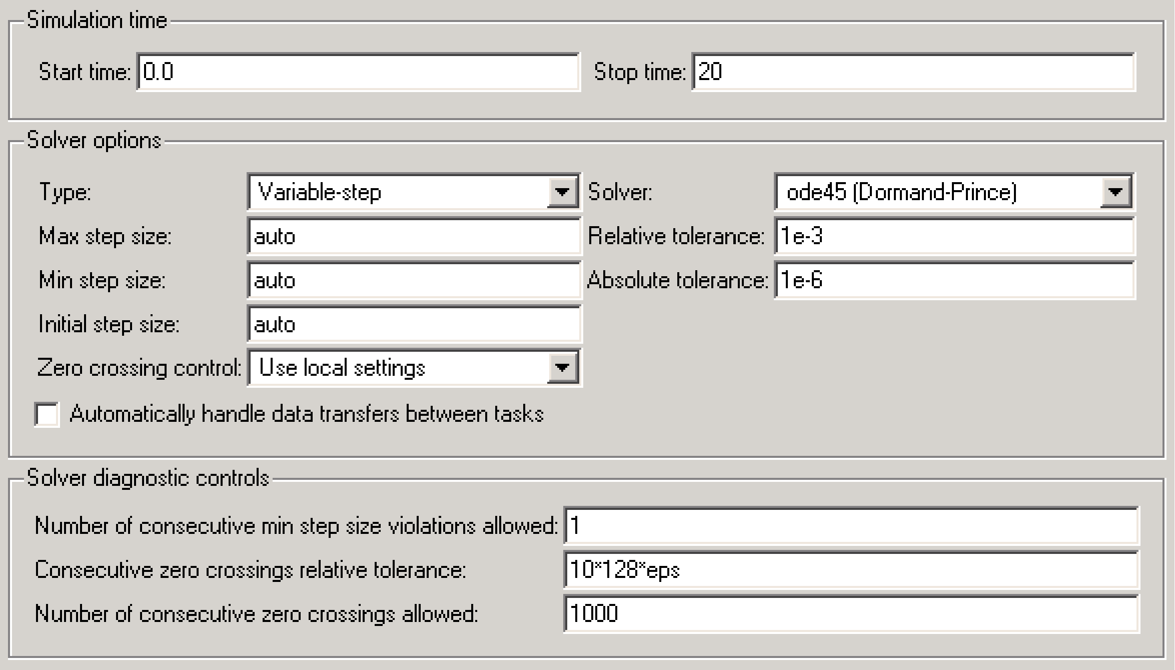
- “Optimization Pane” on page 11-86
- “Diagnostics Pane” on page 11-102
- “Hardware Implementation Pane” on page 11-135
- “Model Referencing Pane” on page 11-139

In most cases, Simulink does not immediately apply a change that you have made on the dialog box. 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. See “Model Parameters” in the online Simulink reference for descriptions of these parameters. The description for each parameter specifies 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.

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

□ Automatically handle data transfers between tasks

Solver diagnostic controls

Number of consecutive min step size violations allowed: 1

Consecutive zero crossings relative tolerance: 10*128*eps

Number of consecutive zero crossings allowed: 1000

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.

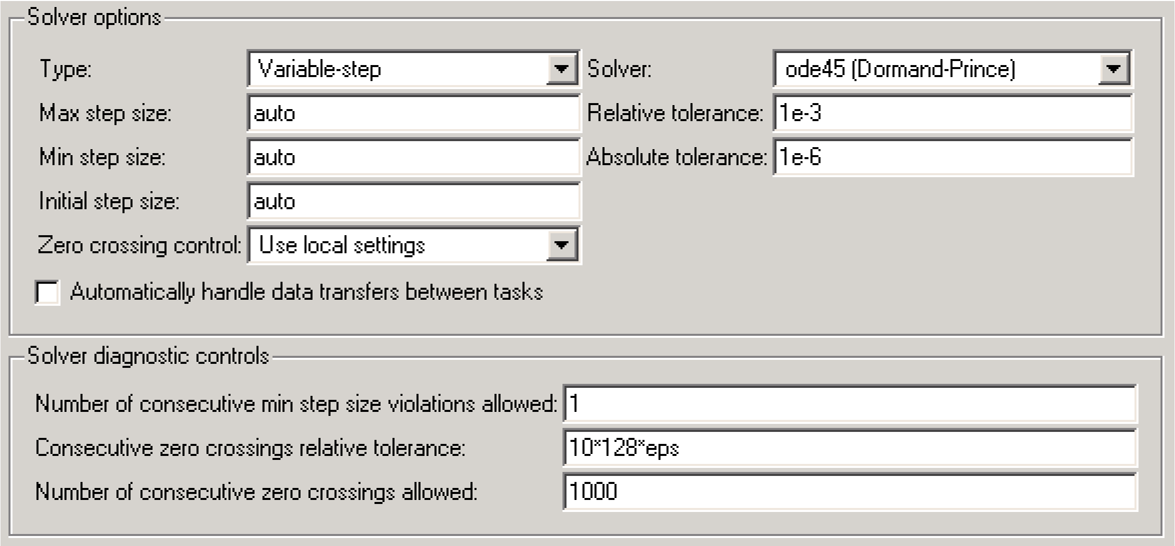
Start time. Specifies the simulation start time. The default start time is 0.0 seconds.

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.



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

□ Automatically handle data transfers between tasks

Solver diagnostic controls

Number of consecutive min step size violations allowed: 1

Consecutive zero crossings relative tolerance: 10*128*eps

Number of consecutive zero crossings allowed: 1000

The contents of the group depends on the solver type.

General Solver Options

The following 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 11-11 and “Improving Simulation Performance and Accuracy” on page 11-149 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 11-12 and "Choosing a Variable-Step Solver" on page 11-17 for information on how to choose the solvers listed in the **Solver** list.

Automatically handle data transfers between tasks. If checked, this option causes Simulink to insert hidden Rate Transition blocks where rate transitions occur between blocks. Simulink adds these blocks configured to

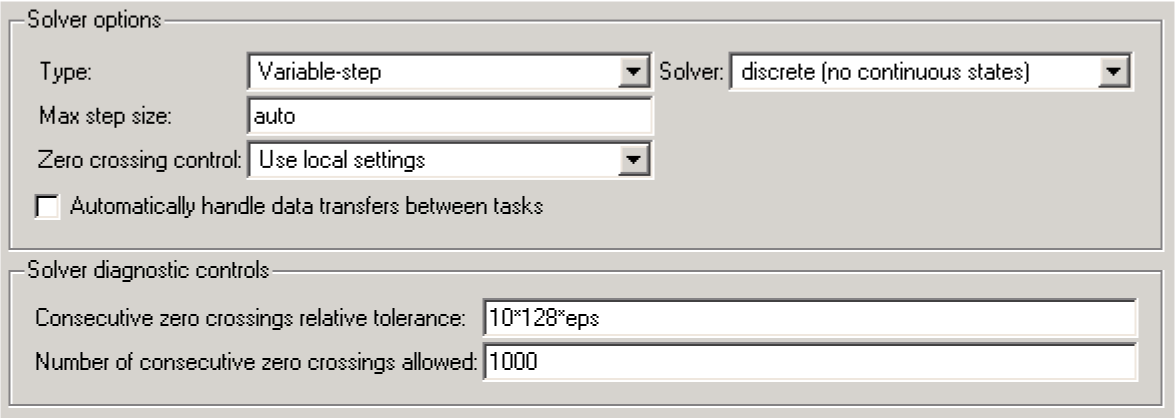
- Ensure data integrity during data transfer
- Ensure deterministic data transfer

See "Rate Transition Block Options" in the Real-Time Workshop documentation for further details.

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

□ Automatically handle data transfers between tasks

Solver diagnostic controls

Consecutive zero crossings relative tolerance: 10*128*eps

Number of consecutive zero crossings allowed: 1000

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 1-20 for more information.

You can override this optimization on a block-by-block basis for the following types of blocks:

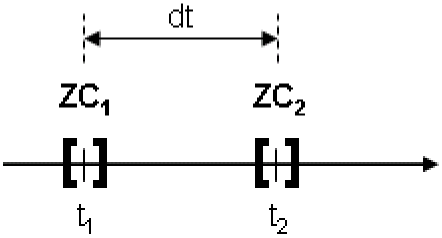
Abs	Integrator	Step
Backlash	MinMax	Switch
Dead Zone	Relay	Switch Case
Enable	Relational Operator	Trigger
Hit Crossing	Saturation	
If	Sign	

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.

The next two options appear if you select either `Use local settings` or `Enable all` as the **Zero crossing control** option.

Consecutive zero crossings relative tolerance. Factor that controls how closely zero-crossing events must occur to be considered consecutive. See "Number of consecutive zero crossings allowed" on page 11-72 for more information.

Number of consecutive zero crossings allowed. Simulink defines zero crossings as consecutive if the time between events is less than a particular interval. The following figure depicts a simulation timeline during which Simulink detects zero crossings ZC_1 and ZC_2, bracketed at successive time steps t_1 and t_2.



Simulink determines that the zero crossings are consecutive if

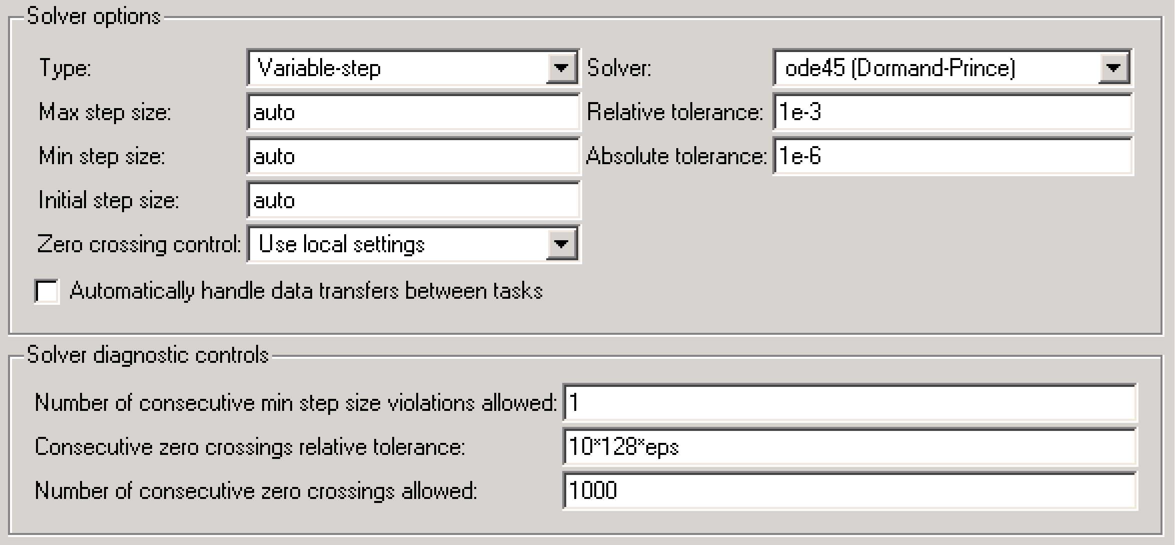
```
dt < RelTolZC * t2
```

where `dt` is the time between zero crossings and `RelTolZC` is the **Consecutive zero crossings relative tolerance** (see "Consecutive zero crossings relative tolerance" on page 11-71).

Simulink counts the number of consecutive zero crossings that it detects. If the count exceeds the value of **Number of consecutive zero crossings allowed**, Simulink displays either a warning or error as specified by the **Consecutive zero crossings violation** diagnostic (see "Consecutive zero crossings violation" on page 11-105). Simulink resets the counter each time it detects nonconsecutive zero crossings (i.e., successive zero crossings that fail to meet the aforementioned condition).

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

□ Automatically handle data transfers between tasks

Solver diagnostic controls

Number of consecutive min step size violations allowed: 1

Consecutive zero crossings relative tolerance: 10*128*eps

Number of consecutive zero crossings allowed: 1000

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 11-85.

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 **Min step size violation** diagnostic to `error` on the **Diagnostics** pane (see “Min step size violation” on page 11-104). 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 11-19).

Absolute tolerance. Absolute tolerance for this solver (see “Specifying Variable-Step Solver Error Tolerances” on page 11-19).

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.

Solver reset method. This option appears only if you select one of the following solvers:

- ode15s
- ode23t
- ode23tb

Its setting controls the solver behavior at solver reset (e.g., after detecting a zero crossing) as follows:

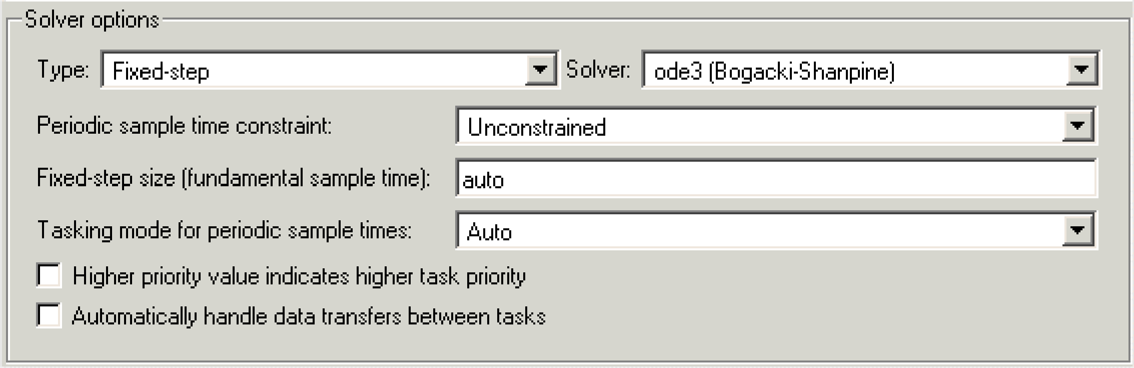
Setting	Reset Behavior
Robust	The solver recomputes the Jacobian matrix needed by the integration step at every solver reset.
Fast	The solver does not recompute the Jacobian matrix at a solver reset.

The fast setting speeds simulation. However, it can result in incorrect solutions in some cases. If you suspect that the simulation is giving incorrect results, try the robust setting. If there is no difference in simulation results between the fast and robust settings, revert to the fast setting.

Number of consecutive min step size violations allowed. Specifies the maximum number of consecutive minimum step size violations allowed during simulation. A minimum step size violation occurs when a variable-step continuous solver takes a smaller step than that specified by the **Min step size** property (see “Min step size” on page 11-74). Simulink counts the number of consecutive violations that it detects. If the count exceeds the value of **Number of consecutive min step size violations allowed**, Simulink displays either a warning or error message as specified by the **Min step size violation** diagnostic (see “Min step size violation” on page 11-104).

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 satisfies 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 11-77 for a description of this field.

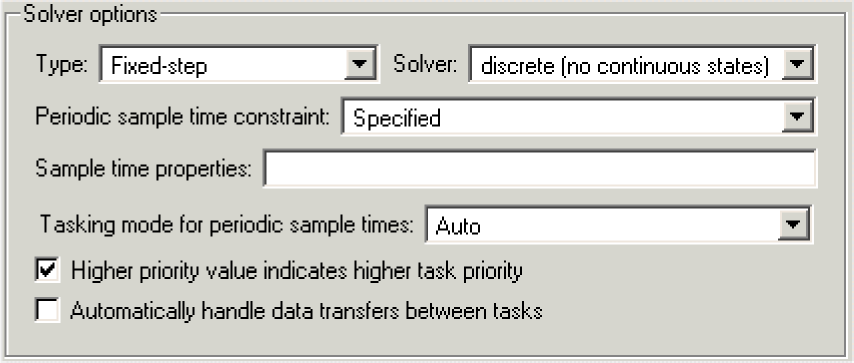
- `Ensure sample time independent`

  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 11-77).

- `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 1-41), 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 three sample times, 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 11-80).

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 11-80), 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. Use this option with sample time diagnostics to specify whether Simulink should check for illegal rate transitions and the action it should take if it detects a transition. The options are

- `MultiTasking`

  This mode causes Simulink, at the beginning of a simulation or when you update the model, to check for the existence of illegal rate transitions between blocks, that is, direct connections between blocks operating at different sample rates. If it detects such a transition, Simulink either halts the simulation or update and displays an error message or prints a warning at the MATLAB command line, depending on the setting of the Simulink multitask rate transition diagnostic (see "Multitask rate transition" on page 11-109).

  Note Use this option for models of real-time multitasking systems to ensure detection of illegal rate transitions between tasks that can result in a task's output not being available when needed by another task. You can then use Rate Transition blocks to eliminate such illegal rate transitions from the model. See "Models with Multiple Sample Rates" in the Real-Time Workshop documentation for more information.

- `SingleTasking`

  This mode checks for illegal rate transitions among blocks only if the Simulink single-task rate transition diagnostic (see “Single task rate transition” on page 11-109) is set to `Error` or `Warning`. By default, the diagnostic is set to `None`. 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.

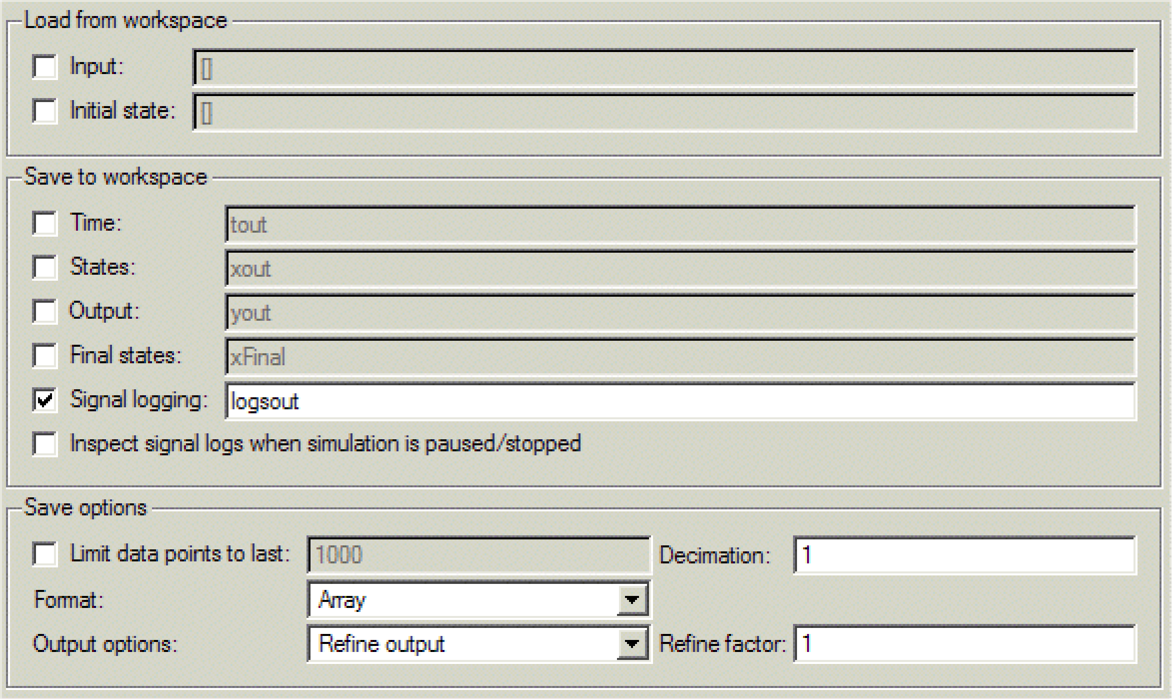
The next two options appear only if you select the `ode14x` solver (see “Implicit Fixed-Step Continuous Solvers” on page 11-16).

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 11-37) in the Model Explorer and display the configuration’s **Data Import/Export** subset.



Load from workspace

□ Input: []

□ Initial state: []

Save to workspace

□ Time: tout

□ States: xout

□ Output: yout

□ Final states: xFinal

☑ Signal logging: logsout

□ Inspect signal logs when simulation is paused/stopped

Save options

□ Limit data points to last: 1000 Decimation: 1

Format: Array

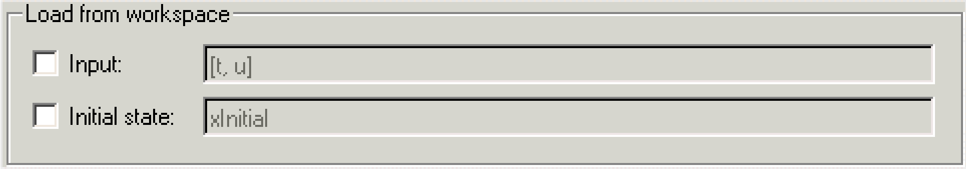
Output options: Refine output Refine factor: 1

This pane includes the following groups of options.

- “Load from workspace” on page 11-82
- “Save to workspace” on page 11-82
- “Save options” on page 11-83

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

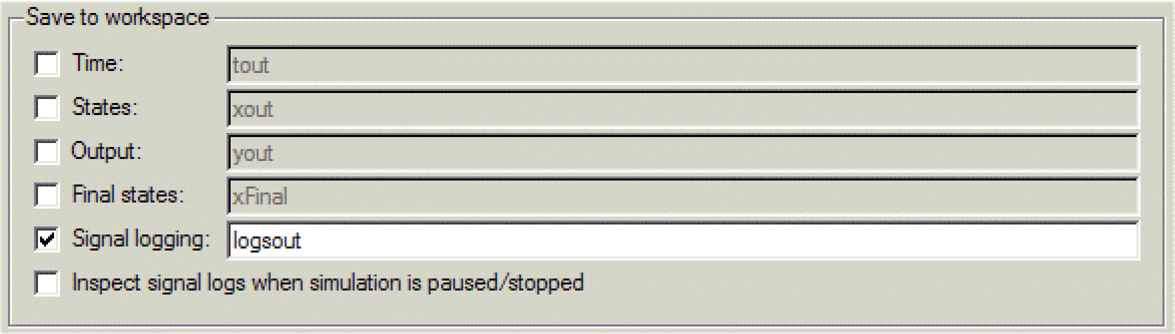
It includes the following controls.

Input. A MATLAB expression that specifies the data to be imported from the MATLAB workspace. See “Importing Data from the MATLAB Workspace” on page 11-21 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 11-31 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

☑ Signal logging: logsout

□ Inspect signal logs when simulation is paused/stopped

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 11-31 for more information.

Output. Name of the MATLAB variable to be used to store signal data exported during this simulation. See "Exporting Data to the MATLAB Workspace" on page 11-27 for more information.

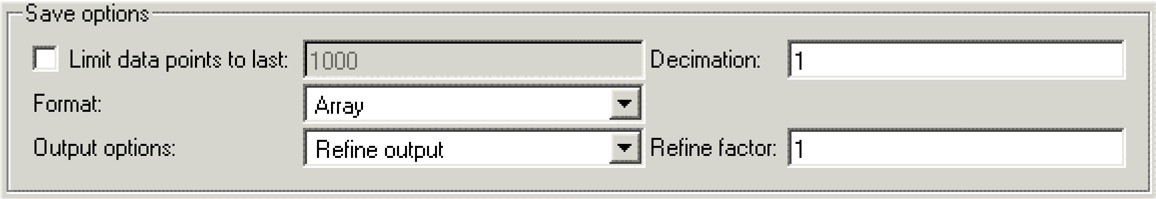
Final states. Specifies the name of a MATLAB variable to be used to store the values of this model's states at the end of a simulation. See "Importing and Exporting States" on page 11-31 for more information.

Signal logging. Globally enables or disables signal logging for this model. The adjacent edit field specifies the name of the signal logging object used to record logged signal data in the MATLAB workspace. See "Logging Signals" on page 5-34.

Inspect signals when simulation is stopped/paused. Checking this option causes Simulink to display logged signals in the MATLAB **Time Series Tools** viewer at the end of a simulation or whenever you pause the simulation. If this option is unchecked, you must select **Tools > Inspect logged signals** from the model editor's menu bar to display logged signals in the **Time Series Tools** viewer.

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

Output options: Refine output

Refine factor: 1

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 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 11-21 for more information on these formats.

Output options. Options for generating additional output signal data.

Note These options appear only if the model specifies a variable-step solver (see “Solver Pane” on page 11-68).

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 beside this control (see “Refine factor” on page 11-85). Use this field to specify the number of points to generate between simulation time steps. For more information, see “Refining Output” on page 11-34.

- `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 output.

For additional information on how Simulink calculates outputs for these three options, see “Specifying Output Options”.

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 11-34.

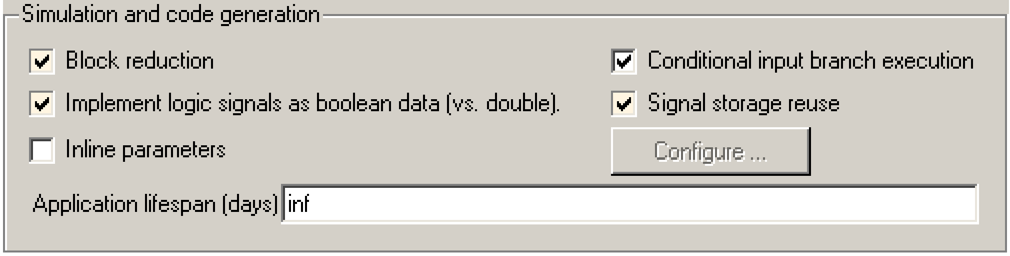
Note Simulink ignores this option for discrete models. This is because the value of data between time steps is undefined for discrete models.

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.

Note Discrete models define outputs only at major time steps. Therefore, Simulink logs output for discrete models only at major time steps. If the **Output times** field specifies other times, Simulink displays a warning message at the MATLAB command line.

Optimization Pane

The **Optimization** pane allows you to select various options that improve simulation performance and the performance of code generated from this model. This pane contains a panel of optimizations that apply both to simulation and to code generated from the model.



Simulation and code generation

☑ Block reduction

☑ Conditional input branch execution

☑ Implement logic signals as boolean data (vs. double).

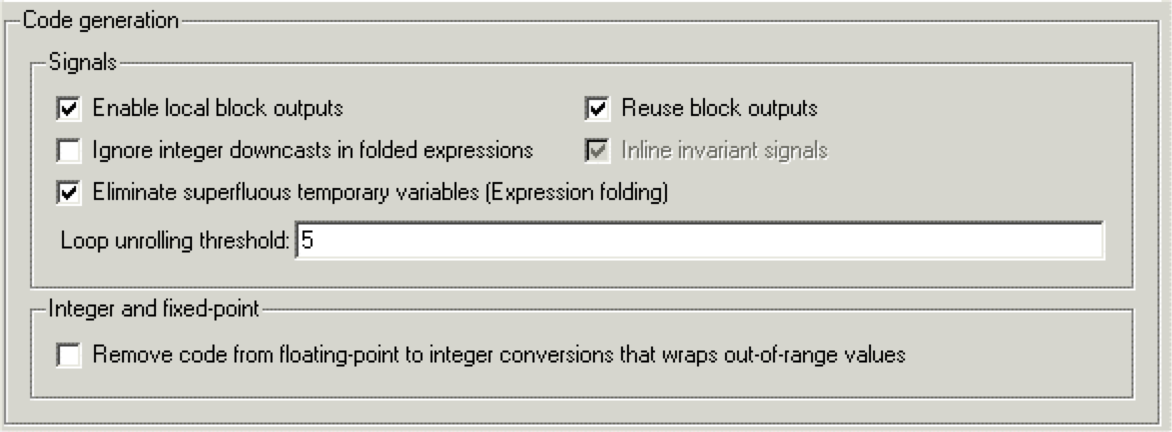
☑ Signal storage reuse

□ Inline parameters

Application lifespan (days) inf

- “Block reduction” on page 11-88
- “Conditional input branch execution” on page 11-89
- “Inline parameters” on page 11-90
- “Implement logic signals as boolean data (vs. double)” on page 11-92
- “Signal storage reuse” on page 11-92
- “Application lifespan (days)” on page 11-93

When Real-Time Workshop is installed on your system, this pane also contains a panel of optimizations that apply only to code generation.



Code generation

Signals

☑ Enable local block outputs

☑ Reuse block outputs

□ Ignore integer downcasts in folded expressions

☑ Inline invariant signals

☑ Eliminate superfluous temporary variables (Expression folding)

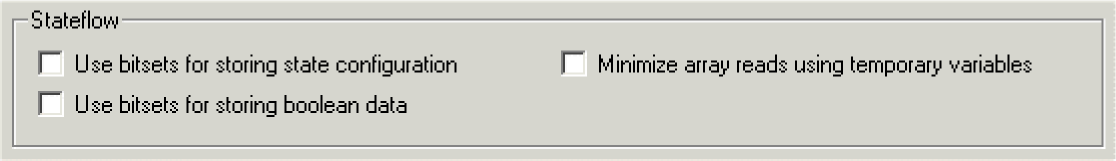
Loop unrolling threshold: 5

Integer and fixed-point

□ Remove code from floating-point to integer conversions that wraps out-of-range values

- "Enable local block outputs" on page 11-94
- "Ignore integer downcasts in folded expressions" on page 11-94
- "Eliminate superfluous temporary variables (Expression folding)" on page 11-94
- "Reuse block outputs" on page 11-94
- "Inline invariant signals" on page 11-95
- "Loop unrolling threshold" on page 11-95
- "Remove code from floating-point to integer conversions that wraps out-of-range values" on page 11-97

The following pane contains Stateflow-related code generation optimizations:



Stateflow

□ Use bitsets for storing state configuration

□ Minimize array reads using temporary variables

□ Use bitsets for storing boolean data

- "Use bitsets for storing state configuration" on page 11-98

- "Use bitsets for storing boolean data" on page 11-98
- "Minimize array reads using temporary variables" on page 11-99

Note These optimizations appear only when Real-Time Workshop and Stateflow are both installed on your system and the model includes Stateflow charts or Embedded MATLAB Function blocks. The settings you make for the Stateflow options also apply to all Embedded MATLAB Function blocks in the model. Note that you do not need a Stateflow license to use Embedded MATLAB Function blocks.

Block reduction

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

This option performs the following kinds of block reduction optimizations.

Accumulator folding. Simulink reduces block diagrams that represent accumulators to a single block.

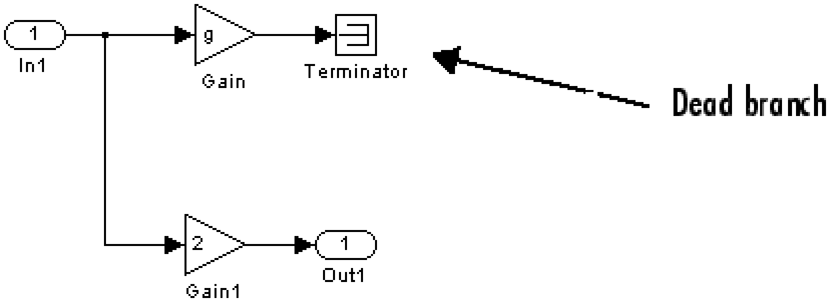
Redundant Type Conversion Removal. Removes unnecessary type conversion blocks. For example, this optimization will remove an `int` type conversion block whose input and output are of type `int`.

Dead Branch Elimination. Eliminates any block that exists on a dead branch of the block diagram, i.e., a branch whose execution does not affect the simulation. A block must meet the following conditions to be considered part of a dead branch:

- The block is in a branch that ends with a block that performs no operation during simulation or in the generated code, for example, a Terminator block or a disabled Assertion block. Note that whether a block performs an operation can depend on whether the model is being simulated or used to generate code or on model settings. For example, a Scope block performs no operation in code generated from a model and hence a branch that ends in a Scope block can be a dead branch for the purposes of code generation, although not for simulation.

- The block is not in any other branch.
- The block does not modify signal storage.

Consider the following model:



The upper branch of this model's block diagram has no effect on the output. Dead branch optimization therefore eliminates the Gain block from the compiled model.

Real-Time Workshop similarly eliminates the code path that includes the dead branch from the code generated for the model:

```
/* Model output function */
static void untitled_output(int_T tid)
{

  /* local block i/o variables */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<Root>/Gain1'
   *  Inport: '<Root>/In1'
   */
  untitled_Y.Out1 = untitled_U.In1 * untitled_P.Gain1_Gain;
}
```

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. See “Expression Folding” in *Real-Time Workshop User’s Guide* for more information.

Inline parameters

By default you can modify (“tune”) many block parameters during simulation (see “Tunable Parameters” on page 1-9). 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.

Note Simulink supports the `off` setting of the inline parameters optimization only for the top model in a model reference hierarchy (see “Model Referencing and the Inline Parameters Optimization” on page 3-69 for more information).

If this option is not selected, Real-Time Workshop generates a global variable declaration for each parameter and uses the variable wherever the generated code needs the parameter's value. User-supplied code can change the value of the parameter at run-time by assigning a value to the variable.

If this option is selected, Real-Time Workshop inserts the actual value of the parameter as a constant expression wherever the generated code needs the value. If the value of a parameter is a constant in the model, the Real-Time Workshop inserts the constant in the generated code. If the value is a workspace variable or MATLAB expression, Real-Time Workshop evaluates the variable or expression and inserts the result as a constant expression in the generated code. User-supplied code cannot change the value of inlined parameters at run-time because they appear as constants in the generated code.

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 11-100). To display the dialog box, click the adjacent **Configure** button. To tune a global parameter, change the value of the corresponding workspace variable and select **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)

Controls the output data type of blocks that generate logic signals. The following sections describe how this optimization affects such blocks.

Note Setting this option off allows the current version of Simulink to run models that were created by earlier versions of Simulink that supported only signals of type `double`. On the other hand, setting this option on reduces the memory requirements of generated code, because a Boolean signal typically requires one byte of storage compared to eight bytes for a `double` signal.

Logical Operator block. This optimization affects only those Logical Operator blocks whose **Output data type mode** parameter specifies `Logical`. If this optimization is enabled, such blocks output a signal of `boolean` data type; otherwise, such blocks output a signal of `double` data type.

Relational Operator block. This optimization affects only those Relational Operator blocks whose **Output data type mode** parameter specifies `Logical`. If this optimization is enabled, such blocks output a signal of `boolean` data type; otherwise, such blocks output a signal of `double` data type.

Combinatorial Logic block. If this optimization is enabled, Combinatorial Logic blocks output a signal of `boolean` data type; otherwise, they output a signal of `double` data type. See Combinatorial Logic in the *Simulink Reference* for an exception to this rule.

Hit Crossing block. If this optimization is enabled, Hit Crossing blocks output a signal of `boolean` data type; otherwise, they output a signal of `double` data type.

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 deselect this option only if 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 how long (in days) an application that contains blocks depending on elapsed or absolute time should be able to execute before timer overflow. Specifying a lifespan, along with the simulation step size, determines the data type used by blocks to store absolute time values. For simulation, setting this parameter to a value greater than the simulation time will ensure time does not overflow. Simulink evaluates this parameter first against the model workspace. If this does not resolve the parameter, Simulink then evaluates it against the base workspace.

The Application lifespan also determines the word size used by timers in the generated code, which can lower RAM usage. For more information, see "Timing Services" in the Real-Time Workshop documentation.

Application lifespan, when combined with the step size of each task, determines the data type used for integer absolute time for each task, as follows:

- If your model does not require absolute time, this option affects neither simulation nor the generated code.
- If your model requires absolute time, this option optimizes the word size used for storing integer absolute time in generated code. This ensures that timers do not overflow within the lifespan you specify. If you set **Application lifespan** to `Inf`, two `uint32` words are used.
- If your model contains fixed-point blocks that require absolute time, this option affects both simulation and generated code.

For example, using 64 bits to store timing data enables models with a step size of 0.001 microsecond (10E-09 seconds) to run for more than 500 years, which would rarely be required. To run a model with a step size of one millisecond (0.001 seconds) for one day would require a 32-bit timer (but it could continue running for 49 days).

Enable local block outputs

Causes the generated code to declare block output signals as local variables if possible. If this option is not selected or it is not possible to declare an output as a local variable, the generated code declares the output as a global variable.

Note The check box for this option is enabled only if signal storage reuse is selected (see "Signal storage reuse" on page 11-92).

See "Signal Storage, Optimization, and Interfacing" in *Real-Time Workshop User's Guide* for more information.

Ignore integer downcasts in folded expressions

This option specifies how Real-Time Workshop should handle 8-bit operations on 16-bit microprocessors and 8- and 16-bit operations on 32-bit microprocessors. To ensure consistency between simulation and code generation, the results of 8 and 16-bit integer expressions must be explicitly downcast. Selecting this option improves code efficiency by avoiding casts of intermediate variables. See "Expression Folding Options" in *Real-Time Workshop User's Guide* for more information.

Eliminate superfluous temporary variables (Expression folding)

Enables expression folding (see "Using and Configuring Expression Folding" in the "Real-Time Workshop User's Guide").

Reuse block outputs

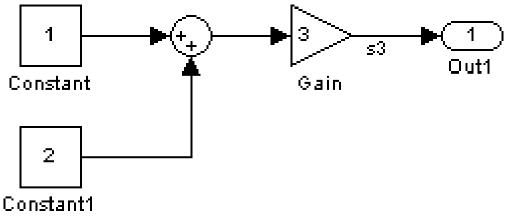
When the Reuse block output check box is selected (the default) Real-Time Workshop reuses signal memory whenever possible. When Reuse block output is cleared, signals are stored in unique locations.

Note Reuse block output is available only when the Signal storage reuse check box is selected.

See “Signal Storage, Optimization, and Interfacing” in *Real-Time Workshop User’s Guide* for further information (including generated code example) on Reuse block output and other signal storage options.

Inline invariant signals

This option applies only if inline parameters is enabled (see “Inline parameters” on page 11-90). If you select this option, Real-Time Workshop uses numeric constants instead of variables to represent invariant signals in generated code. An invariant signal is a signal that does not change during simulation. Consider, for example, the following model:

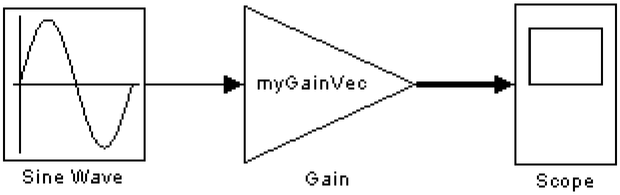


The signal s3 is an invariant signal. This option uses a numeric constant, 9, to represent the value of this signal in the generated code.

Loop unrolling threshold

Specifies the array size at which Real-Time Workshop begins to use a for loop instead of separate assignment statements to assign values to the elements of a signal or parameter array. The default threshold is 5.

For example, consider the following model:



The gain parameter of the Gain block is the vector myGainVec.

Assume that the loop unrolling threshold value is set to the default, 5, and that you have a 10-element vector to myGainVec:

```
myGainVec = [1:10];
```

The generated code declares a 10-element vector variable, myGainVec_P.Gain_Gain[], in the Parameters_model data structure. The size of the gain array exceeds the loop unrolling threshold. Therefore, the code generated for the Gain block uses a for loop, as shown in the following code fragment:

```
{
  int32_T i1;

  /* Gain: '<Root>/Gain' */
  for(i1=0; i1<10; i1++) {
    myGainVec_B.Gain_f[i1] = rtb_foo *
    myGainVec_P.Gain_Gain[i1];
  }
}
```

If myGainVec is declared as

```
myGainVec = [1:3];
```

an array of three elements, myGainVec_P.Gain_Gain[], is declared in the **Parameters_model** data structure. The size of the gain array is below the loop unrolling threshold. The generated code consists of inline references to each element of the array, as in the code fragment below:

```
/* Gain: '<Root>/Gain' */
myGainVec_B.Gain_f[0] = rtb_foo * myGainVec_P.Gain_Gain[0];
myGainVec_B.Gain_f[1] = rtb_foo * myGainVec_P.Gain_Gain[1];
myGainVec_B.Gain_f[2] = rtb_foo * myGainVec_P.Gain_Gain[2];
```

See the *Real-Time Workshop Target Language Compiler* for more information on loop unrolling.

Remove code from floating-point to integer conversions that wraps out-of-range values

This option causes Real-Time Workshop to remove code that ensures that execution of the generated code produces the same results as simulation when out-of-range conversions occur. This reduces the size and increases the speed of the generated code at the cost of potentially producing results that do not match simulation in the case of out-of-range values.

Note Enabling this option affects code generation results only for out-of-range values and hence cannot cause code generation results to differ from simulation results for in-range values.

Consider using this option if code efficiency is critical to your application and the following conditions are true for at least one block in the model:

- Computing the block's outputs or parameters involves converting floating-point data to integer or fixed-point data
- The block's **Saturate on integer overflow** option is disabled:

The following code fragment shows the code generated for a conversion with this option disabled

```
  _fixptlowering0 = (rtb_Switch[i1] + 9.0) / 0.09375;
     _fixptlowering1 = fmod(_fixptlowering0 >= 0.0 ?
floor(_fixptlowering0) :
        ceil(_fixptlowering0), 4.2949672960000000E+009);
      if(_fixptlowering1 < -2.1474836480000000E+009) {
        _fixptlowering1 += 4.2949672960000000E+009;
      } else if(_fixptlowering1 >= 2.1474836480000000E+009) {
        _fixptlowering1 -= 4.2949672960000000E+009;
      }
      cg_in_0_20_0[i1] = (int32_T)_fixptlowering1;
```

Note The code generator uses the `fmod` function to handle out-of-range conversion results.

The code generated for the conversion when you select this optimization follows:

```
cg_in_0_20_0[i1] = (int32_T)((rtb_Switch[i1] + 9.0) / 0.09375);
```

Use bitsets for storing state configuration

Enabling this option specifies that bitsets be used for storing state configuration variables. This can significantly reduce the amount of memory required to store the variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

Use bitsets for storing boolean data

Enabling this option specifies that bitsets be used for storing Boolean data. This can significantly reduce the amount of memory required to store Boolean variables. However, it can increase the amount of memory required to store target code if the target processor does not include instructions for manipulating bitsets.

Minimize array reads using temporary variables

In certain microprocessors, global array read operations are more expensive than accessing a temporary variable on stack. Using this option minimizes array reads by using temporary variables when possible.

For example, the generated code

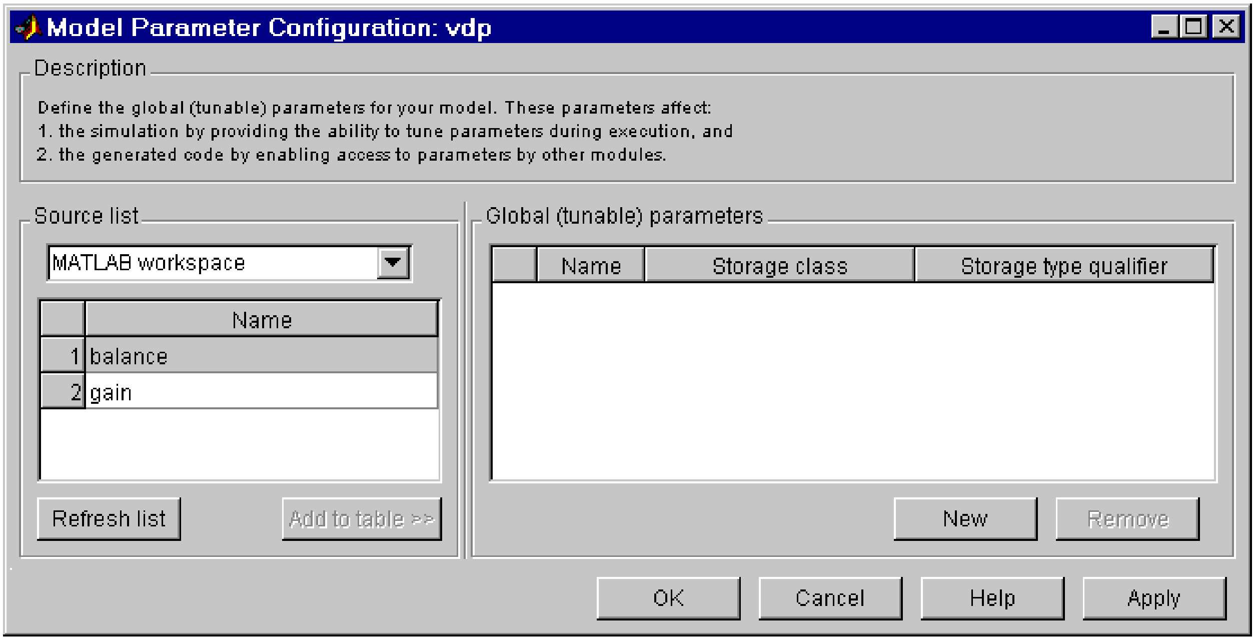
```
a[i] = foo();
if(a[i]<10 && a[i]>1) {
    y = a[i]+5;
}else{
z = a[i];
}
```

now becomes

```
a[i] = foo();
temp = a[i];
if(temp<10 && temp>1) {
    y = temp+5;
}else{
    z = temp;
}
```

Model Parameter Configuration Dialog Box

The **Model Parameter Configuration** dialog box allows you to override the **Inline parameters** option (see “Inline parameters” on page 11-90) 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

Note Simulink ignores the settings of this dialog box if a model contains references to other models. However, you can still tune parameters of such models, using `Simulink.Parameter` objects (see “Model Referencing and the Inline Parameters Optimization” on page 3-69 for more information).

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.

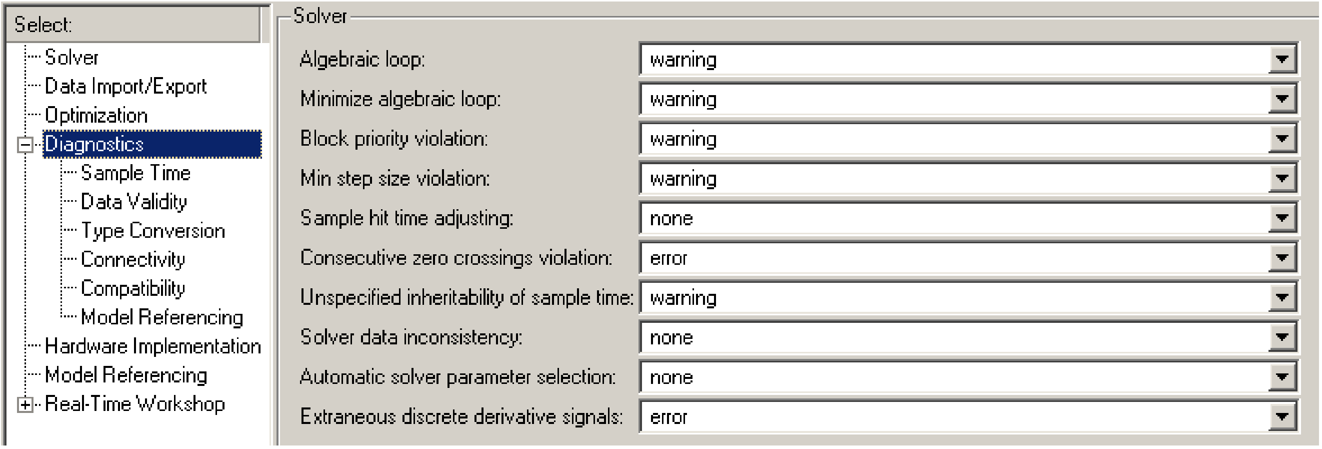
Note This option does not create the corresponding variable in the MATLAB workspace. You must create the variable yourself.

Storage class. Used for code generation. See *Real-Time Workshop User's Guide* for more information.

Storage type qualifier. Used for code generation. See *Real-Time Workshop User's Guide* for more information.

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.



Solver

Algebraic loop: warning

Minimize algebraic loop: warning

Block priority violation: warning

Min step size violation: warning

Sample hit time adjusting: none

Consecutive zero crossings violation: error

Unspecified inheritability of sample time: warning

Solver data inconsistency: none

Automatic solver parameter selection: none

Extraneous discrete derivative signals: error

The options are typically to do nothing or to display a warning or an error message (see “Diagnosing Simulation Errors” on page 11-144). A warning message does not terminate a simulation, but an error message does.

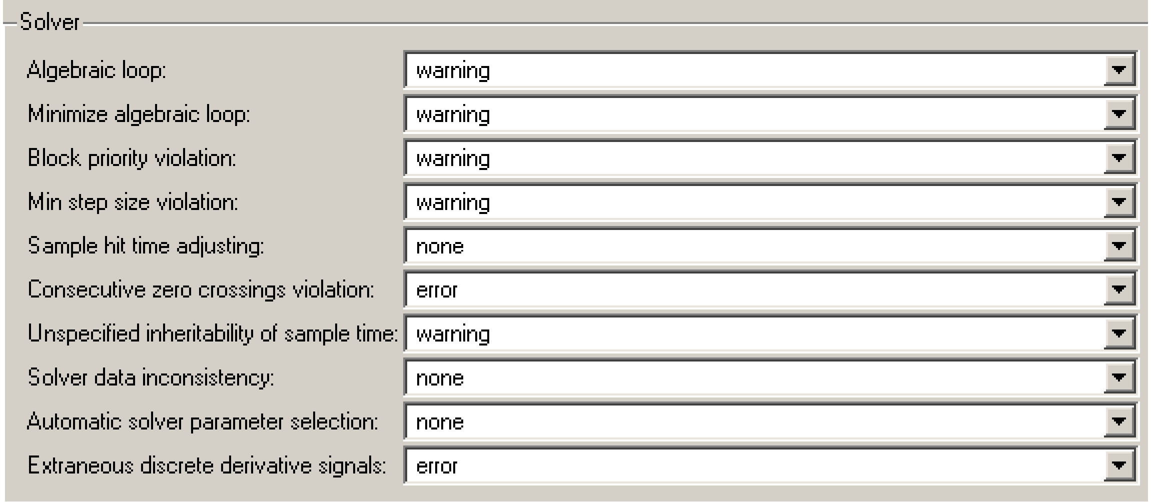
The pane displays groups of controls corresponding to various categories of abnormal conditions that can occur during a solution. The pane initially displays solver diagnostics. To display controls for another category, left-click the category in the **Select** list on the left side of the **Diagnostics** pane. See the following sections for information on using the controls on the **Diagnostics** pane:

- “Solver Diagnostics” on page 11-103
- “Sample Time Diagnostics” on page 11-109
- “Data Validity Diagnostics” on page 11-111
- “Signal Validity Diagnostics” on page 11-112
- “Parameter Validity Diagnostics” on page 11-115
- “Data Store Validity Diagnostics” on page 11-118

- “Debugging Data Validity Diagnostics” on page 11-119
- “Type Conversion Diagnostics” on page 11-120
- “Connectivity Diagnostics” on page 11-121
- “Compatibility Diagnostics” on page 11-125
- “Model Reference Diagnostics” on page 11-131

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
Sample hit time adjusting:	none
Consecutive zero crossings violation:	error
Unspecified inheritability of sample time:	warning
Solver data inconsistency:	none
Automatic solver parameter selection:	none
Extraneous discrete derivative signals:	error

Algebraic loop. Specifies diagnostic action to take if Simulink detects an algebraic loop while compiling the model. See “Algebraic Loops” on page 1-26 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 1-28).

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 1-28) 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. Specifies diagnostic action to take if Simulink detects a block priority specification error while compiling the model.

Min step size violation. Specifies diagnostic action to take if Simulink detects that 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 11-74 and "Maximum order" on page 11-74 for more information. Simulink allows you to specify the maximum number of consecutive minimum step size violations permitted (see "Number of consecutive min step size violations allowed" on page 11-75).

Sample hit time adjusting. Specifies diagnostic action to take if Simulink makes a minor adjustment to a sample hit time while running the model. Simulink might change a sample hit time if that hit time is close to the hit time for another task. If Simulink considers the difference to be due only to numerical errors (for example, precision issues or round-off errors), it changes the sample hits of the faster task or tasks to exactly match the time of the slowest task that has that hit.

Note Over time, these sample hit changes might cause a discrepancy between the numerical simulation results and the actual theoretical results.

When set to `warning`, the MATLAB Command Window displays a warning message like the following when Simulink detects a change in the sample hit time:

```
  Warning: Timing engine warning: Changing the hit time for ...
```

If the field is set to `none`, Simulink does not display a warning message.

Consecutive zero crossings violation. Specifies diagnostic action to take when Simulink detects the maximum number of consecutive zero crossings allowed (see “Number of consecutive zero crossings allowed” on page 11-72). When Simulink detects this violation, it reports the current simulation time, the number of consecutive zero crossings counted, and the type and name of the block in which Simulink detected the zero crossings. For more information, see “Preventing Excessive Zero Crossings” on page 1-23.

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 11-76).

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*.

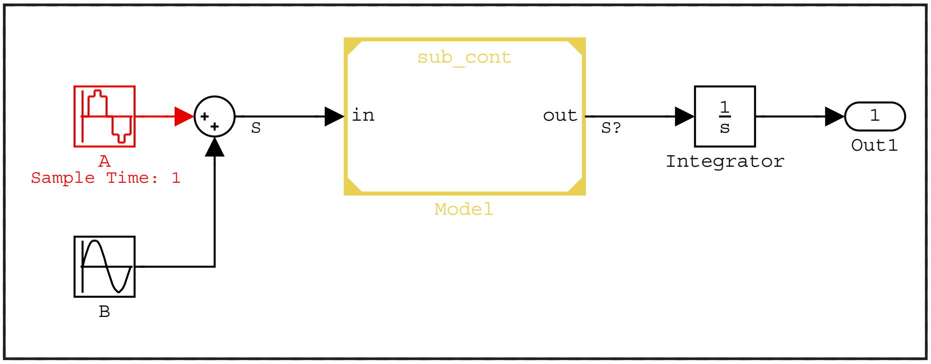
When consistency checking is enabled, Simulink recomputes the appropriate values and compares them to the cached values. If the values are not the same, a consistency error occurs. Simulink compares computed values for these quantities:

- 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 diagnostic. In this case, Simulink changes the solver type to discrete and displays a warning message about this change at the MATLAB command line.

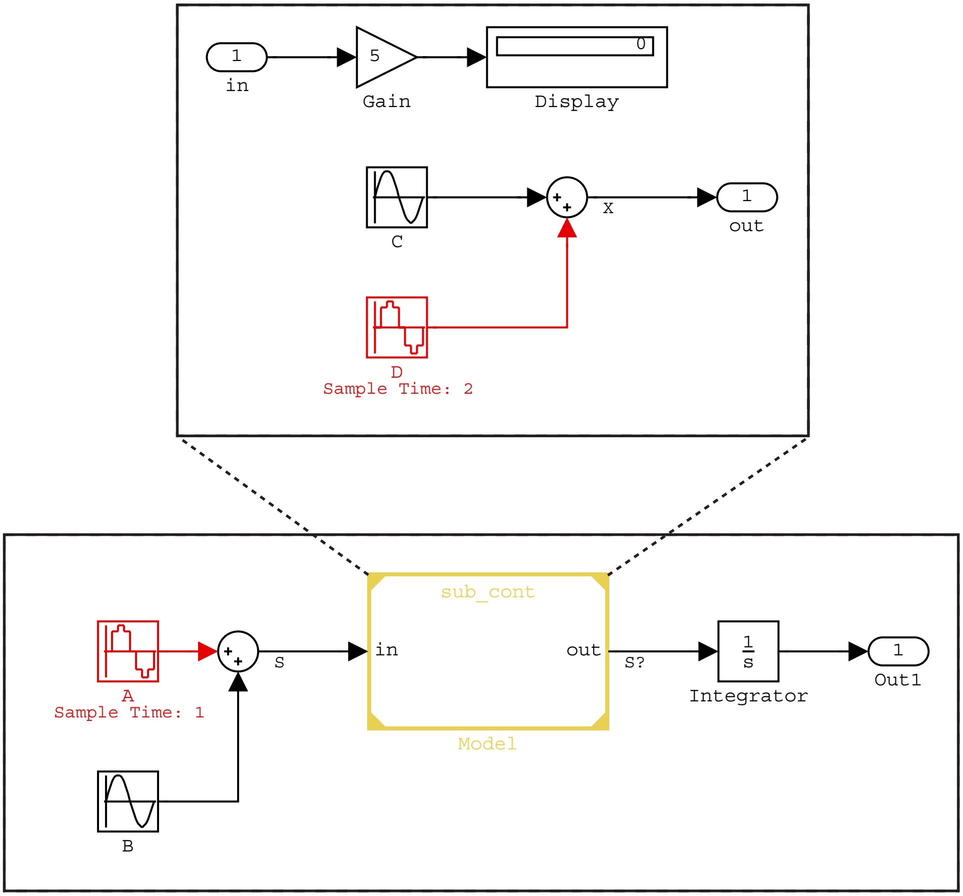
Extraneous discrete derivative signals. A discrete signal appears to pass through a Model block to the input of a block with continuous states, such as an Integrator block. Simulink cannot determine with certainty the minimum rate at which it needs to reset the solver to solve this model accurately. Consequently, if this diagnostic is set to `Error`, Simulink halts when compiling this model and displays an error. If this diagnostic is set to `None` or `Warning`, Simulink resets the solver whenever the value of the discrete signal changes. This ensures accurate simulation of the model, assuming that the discrete signal really is the source of the signal entering the block with continuous states. However, if the discrete signal is not the actual source of the signal entering the block with continuous states, resetting the solver at the rate the discrete signal changes can lead to the solver being reset more frequently than necessary, thus unnecessarily slowing down the simulation.

The following model illustrates the rationale for this diagnostic.



In this model, it is possible, but not certain, that signal S passes through the Model block and enters the Integrator block. The signal emerging from the Model block is labeled S? to indicate that it is not possible, simply from examining the top model alone, to determine that its source actually is S. Because S is the sum of a continuous and a discrete signal, discontinuities occur in S at the sample rate of its discrete component, A, i.e., once a second of simulation time.

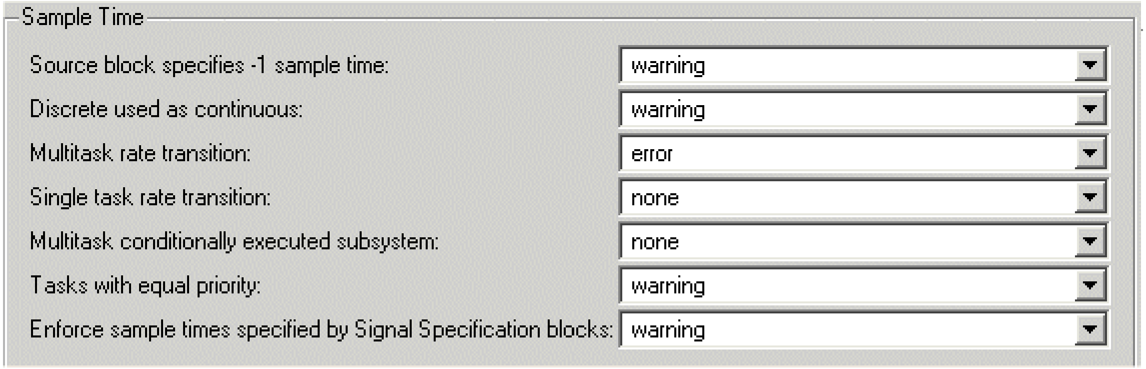
Assuming that the source of S? really is S, Simulink would have to reset the solver once a second to solve the top model accurately. However, examination of the referenced model reveals that the source of S? is actually X.



Note that X, like S, is the sum of a continuous and a discrete signal. However, the discontinuities in X occur every two seconds, half the rate at which discontinuities occur in S. Consequently, to simulate the top model accurately, Simulink needs to reset the solver only every two seconds. However, because the content of the model referenced by the Model block is inaccessible to Simulink when it compiles the top model, preparatory to simulating it, Simulink cannot determine the minimum rate at which it needs to reset the solver to solve this model. Consequently, if this diagnostic is set to `Error`, Simulink displays an error if you try to update or simulate this model. If this diagnostic is set to `None` or `Warning`, Simulink resets the solver whenever the value of S changes, i.e., twice as often as necessary to solve the model accurately.

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

Multitask conditionally executed subsystem: none

Tasks with equal priority: warning

Enforce sample times specified by Signal Specification blocks: 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 11-79).

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 11-79).

Multitask conditionally executed subsystem. Specifies diagnostic action to take (none, error, or warning) if Simulink detects either of the following conditions:

- Your model uses multitasking solver mode (see "Tasking mode for periodic sample times" on page 11-79) and it contains an enabled subsystem that operates at multiple rates.
- Your model contains a conditionally executed subsystem that can reset its states and that itself contains an asynchronous subsystem.

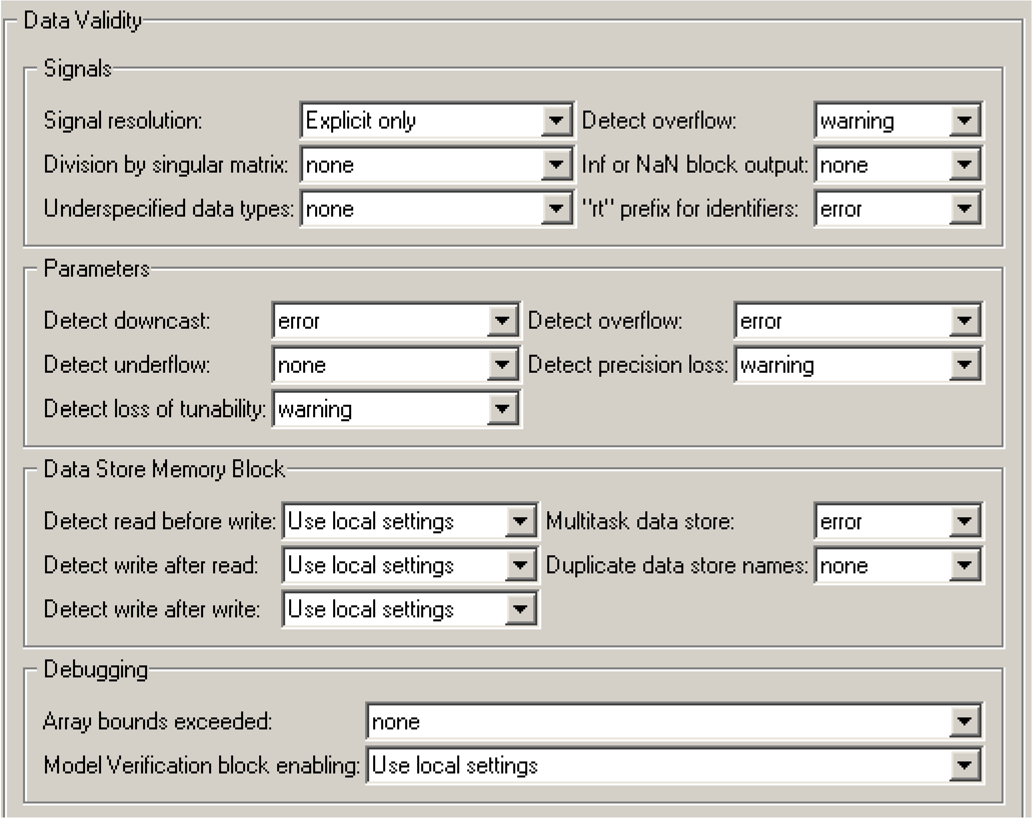
Such subsystems can cause corrupted data or non-deterministic behavior in a real-time system using code generated from the model. In the first case, consider using single-tasking solver mode (see "Tasking mode for periodic sample times" on page 11-79) or using a single-rate enabled subsystem instead. In the second case, consider moving the asynchronous subsystem outside the conditionally executed subsystem.

Tasks 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.

Enforce sample times specified by Signal Specification blocks. The sample time of the source port of a signal specified by a Signal Specification block differs from the signal's destination port.

Data Validity 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 Validity

Signals

Signal resolution: Explicit only

Detect overflow: warning

Division by singular matrix: none

Inf or NaN block output: none

Underspecified data types: none

"rt" prefix for identifiers: error

Parameters

Detect downcast: error

Detect overflow: error

Detect underflow: none

Detect precision loss: warning

Detect loss of tunability: warning

Data Store Memory Block

Detect read before write: Use local settings

Multitask data store: error

Detect write after read: Use local settings

Duplicate data store names: none

Detect write after write: Use local settings

Debugging

Array bounds exceeded: none

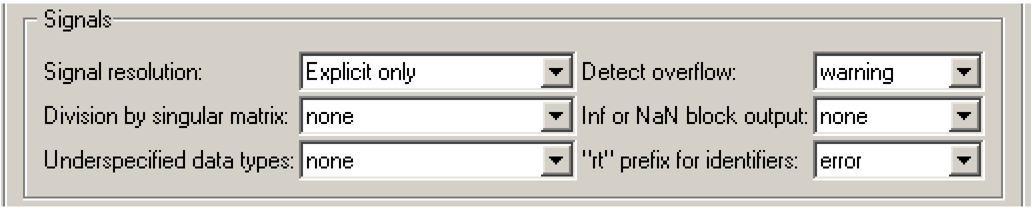
Model Verification block enabling: Use local settings

This group contains the following subgroups.

- "Signal Validity Diagnostics" on page 11-112
- "Parameter Validity Diagnostics" on page 11-115
- "Data Store Validity Diagnostics" on page 11-118
- "Debugging Data Validity Diagnostics" on page 11-119

Signal Validity Diagnostics

The following group of data validity diagnostics pertains to signals.



Signals

Signal resolution: Explicit only

Division by singular matrix: none

Underspecified data types: none

Detect overflow: warning

Inf or NaN block output: none

"rt" prefix for identifiers: error

See “Signal Properties Dialog Box” on page 5-42 for more information about the signal properties mentioned in this section.

Signal resolution. *Signal resolution* is the process of associating a `Simulink.Signal` object that exists in the base workspace with a named signal in a model. If the signal has the same name as the signal object, Simulink can associate the signal with the object. The signal is then said to be *resolved* to the object.

Certain blocks, such as Unit Delay and Data Store Memory, allow you to resolve a named state within the block to a `Simulink.Signal` object. For brevity, the rest of the section mentions only resolving signals, but the same capabilities apply to resolving states.

When a signal is resolved to a signal object, the object specifies the properties of the signal. Multiple signals can have the same name. All of those signals can resolve to the same signal object and have the properties that the object specifies. Changing any parameter value of the signal object then changes that value for all signals that resolve to that object.

The essential requirement for resolving a signal to a signal object is that the object must exist in the base workspace and have the same name as the signal. However, a signal *does not* resolve to a same-named object just because the object exists: you must provide some specification that the resolution should occur. Unless such a specification exists, no resolution happens.

You can control signal resolution in two ways: locally for any individual signal, and globally for all signals that do not contain local specifications.

Local Signal Resolution Control: To control the resolution of an individual signal, right-click the signal to open its Signal Properties dialog box, enter the name of the desired signal object in the **Signal name** field, and select the **Signal name must resolve to a signal object** option. If **Signal name** is a valid signal object, the signal now specifies its resolution locally, and global resolution specifications do not affect it.

Local signal resolution is also called *explicit signal resolution*, because the individual signal explicitly states the resolution that it requires. To prevent a previously specified local resolution from occurring, deselect **Signal name must resolve to a signal object**.

Global Signal Resolution Control: You can configure Simulink to automatically resolve any named signal that is not locally resolved. Such resolution is called *implicit signal resolution*, because nothing explicitly states which resolutions will occur. The resolutions are defined implicitly by the matches between signal names and object names.

By default, implicit signal resolution does not occur: you must actively specify it. You can control global signal resolution by setting the value of **Signal resolution** in the **Data Validity** pane **Signals** subpane to one of the following:

- `Explicit only`

  Do not perform implicit signal resolution. Only explicitly (locally) specified signal resolutions occur.

- `Explicit and warn implicit`

  Perform implicit signal resolution wherever possible, posting a warning of each implicit resolution.

- `Explicit and implicit`

  Perform implicit signal resolution wherever possible without posting any warnings.

Signal Resolution Control Parameters: You can use the API to control signal resolution by setting the configuration parameter `SignalResolutionControl`, which implements **Signal resolution** internally. The corresponding GUI and API values appear in the following table:

Signal resolution (GUI)	`SignalResolutionControl` **(API)**
Explicit only	`UseLocalSettings`
Explicit and implicit	`TryResolveAll`
Explicit and warn implicit	`TryResolveAllWithWarnings`

Eliminating Implicit Resolution: The MathWorks discourages using implicit signal resolution except for fast prototyping, because implicit resolution slows performance, complicates model validation, and can have nondeterministic effects. Simulink provides the `disableimplicitsignalresolution` function, which you can use to change settings throughout a model so that it does not use implicit signal resolution. See the function's reference documentation, or type:

```
help disableimplicitsignalresolution
```

in the MATLAB Command Window.

Division by singular matrix. The Product block detected a singular matrix while inverting one of its inputs in matrix multiplication mode.

Underspecified data types. Simulink could not infer the data type of a signal during data type propagation.

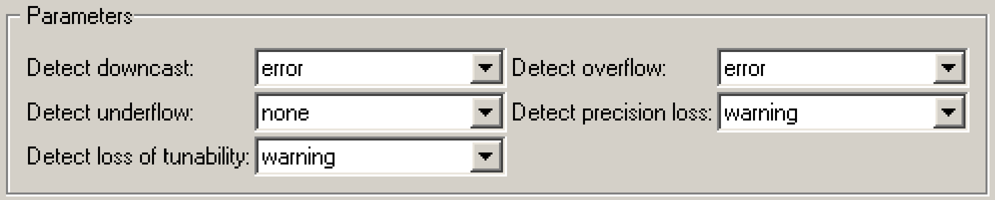
Detect overflow. The value of a signal or parameter is too large to be represented by the signal or parameter's data type. See "Working with Data Types" on page 7-2 for more information.

Inf or NaN block output. The value of a block output is `Inf` or `NaN` at the current time step.

"rt" prefix for identifiers. The default setting causes code generation to terminate with an error if it encounters a Simulink object name, e.g., the name of a parameter or block or signal, that begins with `rt`. This is intended to prevent inadvertent clashes with generated identifiers whose names begins with `rt`.

Parameter Validity Diagnostics

The following group of data validity diagnostics pertains to parameters.



Parameters

Detect downcast: error

Detect overflow: error

Detect underflow: none

Detect precision loss: warning

Detect loss of tunability: warning

Detect 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.

Detect overflow. Simulink has encountered a parameter whose data type's range is not large enough to accommodate the parameter's ideal value, i.e., the ideal value is either too large or too small to be represented by the data type. For example, suppose that the parameter's ideal value is 200 and its data type is `uint8`. Overflow occurs in this case because the maximum value that `uint8` can represent is 127.

Note that parameter overflow differs from parameter precision loss, which occurs when the ideal parameter value is within the range of the data type and scaling being used, but cannot be represented exactly.

Both parameter overflow and precision loss are quantization errors, and the distinction between them can be a fine one. The **Detect overflow** diagnostic reports all quantization errors greater than one bit. For very small parameter quantization errors, precision loss will be reported rather than an overflow when

$$(Max + Slope) \geq V_{ideal} > (Min - Slope)$$

where

- Max is the maximum value representable by the parameter data type
- Min is the minimum value representable by the parameter data type
- $Slope$ is the slope of the parameter data type (slope = 1 for integers)
- V_{ideal} is the ideal value of the parameter

Detect parameter underflow. Simulink has encountered a parameter whose data type does not have enough precision to represent the parameter's ideal value because the ideal value is too small. As a result, casting the ideal value to the data type causes the parameter's modeled value to become zero, i.e., to differ from its ideal value.

Detect precision loss. Simulink has encountered a parameter whose data type does not have enough precision to represent the parameter's value exactly. As a result, the modeled value differs from the ideal value.

Note that parameter precision loss differs from parameter overflow, which occurs when the range of the parameter's data type, i.e., that maximum value that it can represent, is smaller than the ideal value of the parameter. Both parameter overflow and precision loss are quantization errors, and the distinction between them can be a fine one. The **Detect Parameter overflow** diagnostic reports all parameter quantization errors greater than one bit. For very small parameter quantization errors, precision loss will be reported rather than an overflow when

$$(Max + Slope) \geq V_{ideal} > (Min - Slope)$$

where

- *Max* is the maximum value representable by the parameter data type.
- *Min* is the minimum value representable by the parameter data type.
- *Slope* is the slope of the parameter data type (slope = 1 for integers).
- V_{ideal} is the full-precision, ideal value of the parameter.

Detect loss of tunability.

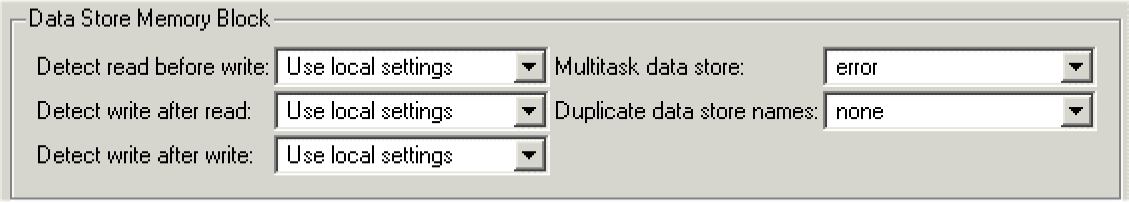
If a tunable workspace variable is modified by Mask Initialization code, or is used in an arithmetic expression with unsupported operators or functions, the expression is reduced to a numeric expression and therefore cannot be tuned. You can use the **Detect loss of tunability** diagnostic to report such loss of tunability. The possible values are:

- `none` — Loss of tunability can occur without notification.
- `warning` — Loss of tunability generates a warning (default).
- `error` — Loss of tunability generates an error.

The equivalent parameter is `ParameterTunabilityLossMsg`. For a list of supported operators and functions, see “Tunable Expression Limitations”.

Data Store Validity Diagnostics

The following group of data validity diagnostics pertains to data stores defined by Data Story Memory blocks and by `Simulink.Signal` objects (see "Working with Data Stores" on page 3-115).



Data Store Memory Block

Detect read before write: Use local settings

Detect write after read: Use local settings

Detect write after write: Use local settings

Multitask data store: error

Duplicate data store names: none

Detect read before write. The model is attempting to read data from a data store in which it has not stored data in this time step. This option has the following settings:

- `Use local settings`

  For each data store defined by a Data Store Memory block, use the setting specified by the block. This option disables the diagnostic for global data stores (i.e., data stores defined by `Simulink.Signal` objects).

- `Disable All`

  Disables this diagnostic for all data stores accessed by the model.

- `Enable All As Warnings`

  Displays diagnostic as a warning at the MATLAB command line.

- `Enable All As Errors`

  Halts the simulation and displays the diagnostic in an error dialog box.

Detect write after read. The model is attempting to store data in a data store after previously reading data from it in the current time step. This diagnostic has the same options as the previous diagnostic.

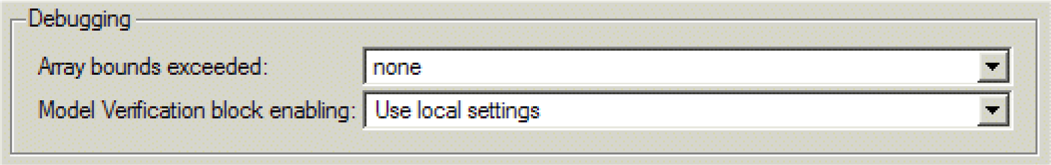
Detect write after write. The model is attempting to store data in a data store twice in succession in the current time step. This diagnostic has the same options as the previous diagnostic.

Multitask data store. One task reads data from a Data Store Memory block to which another task writes data. Such a situation is safe only if one of the tasks cannot interrupt the other, e.g., the data store is a scalar and the writing task uses an atomic copy operation to update the store or the target does not allow the tasks to preempt each other. You should therefore disable this diagnostic, i.e., set it to `none`, only if the application warrants it, e.g. the application uses a cyclic scheduler that prevents tasks from preempting each other.

Duplicate data store names. The model contains multiple Data Store Memory blocks that specify the same data store name.

Debugging Data Validity Diagnostics

The following group of data validity diagnostics pertains to model debugging.



Debugging

Array bounds exceeded: none

Model Verification block enabling: Use local settings

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.

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`

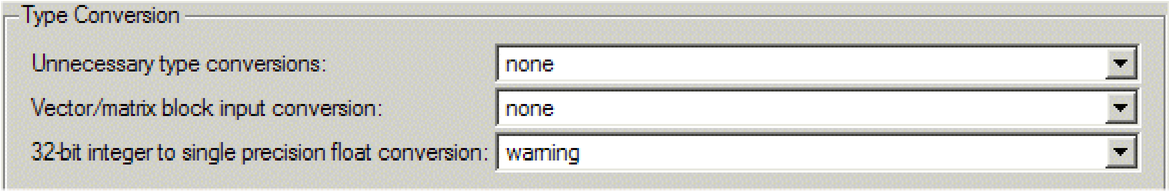
  Enables all model verification blocks in the model regardless of the settings of their **Enable assertion** parameters.

- `Disable all`

  Disables all model verification blocks in the model regardless of the settings of their **Enable assertion** parameters.

Type 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.



Type Conversion

Unnecessary type conversions: none

Vector/matrix block input conversion: none

32-bit integer to single precision float conversion: warning

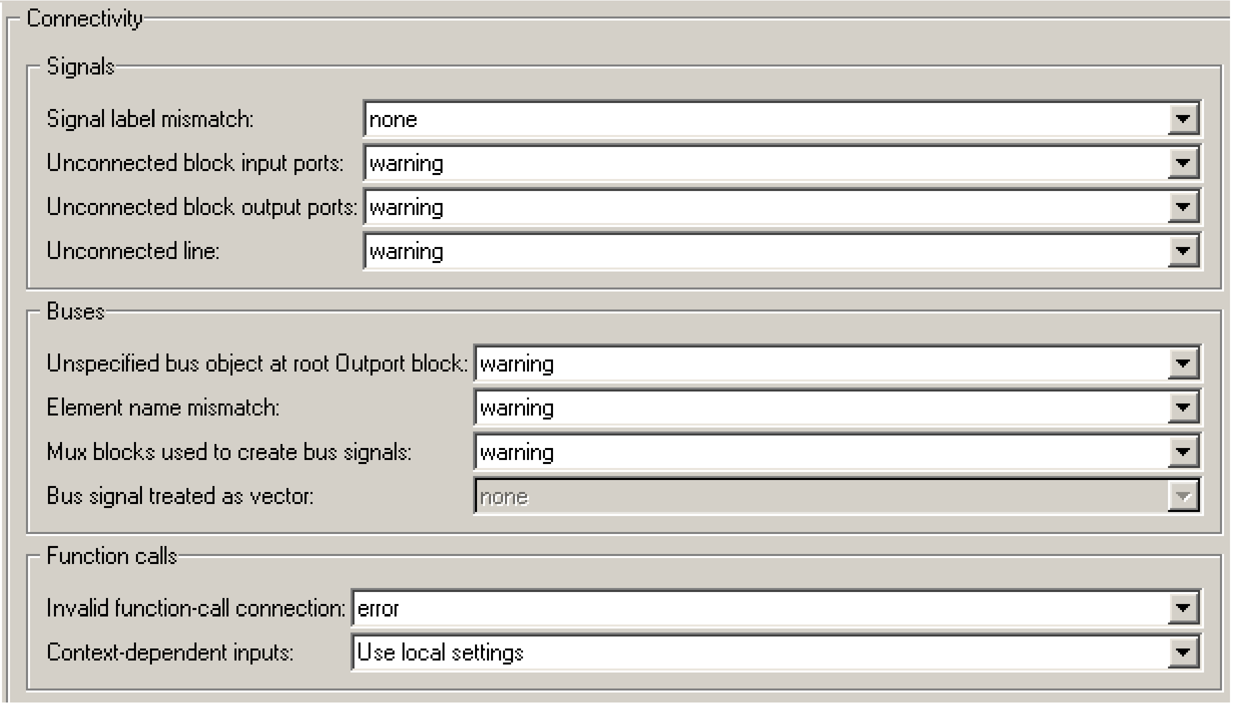
Unnecessary type conversions. A Data Type Conversion block is used where no type conversion is necessary.

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 5-17).

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.

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

Signals

Signal label mismatch: none

Unconnected block input ports: warning

Unconnected block output ports: warning

Unconnected line: warning

Buses

Unspecified bus object at root Outport block: warning

Element name mismatch: warning

Mux blocks used to create bus signals: warning

Bus signal treated as vector: none

Function calls

Invalid function-call connection: error

Context-dependent inputs: Use local settings

Signal label mismatch. The simulation encountered virtual signals that have a common source signal but different labels (see “Virtual Signals” on page 5-10).

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 or an unmatched Goto or From block.

Unspecified bus object at root Outport block. Specifies diagnostic action to take while generating a simulation target for a referenced 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`).

Element name mismatch. Specifies diagnostic action to take if the name of a bus element does not match the name specified by the corresponding bus object. You can use this diagnostic along with bus objects to ensure that your model meets bus element naming requirements imposed by some blocks, such as the Switch block.

Mux blocks used to create bus signals. This diagnostic detects use of Mux blocks to create buses. The diagnostic considers a signal created by a Mux block to be a bus if the signal meets either or both of the following conditions:

- A Bus Selector block individually selects one or more of the signal's elements (as distinct from the entire signal).
- The signal's components have differing data types, numeric types (complex or real), dimensionality, storage classes (see the *Real-Time Workshop User's Guide* for information on storage classes), or sampling modes (see the Signal Processing Blockset documentation for information on frame-based sampling).

The diagnostic has the following options:

- `error`

  This option enforces the following "strict bus" behavior during model editing, updating, and simulation:

  - A Mux block with more than one input is allowed to output only a vector signal. A Mux block with only one input is allowed to output only a scalar, vector, or matrix signal. Simulink displays all nonscalar Mux outputs as wide signals.
  - The dialog boxes for Bus Creator and Bus Creator blocks allow you to select input signals created by Mux blocks but not the individual elements of those signals. For example, suppose that the bus connected to a Bus Selector includes a vector signal created by a Mux block. The Bus Selector allows you to select the vector signal but not any of its elements.

  If this option detects a Mux block that violates strict bus behavior while updating or simulating the model, it halts the model update or simulation and displays a message in the Simulink Diagnostic Viewer. The message identifies the offending Mux block.

- `warning`

  This option does not enforce strict bus behavior. However, if it detects a Mux block that creates a bus during model update or simulation, it displays a message in the MATLAB Command Window that identifies the offending block. It does this for the first ten Mux blocks that it encounters that violate strict bus behavior.

- `none`

  Disables checking for Mux blocks used to create buses. This is the default setting for this diagnostic.

Note You can eliminate warnings and errors about Mux blocks used to create buses by using `slreplace_mux` to remove Mux blocks that violate strict bus behavior from your model. Before executing the command, you should set this diagnostic to `warning` or `none`.

See "Intermixing Composite Signal Types" on page 6-21 for more information.

Bus signal treated as vector. This diagnostic detects the use of virtual bus signals used to specify muxes/vectors. The diagnostic considers a virtual bus signal to be used as a mux/vector if it is input to a Demux block or to any block that can input a mux or a vector but is not formally defined as bus-capable. See “Bus-Capable Blocks” on page 6-16 for details.

Virtual buses can be used as muxes/vectors only when they contain no nested buses and all constituent signals have the same attributes. This practice is deprecated as of R2007a (V6.6) and may cease to be supported at some future time. The MathWorks therefore discourages mixing vectors, muxes, and virtual buses in new applications, and encourages upgrading existing applications to avoid such mixtures.

This diagnostic is available only when **Mux blocks used to create bus signals** is set to `error`. This diagnostic has the following options:

- `error`

  This option causes Simulink to generate an error when it builds a model that uses any virtual bus as a mux/vector.

- `warning`

  This option does not enforce strict bus behavior. If Simulink detects a bus used as a mux/vector, it displays a message in the MATLAB Command Window.

- `none`

  Disables checking for buses used as muxes/vectors. This is the default setting for this diagnostic.

Note You can eliminate warnings and errors about buses used as muxes/vectors by using `Simulink.BlockDiagram.addBusToVector` to insert a Bus to Vector block into any bus signal that is used as a mux/vector. Before executing the command, you should set this diagnostic to `warning` or `none`.

See “Intermixing Composite Signal Types” on page 6-21 for more information.

Invalid function call connection. Simulink has detected an incorrect use of a function-call subsystem in your model. See the Simulink “Subsystem Semantics” demos for examples of invalid uses of function-call subsystems. Disabling this error message can lead to invalid simulation results.

Context-dependent inputs. Controls whether Simulink displays a warning if it has to compute any of a function-call subsystem’s inputs directly or indirectly during execution of a call to a function-call subsystem. See the Simulink "Subsystem Semantics" demos for examples of such function-call subsystems. The options are

- `Use local settings`

  Causes Simulink to issue a warning only if the corresponding diagnostic is selected on the function-call subsystem’s parameters dialog box (see the documentation for the Subsystem block’s parameter dialog box for more information).

- `Enable all`

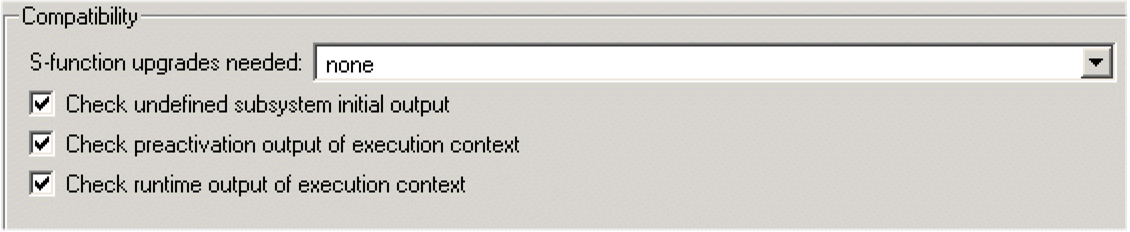
  Enables this diagnostic for all function-call subsystems in this model.

- `Disable all`

  Disables this diagnostic for all function-call subsystems in this model.

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

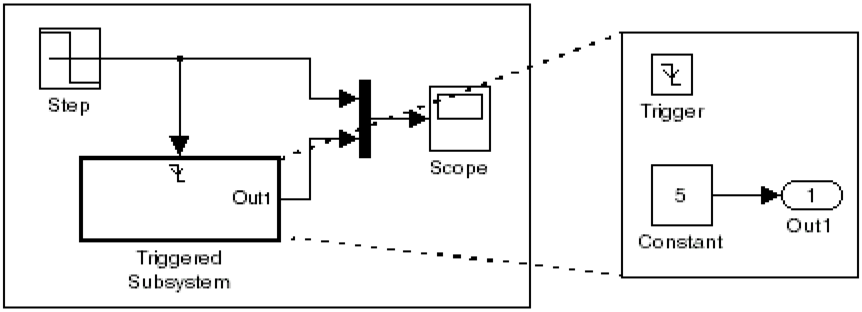
☑ Check runtime output of execution context

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 subsystem 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 [].

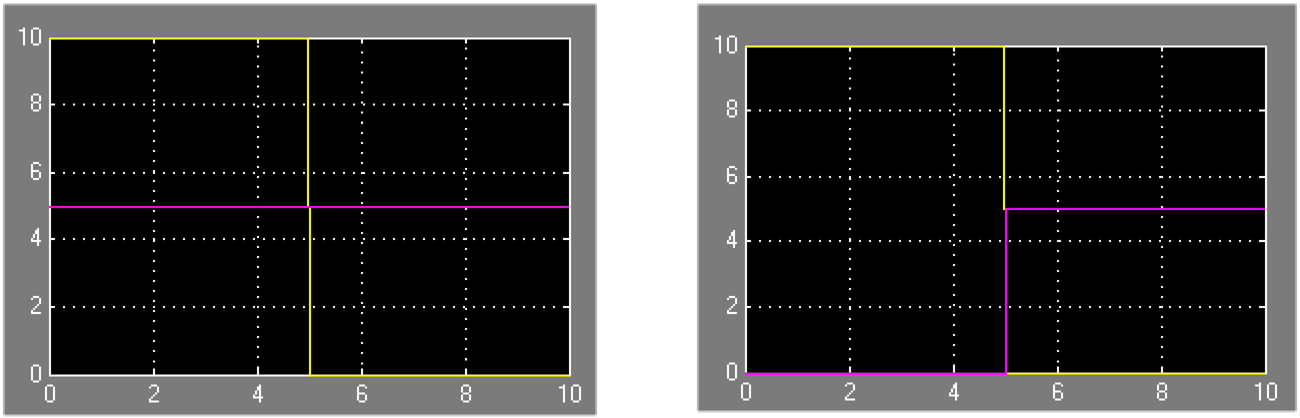
Models with such subsystems can produce initial results (i.e., before initial activation of the conditionally executed subsystem) in the current release that differ from initial results produced in Release 13 or earlier releases.

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

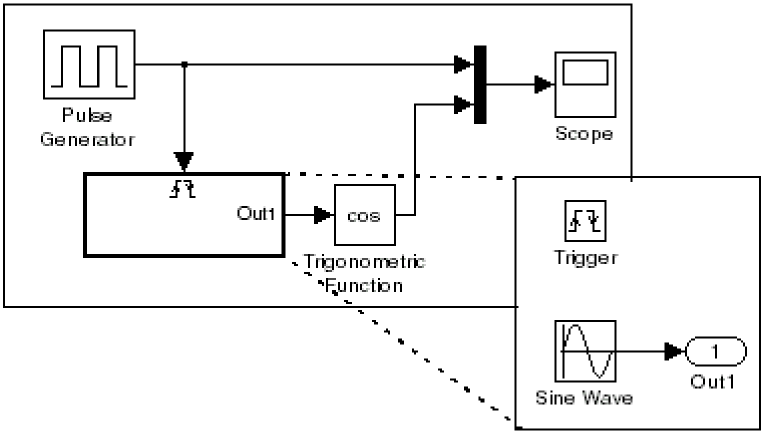
Notice that the initial output of the triggered subsystem differs between the two releases. This is because Release 13 and earlier releases use the initial output of the block connected to the output port (i.e., the Constant block) as the triggered subsystem's initial output. By contrast, this release 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:

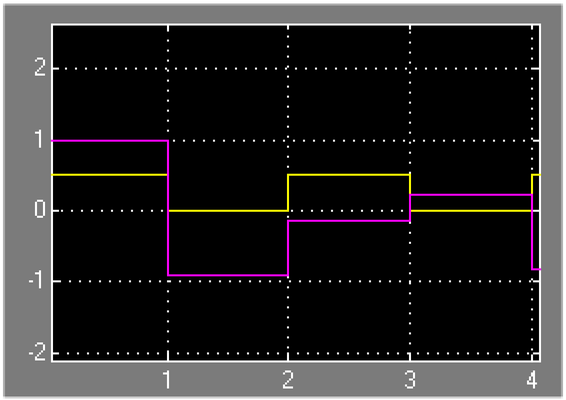
- The block produces nonzero output for zero input (e.g., a Cosine block).
- The block is connected to an output of a conditionally executed subsystem.
- The block inherits its execution context from that subsystem.
- The Outport to which it is connected has an undefined initial condition, i.e., the 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.

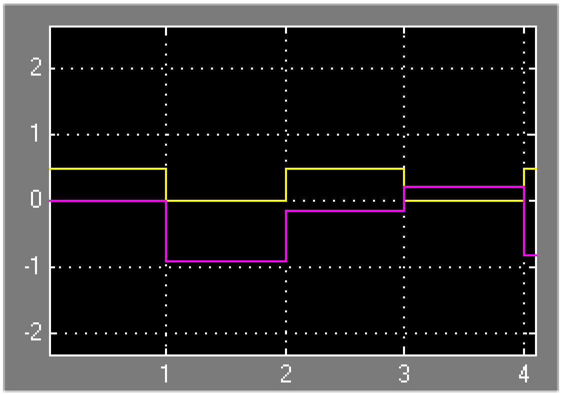
Consider for example the following model.



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

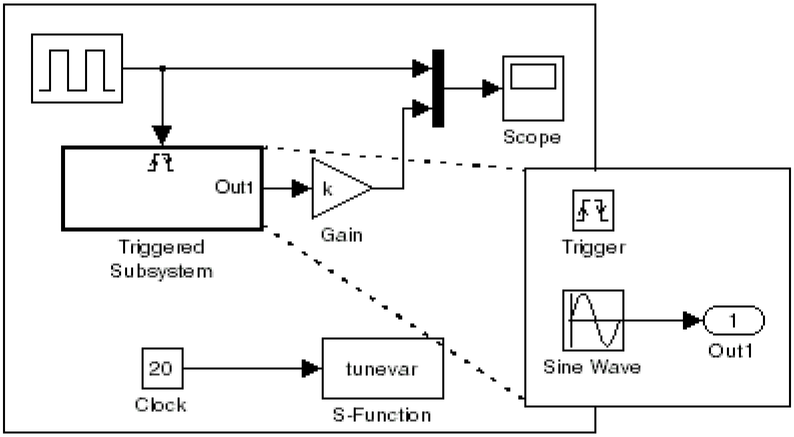
Note that the initial output of the cos block differs between the two releases. This is because in Release 13, the cos block belongs to the execution context of the root system and hence executes at every time step whereas in the current release, the cos block belongs to the execution context of the triggered subsystem and hence executes only when the triggered subsystem executes.

Check run-time 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 [].

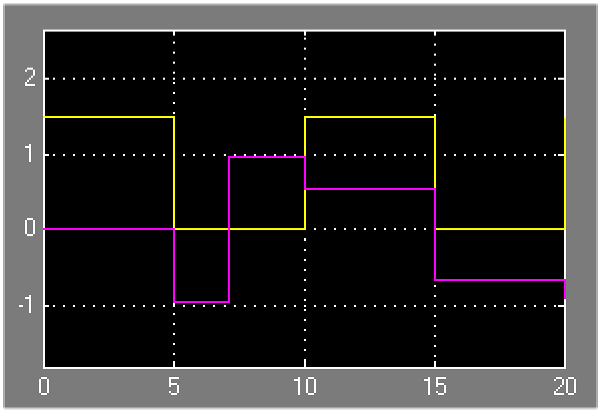
Models with blocks that meet these criteria can produce results when the parameter is tuned in the current release that differ from results produced in Release 13 or earlier releases.

Consider for example the following model.

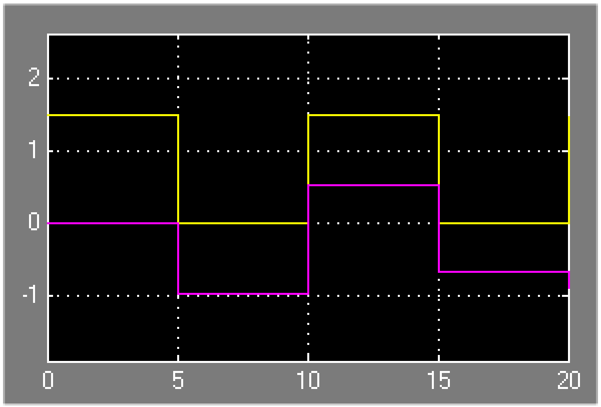


In this model, the `tunevar` S-function changes the value of the Gain block's `k` parameter and updates the diagram at simulation time 7 (i.e., it simulates tuning the parameter).

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

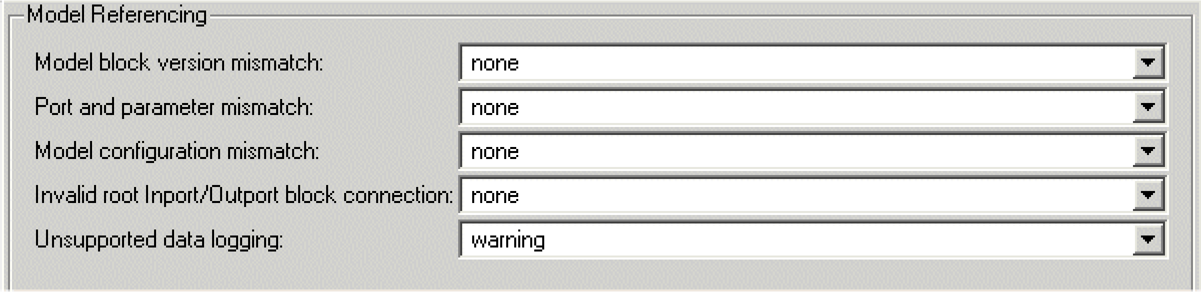


Current Release

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 an incompatibility between this version of Simulink and the model when updating or simulating the model.



Model Referencing

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 3-84), 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.

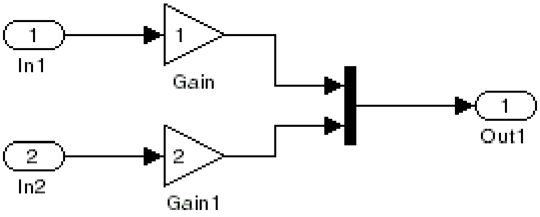
Model block icons can display a message indicating port or parameter mismatches. To enable this feature, select **Block displays > Model Block I/O Mismatch** from the parent model's **Format** menu.

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.

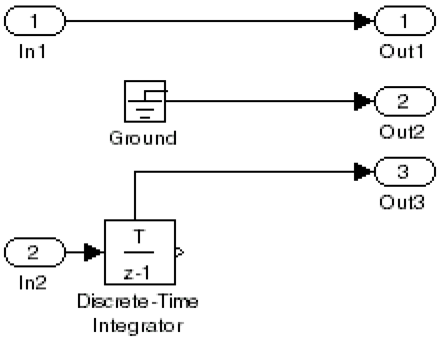
Invalid root Inport/Outport block connection. Specifies the diagnostic action to take during code generation if Simulink detects invalid internal connections to this model's root-level Output port blocks.

When this option is set to error, Simulink reports an error if any of the following types of connections appear in this model.

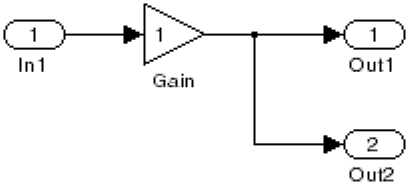
- A root Output port is connected directly or indirectly to more than one nonvirtual block port, for example:



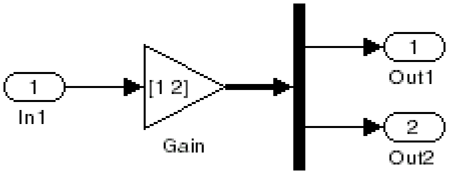
- A root Output port is connected to a root Inport block, a Ground block, or a nondata port (e.g, a state port).



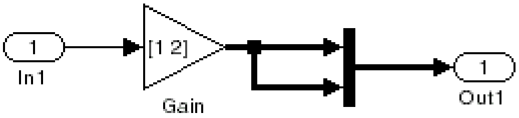
- Two root Outport blocks cannot be connected to the same block port.



- 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.

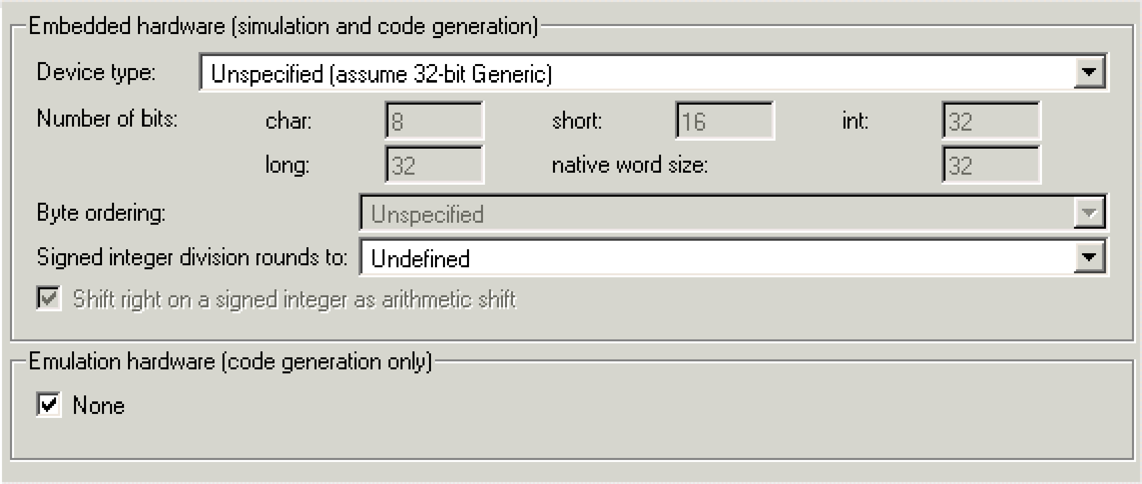
If you select `warning`, Simulink warns you that a connection constraint has been violated and attempts to satisfy the constraint by inserting hidden blocks.

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 5-36 for information on how to log signals from a reference to this model. See *Simulink Reference* for more information.

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 11-137 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 integer 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 “Hardware Implementation Options” in 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.

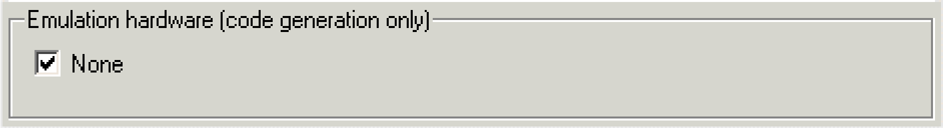
Shift right on a signed integer as arithmetic shift. Select this option if the C compiler implements a signed integer right shift as an arithmetic right shift. An arithmetic right shift fills bits vacated by the right shift with the value of the most significant bit, which indicates the sign of the number 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 "Hardware Implementation Options" in the Real-Time Workshop documentation for more information.

Note For guidelines to observe when configuring **Embedded hardware** controls for code generation, see "Hardware Implementation Options" in the Real-Time Workshop documentation.

Emulation hardware

This group of controls allows you to specify the characteristics of hardware used to test code generated from this model.

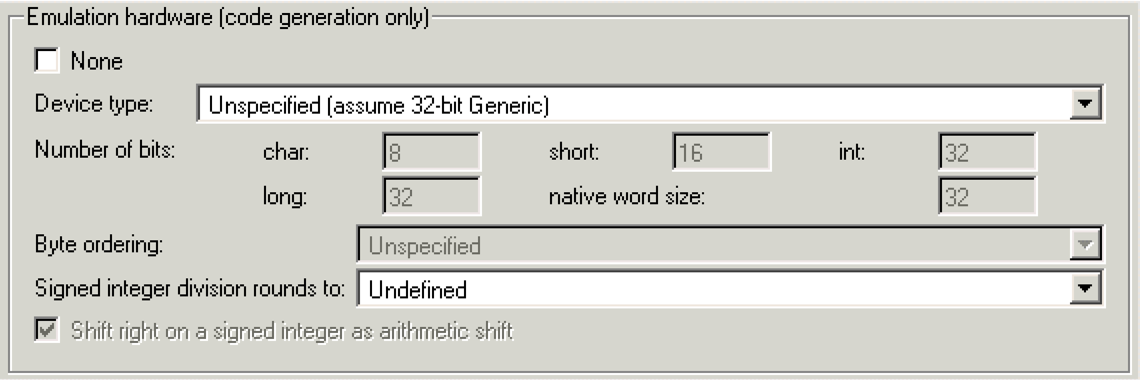


Emulation hardware (code generation only)

☑ 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, deselect 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

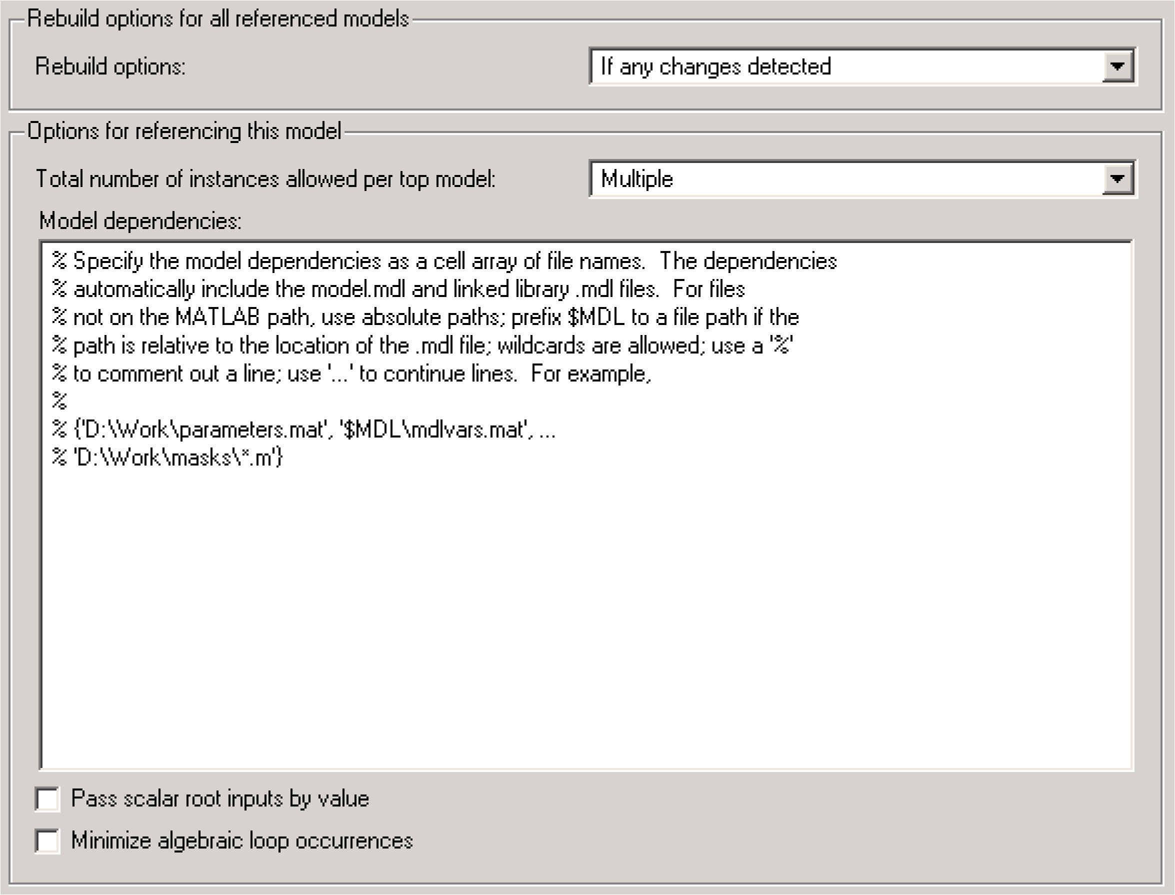
☑ Shift right on a signed integer as arithmetic shift

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 11-135 for information on using these controls.

Note For guidelines to observe when configuring **Emulation hardware** controls, see “Hardware Implementation Options” in the Real-Time Workshop documentation.

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.



Rebuild options for all referenced models

Rebuild options: If any changes detected

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 11-140)
- Including the current model in other models (see "Options for referencing this model" on page 11-142)

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 options. 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`

  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 11-142)
  - 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` or the `If any changes detected` option.

- `Never`

  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` 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)
- `Zero`

If you specify `Zero`, 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 multiply 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 a warning.

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 *Real-Time Workshop User's Guide* 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 1-28 for more information.

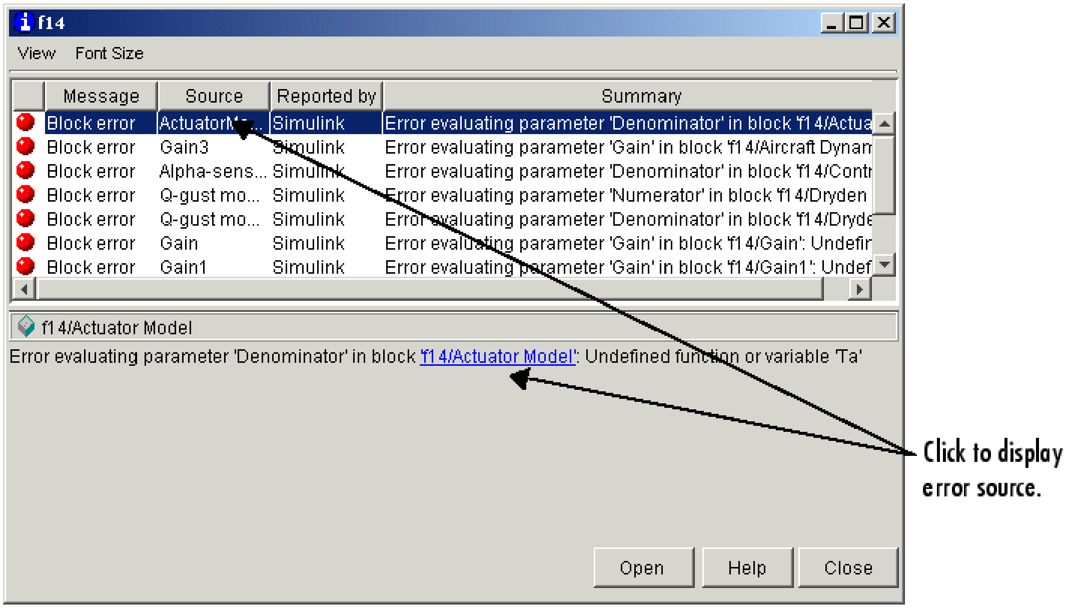
Diagnosing Simulation Errors

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 sections explain how to use the viewer to determine the cause of the errors, and how to create custom error messages:

- “Simulation Diagnostics Viewer” on page 11-144
- “Creating Custom Simulation Error Messages” on page 11-146

Simulation Diagnostics Viewer

The viewer comprises an Error Summary pane and an Error Message pane.



	Message	Source	Reported by	Summary
●	Block error	ActuatorM...	Simulink	Error evaluating parameter 'Denominator' in block 'f14/Actua
●	Block error	Gain3	Simulink	Error evaluating parameter 'Gain' in block 'f14/Aircraft Dynam
●	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.)

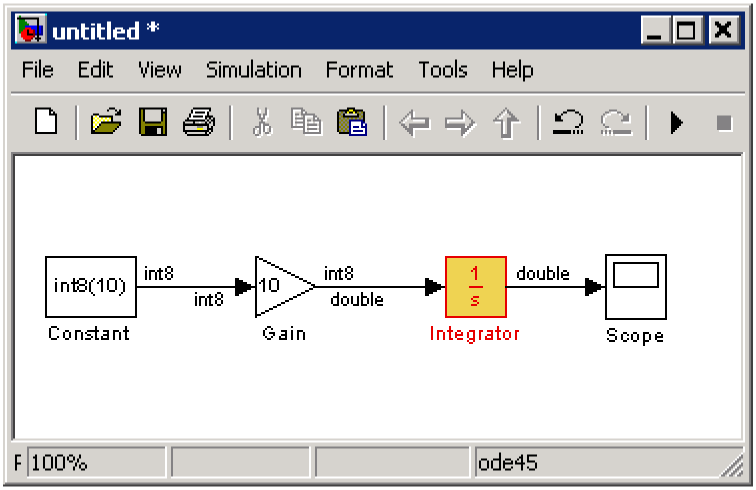
Summary. Error message, abbreviated to fit in the column

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. This includes instances invoked by block or model callbacks, or S-functions that you create or that are executed by the MATLAB Fcn block.

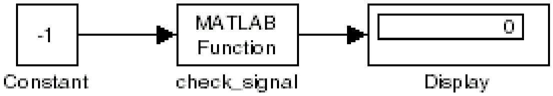
You can use the MATLAB error function in callbacks and S-functions or in the MATLAB Fcn block to create custom error messages specific to your application in several ways:

- Display the contents of a text string
- Include hyperlinks to an object
- Link to an HTML file

Displaying A Text String

To display the contents of a text string, pass to the error function the string enclosed by quotation marks.

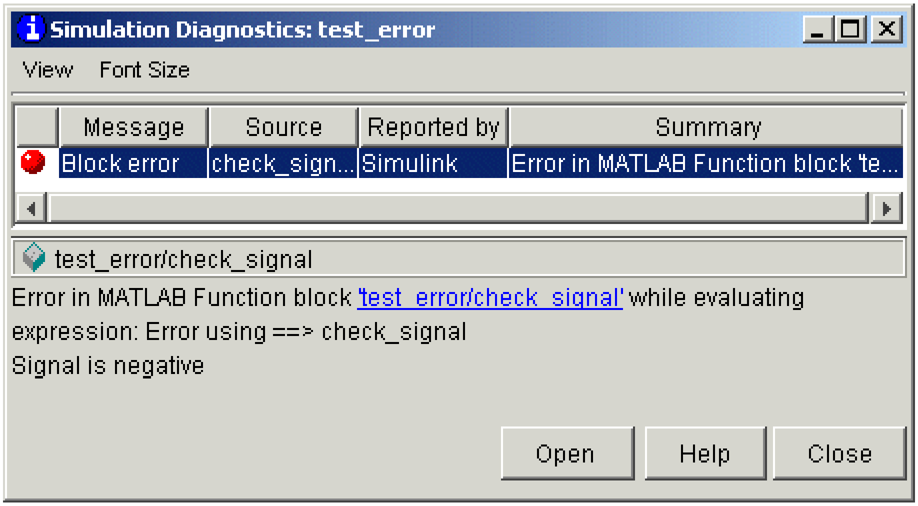
The following example shows how the user-created function `check_signal` can be made to display the string `Signal is negative`.



The MATLAB Fcn block invokes the following function:

```
function y=check_signal(x)
  if x<0
    error('Signal is negative');
  else
    y=x;
  end
```

Executing this model displays the 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

To include a hyperlink to a block, file, or directory, include the item's path in the error message enclosed in quotation marks

- `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`.

Displaying Hyperlinks to Specific Files

This example shows how to display a hyperlink to a specific HTML file.

```
error('Signal is negative.  See %s', '<a href="([docroot
''/toolbox/simulink/ug/f11-33333.html''])">help</a>')
```

In this example the Simulation Diagnostics Viewer displays a text hyperlink labeled `help`. Clicking the link opens the HTML file.

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.

The following topics provide information for speeding up and improving the simulation accuracy:

- "Speeding Up the Simulation" on page 11-149
- "Improving Simulation Accuracy" on page 11-150

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 11-74.

- 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 1-26.
- 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.

Certain modeling constructs can also produce unexpected or inaccurate simulation results.

- A Source block that inherits its sample time can produce different simulation results if, for example, the sample times of the downstream blocks are modified (see "Propagating Sample Times Back to Source Blocks").
- A Derivative block found in an algebraic loop can result in a loss in solver accuracy.

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 use either the sim command or the set_param command to run a simulation programmatically.

- "Using the sim Command" on page 11-152
- "Using the set_param Command" on page 11-152

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 11-37.

Using the set_param Command

You can use the set_param command to start, stop, pause, continue a simulation, update a block diagram, or write all logging variables to the base workspace. 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', 'update', or 'WriteDataLogs'.

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).

Note You cannot use set_param to run a simulation in a MATLAB session that does not have a display, i.e., if you used matlab -nodisplay to start the session.

Running a Simulation from an S-Function

S-functions can use the set_param command to control simulation execution. A C MEX S-function can use the mexCallMatlab macro to call the set_param command itself.

12

Analyzing Simulation Results

The following sections explain how to use Simulink tools for analyzing the results of simulations.

Viewing Output Trajectories

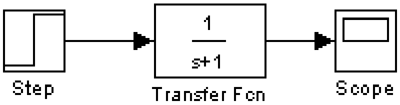
Output trajectories from Simulink can be plotted using one of three methods:

- Feed a signal into either a Scope or an XY Graph block.
- Write output to return variables and use MATLAB plotting commands.
- Write output to the workspace using To Workspace blocks and plot the results using MATLAB plotting commands.

- "Using the Scope Block" on page 12-2
- "Using Return Variables" on page 12-2
- "Using the To Workspace Block" on page 12-3

Using the Scope Block

You can display output trajectories on a Scope block during simulation as illustrated by the following model.

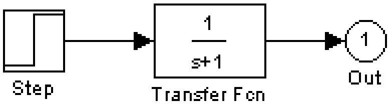


The display on the Scope shows the output trajectory. The Scope block enables you to zoom in on an area of interest or save the data to the workspace.

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

Using Return Variables

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



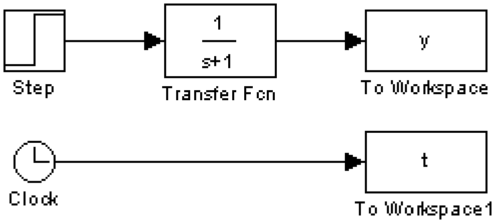
The block labeled Out is an Outport block from the Ports & Subsystems library. The output trajectory, `yout`, is returned by the integration solver. For more information, see “Data Import/Export Pane” on page 11-81.

You can also run this simulation from the **Simulation** menu by specifying variables for the time, output, and states on the **Data Import/Export** pane of the **Configuration Parameters** dialog box. You can then plot these results using

```
plot(tout,yout)
```

Using the To Workspace Block

The To Workspace block can be used to return output trajectories to the MATLAB workspace. The following model illustrates this use:



The variables `y` and `t` appear in the workspace when the simulation is complete. You store the time vector by feeding a Clock block into a To Workspace block. You can also acquire the time vector by entering a variable name for the time on the **Data Import/Export** pane of the **Configuration Parameters** dialog box, for menu-driven simulations, or by returning it using the `sim` command (see “Data Import/Export Pane” on page 11-81 for more information).

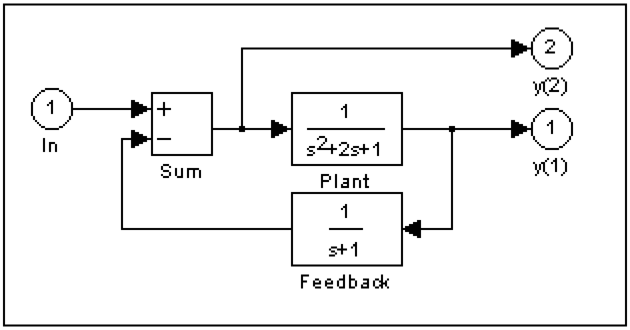
The To Workspace block can accept an array input, with each input element’s trajectory stored in the resulting workspace variable.

Linearizing Models

Simulink provides the `linmod`, `linmod2`, and `dlinmod` functions to extract linear models in the form of the state-space matrices *A*, *B*, *C*, and *D*. State-space matrices describe the linear input-output relationship as

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

where *x*, *u*, and *y* are state, input, and output vectors, respectively. For example, the following model is called `lmod`.



To extract the linear model of this Simulink system, enter this command.

```
[A,B,C,D] = linmod('lmod')

A =
    -2    -1    -1
     1     0     0
     0     1    -1
B =
     1
     0
     0
C =
     0     1     0
     0     0    -1
D =
     0
     1
```

Inputs and outputs must be defined using Inport and Outport blocks from the Ports & Subsystems library. Source and sink blocks do not act as inputs and outputs. Inport blocks can be used in conjunction with source blocks, using a Sum block. Once the data is in the state-space form or converted to an LTI object, you can apply functions in Control System Toolbox for further analysis:

- Conversion to an LTI object

  ```
  sys = ss(A,B,C,D);
  ```

- Bode phase and magnitude frequency plot

  ```
  bode(A,B,C,D) or bode(sys)
  ```

- Linearized time response

  ```
  step(A,B,C,D) or step(sys)
  impulse(A,B,C,D) or impulse(sys)
  lsim(A,B,C,D,u,t) or lsim(sys,u,t)
  ```

You can use other functions in Control System Toolbox and Robust Control Toolbox for linear control system design.

When the model is nonlinear, an operating point can be chosen at which to extract the linearized model. Extra arguments to linmod specify the operating point.

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

For discrete systems or mixed continuous and discrete systems, use the function dlinmod for linearization. This has the same calling syntax as linmod except that the second right-hand argument must contain a sample time at which to perform the linearization.

For models containing references to other models using the Model block, the linmod command must be called with the 'v5' argument to invoke the perturbation algorithm created prior to MATLAB 5.3. This algorithm also allows you to specify the perturbation values used to perform the perturbation of all the states and inputs of the model. See the linmod reference page for more information.

```
[A,B,C,D]=linmod('sys',x,u,para,xpert,upert,'v5')
```

Linearization Using the 'v5' Algorithm

Using linmod with the 'v5' option to linearize a model that contains Derivative or Transport Delay blocks can be troublesome. Before linearizing, replace these blocks with specially designed blocks that avoid the problems. These blocks are in the Simulink Extras library in the Linearization sublibrary.

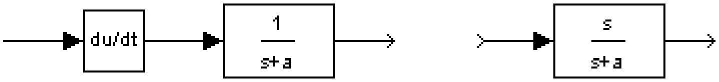
You access the Extras library by opening the Blocksets & Toolboxes icon:

- For the Derivative block, use the Switched derivative for linearization.
- For the Transport Delay block, use the Switched transport delay for linearization. (Using this block requires that you have Control System Toolbox.)

When using a Derivative block, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, it is better implemented (although this is not always possible) with a single Transfer Fcn block of the form

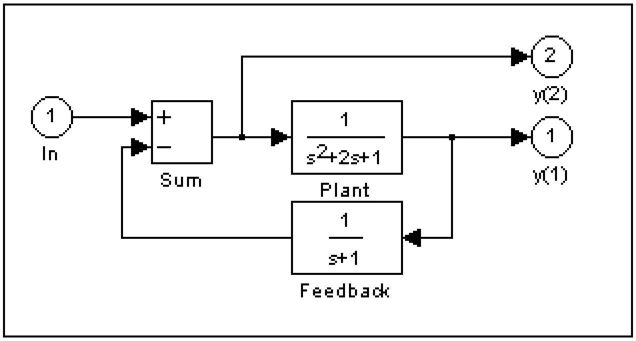
$$\frac{s}{s+a}$$

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



Finding Steady-State Points

The Simulink `trim` function uses a Simulink model to determine steady-state points of a dynamic system that satisfy input, output, and state conditions that you specify. Consider, for example, this model, called `lmod`.



You can use the `trim` function to find the values of the input and the states that set both outputs to 1. First, make initial guesses for the state variables (`x`) and input values (`u`), then set the desired value for the output (`y`).

```
x = [0; 0; 0];
u = 0;
y = [1; 1];
```

Use index variables to indicate which variables are fixed and which can vary.

```
ix = [];      % Don't fix any of the states
iu = [];      % Don't fix the input
iy = [1;2];   % Fix both output 1 and output 2
```

Invoking trim returns the solution. Your results might differ because of roundoff error.

```
[x,u,y,dx] = trim('lmod',x,u,y,ix,iu,iy)

x =
   0.0000
   1.0000
   1.0000
u =
   2
y =
   1.0000
   1.0000
dx =
   1.0e-015 *
    -0.2220
    -0.0227
     0.3331
```

Note that there might be no solution to equilibrium point problems. If that is the case, trim returns a solution that minimizes the maximum deviation from the desired result after first trying to set the derivatives to zero. For a description of the trim syntax, see trim in the *Simulink Reference*.

Creating Block Masks

This section explains how to create custom user interfaces (masks) for Simulink subsystems.

About Masks

A mask is a custom user interface for a subsystem that hides the subsystem's contents, making it appear to the user as an atomic block with its own icon and parameter dialog box. Note that this is different from an Atomic Subsystem, which Simulink treats as a unit when determining the execution order of block methods. Masking a subsystem provides only graphical, not functional, grouping. The Simulink Mask Editor enables you to create a mask for any subsystem. Masking a subsystem allows you to

- Replace the parameter dialogs of a subsystem and its contents with a single parameter dialog with its own block description, parameter prompts, and help text
- Replace a subsystem's standard icon with a custom icon that depicts its purpose
- Prevent unintended modification of subsystems by hiding their contents behind a mask
- Create a custom block by encapsulating a block diagram that defines the block's behavior in a masked subsystem and then placing the masked subsystem in a library

Note You can also mask S-Function and Model blocks. The instructions for masking Subsystem blocks apply to S-Function and Model blocks as well except where noted.

For more information on masks, see:

- "Mask Features" on page 13-2
- "Creating Masks" on page 13-5

Mask Features

Masks can include any of the following features.

Mask Icon

The mask icon replaces a subsystem's standard icon, i.e., it appears in a block diagram in place of the standard icon for a subsystem block. Simulink uses MATLAB code that you supply to draw the custom icon. You can use any MATLAB drawing command in the icon code. This gives you great flexibility in designing an icon for a masked subsystem.

Mask Parameters

Simulink allows you to define a set of user-settable parameters for a masked subsystem. Simulink stores the value of a parameter in the mask workspace (see "Mask Workspace" on page 13-4) as the value of a variable whose name you specify. These associated variables allow you to link mask parameters to specific parameters of blocks inside a masked subsystem (internal parameters) such that setting a mask parameter sets the associated block parameter (see "Linking Mask Parameters to Block Parameters" on page 13-38).

Note If you intend to allow the user to specify the model referenced by a masked Model block or a Model block in a masked subsystem, you must ensure that the mask requires that the user specify the model name as a literal value rather than as a workspace variable. This is because Simulink updates model reference targets before evaluating block parameters. The recommended way to force the user to specify the model name as a literal is to use a pop-up control on the mask to specify the model name. See "Pop-Up Control" on page 13-30 for more information.

Mask Parameter Dialog Box

The mask parameter dialog box contains controls that enable a user to set the values of the mask's parameters and hence the values of any internal parameters linked to the mask parameters.

The mask parameter dialog box replaces the subsystem's standard parameter dialog box, i.e., clicking on the masked subsystem's icon causes the mask dialog box to appear instead of the standard parameter dialog box for a Subsystem block

Note Use the `'mask'` option of the `open_system` command to open a block's mask dialog box at the MATLAB command line or in an M program.

You can customize every feature of the mask dialog box, including which parameters appear on the dialog box, the order in which they appear, parameter prompts, the controls used to edit the parameters, and the parameter callbacks (code used to process parameter values entered by the user).

Mask Initialization Code

The initialization code is MATLAB code that you specify and that Simulink runs to initialize the masked subsystem at critical times, such as model loading and the start of a simulation run (see "Initialization Pane" on page 13-32). You can use the initialization code to set the initial values of the masked subsystem's mask parameters.

Mask Workspace

Simulink associates a workspace with each masked subsystem that you create. Simulink stores the current values of the subsystem's parameters in the workspace as well as any variables created by the block's initialization code and parameter callbacks. You can use model and mask workspace variables to initialize a masked subsystem and to set the values of blocks inside the masked subsystem, subject to the following rules.

- The **Permit Hierarchical Resolution** option of the subsystem is set to `All` or `ExplicitOnly` (see Permit Hierarchical Resolution).
- A block parameter expression can refer only to variables defined in the mask workspaces of the subsystem or nested subsystems that contain the block or in the model's workspace.
- A valid reference to a variable defined on more than one level in the model hierarchy resolves to the most local definition.

  For example, suppose that model M contains masked subsystem A, which contains masked subsystem B. Further suppose that B refers to a variable `x` that exists in both A's and M's workspaces. In this case, the reference resolves to the value in A's workspace.

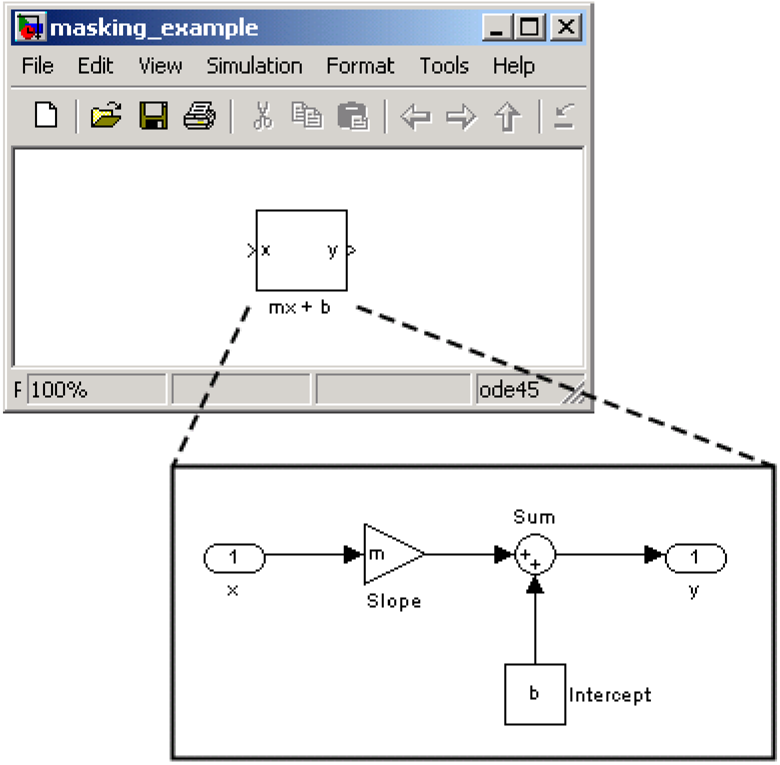
- A masked subsystem's initialization code can refer only to variables in its local workspace.
- The mask workspace of a Model block is not visible to the model that it references. Any variables used by the referenced model must resolve to workspaces defined in the referenced model or to the base (i.e., the MATLAB) workspace.

Creating Masks

See "Masking a Subsystem" on page 13-14 for an overview of the process of creating a masked subsystem. See "Masked Subsystem Example" on page 13-6 for an example of the process.

Masked Subsystem Example

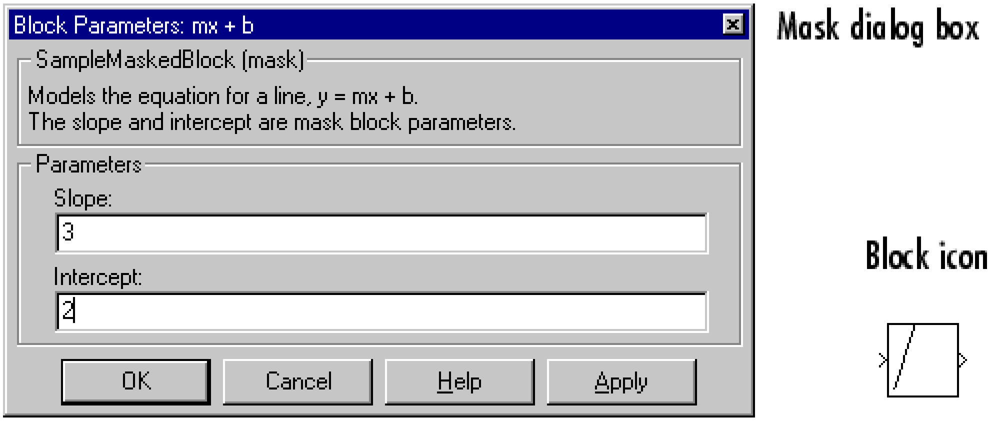
This simple subsystem (masking_example) models the equation for a line, `y = mx + b`.



Ordinarily, when you double-click a Subsystem block, the Subsystem block opens, displaying its blocks in a separate window. The `mx + b` subsystem contains a Gain block, named Slope, whose **Gain** parameter is specified as `m`, and a Constant block, named Intercept, whose **Constant value** parameter is specified as `b`. These parameters represent the slope and intercept of a line.

This example creates a custom dialog box and icon for the subsystem. One dialog box contains prompts for both the slope and the intercept.

After you create the mask, double-click the Subsystem block to open the mask dialog box. The mask dialog box and icon look like this:



Block Parameters: mx + b

SampleMaskedBlock (mask)

Models the equation for a line, y = mx + b.
The slope and intercept are mask block parameters.

Parameters

Slope: 3

Intercept: 2

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

To create a mask for this subsystem, you need to

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

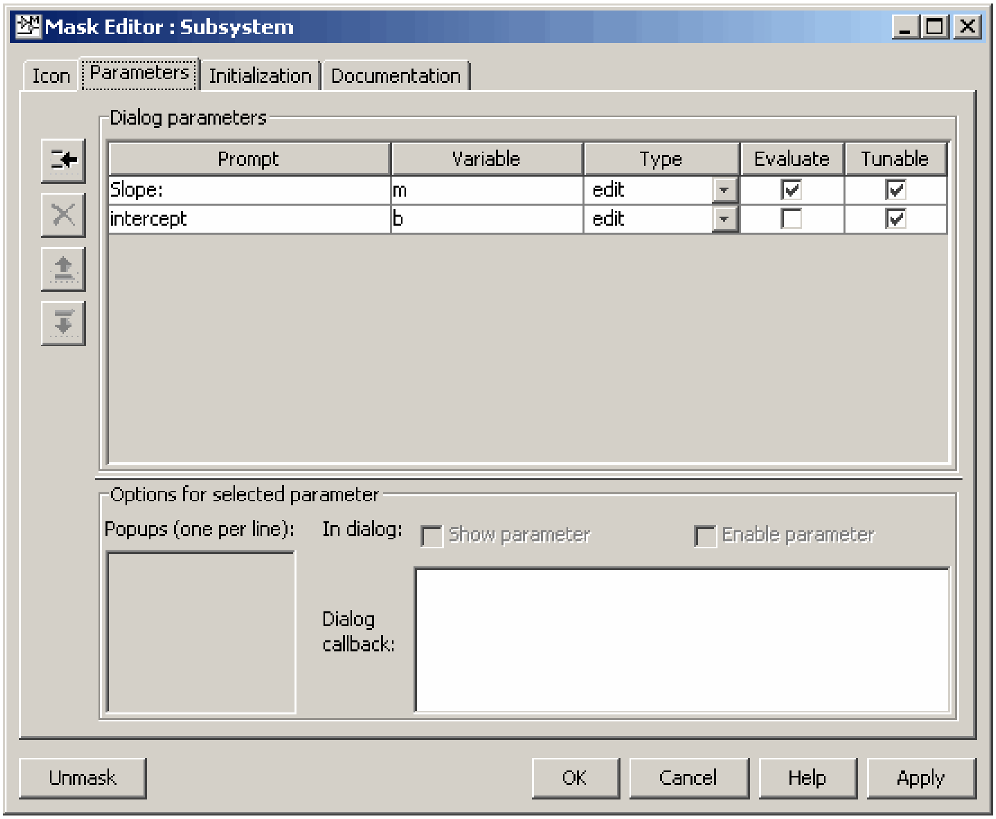
For more information, see:

- "Creating Mask Dialog Box Prompts" on page 13-8
- "Creating the Block Description and Help Text" on page 13-10
- "Creating the Block Icon" on page 13-11

Creating Mask Dialog Box Prompts

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

This example primarily uses the Mask Editor's **Parameters** pane to create the masked subsystem's dialog box (see "Parameters Pane" on page 13-23).



Prompt	Variable	Type	Evaluate	Tunable
Slope:	m	edit	☑	☑
intercept	b	edit	□	☑

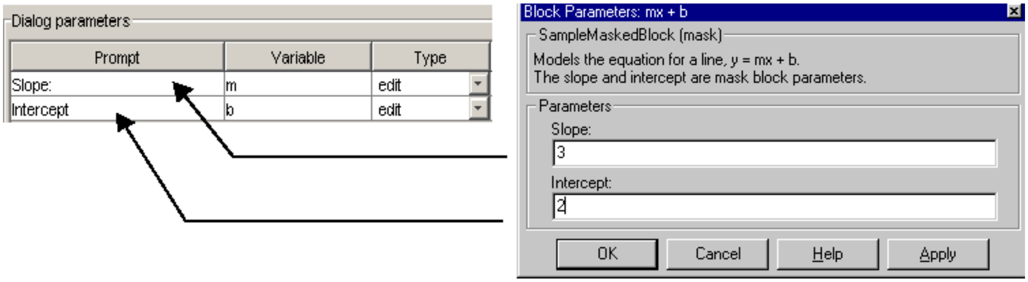
The Mask Editor enables you to specify these attributes of a mask parameter:

- Prompt, the text label that describes the parameter
- Control type, the style of user interface control that determines how parameter values are entered or selected
- Variable, the name of the variable that stores the parameter value

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

The slope and intercept are defined as edit controls. This means that the user types values into edit fields in the mask dialog box. These values are stored in variables in the *mask workspace*. A masked block can access variables in its mask workspace. In this example, the value entered for the slope is assigned to the variable `m`. The Slope block in the masked subsystem gets the value for the slope parameter from the mask workspace.

This figure shows how the slope parameter definitions in the Mask Editor map to the actual mask dialog box parameters.



Dialog parameters

Prompt	Variable	Type
Slope:	m	edit
Intercept	b	edit

Block Parameters: mx + b

SampleMaskedBlock (mask)

Models the equation for a line, *y* = mx + b.
The slope and intercept are mask block parameters.

Parameters

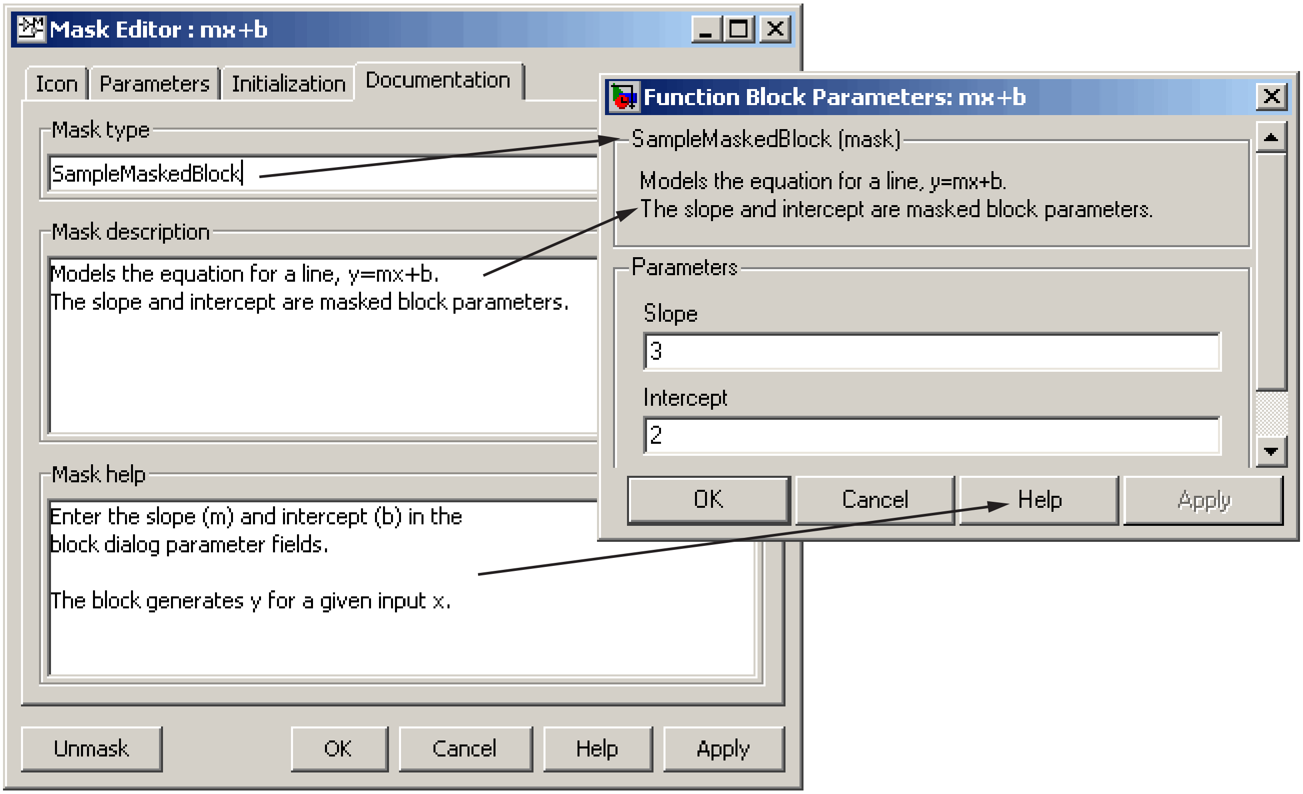
Slope: 3

Intercept: 2

After you create the mask parameters for slope and intercept, click **OK**. Then double-click the Subsystem block to open the newly constructed dialog box. Enter `3` for the **Slope** and `2` for the **Intercept** parameter.

Creating the Block Description and Help Text

The mask type, block description, and help text are defined on the **Documentation** pane (see “Documentation Pane” on page 13-35). For this sample masked block, the pane looks like this.



Mask type: SampleMaskedBlock

Mask description:
Models the equation for a line, y=mx+b.
The slope and intercept are masked block parameters.

Mask help:
Enter the slope (m) and intercept (b) in the
block dialog parameter fields.

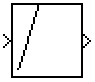
The block generates y for a given input x.

SampleMaskedBlock (mask)
Models the equation for a line, y=mx+b.
The slope and intercept are masked block parameters.

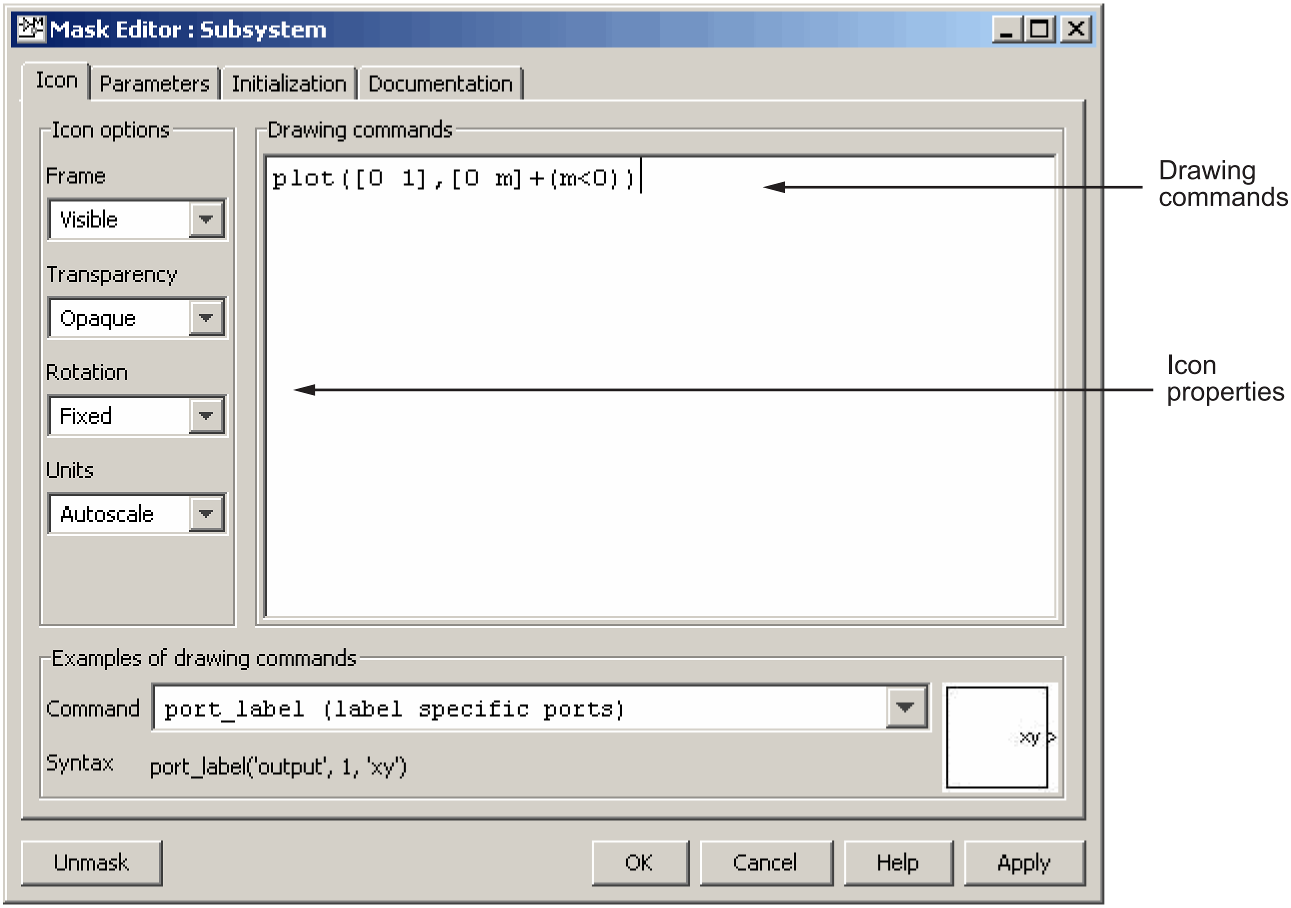
Parameters
Slope: 3
Intercept: 2

Creating the Block Icon

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



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



Icon options

Frame: Visible

Transparency: Opaque

Rotation: Fixed

Units: Autoscale

Drawing commands

```
plot([0 1],[0 m]+(m<0))
```

Examples of drawing commands

Command: port_label (label specific ports)

Syntax: port_label('output', 1, 'xy')

Drawing commands

Icon properties

The drawing command

```
plot([0 1],[0 m]+(m<0))
```

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

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

Select **Normalized** as the **Drawing coordinates** parameter, located at the bottom of the list of icon properties, to specify that the icon be drawn in a frame whose bottom-left corner is (0,0) and whose top-right corner is (1,1). See “Icon Pane” on page 13-19 for more information.

Masking a Block

You can mask almost any block in a Simulink model.

- Use the model editor to mask Subsystem, Model, and S-function blocks. See "Masking a Subsystem" on page 13-14 for more information.
- Use the `set_param` command to mask built-in blocks. See "Masking a Built-in Block" on page 13-16 for more information.

Note Simulink does not support masking port block, e.g., Inport, Outport, Trigger, etc., used in a library.

Note Ensure that the mask parameter names differ from existing block parameter names. You cannot use built-in block parameter names, such as `name`, as mask parameter names. Simulink will return an error.

For information on masking a Subsystem and masking a Built-in Block, see:

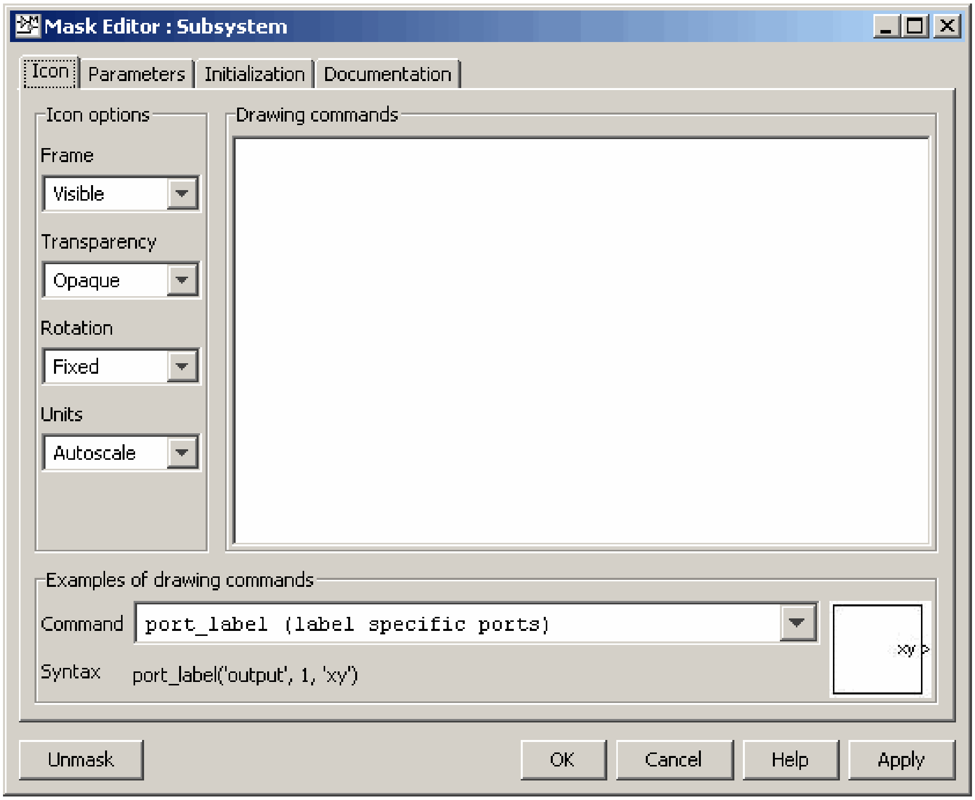
- "Masking a Subsystem" on page 13-14
- "Masking a Built-in Block" on page 13-16

Masking a Subsystem

To mask a Subsystem:

1. Select the block.
2. Select **Mask Subsystem** from the model editor's **Edit** menu or from the block's context menu. (Right-click the subsystem block to display its context menu.)

   The Mask Editor appears.



Frame: Visible

Transparency: Opaque

Rotation: Fixed

Units: Autoscale

Command: `port_label (label specific ports)`

Syntax: port_label('output', 1, 'xy')

See “Mask Editor” on page 13-17 for a detailed description of the Mask Editor.

3 Use the Mask Editor’s tabbed panes to perform any of the following tasks.

- Create a custom icon for the masked subsystem (see “Icon Pane” on page 13-19).
- Create parameters that allow a user to set subsystem options (see “Mask Editor” on page 13-17).
- Initialize the masked subsystem’s parameters
- Create online user documentation for the subsystem

4 Click **Apply** to apply the mask to the subsystem or **OK** to apply the mask and dismiss the Mask Editor.

Masking a Built-in Block

You can reduce the size of your model by directly masking built-in blocks instead of placing them inside a subsystem. To mask a built-in block:

1 Select the block in the model.

2 Type the following command at the MATLAB command prompt:

```
set_param(gcb,'Mask','on')
```

3 Select **Edit Mask** from the model editor's **Edit** menu or from the block's context menu. (Right-click the block to display its context menu.) The Mask Editor appears. See "Mask Editor" on page 13-17 for a detailed description of the Mask Editor.

4 Use the Mask Editor's tabbed panes to perform any of the following tasks.

- Create a custom icon for the masked block (see "Icon Pane" on page 13-19).
- Create a custom mask parameter dialog box for the block (see "Parameters Pane" on page 13-23).
- Initialize the masked block's workspace
- Create online user documentation for the block

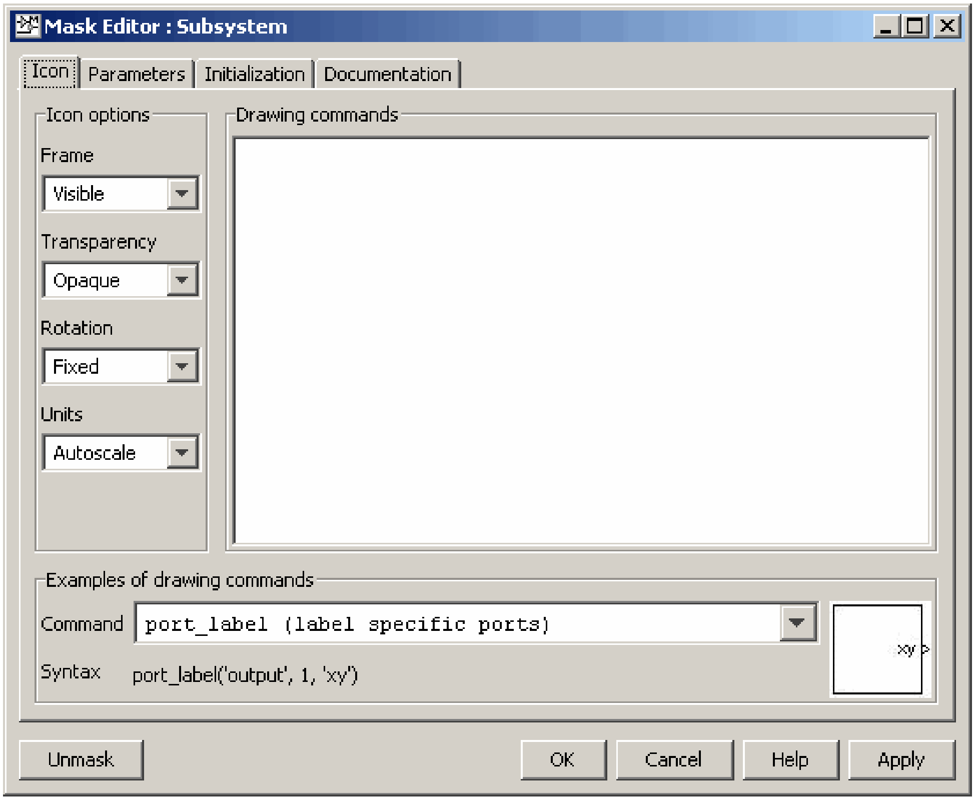
5 Click **Apply** to apply the mask to the block or **OK** to apply the mask and dismiss the Mask Editor.

Mask Editor

The Mask Editor allows you to create or edit a subsystem's mask.

- To create a subsystem mask, select the subsystem block icon and then select **Mask Subsystem** from the **Edit** menu of the model window containing the subsystem's block.
- To create a mask on a built-in block, see "Masking a Built-in Block" on page 13-16.
- To edit an existing mask, select the block's icon and then select **Edit Mask** from the **Edit** menu of the model window containing the subsystem's block.

A Mask Editor like the following appears in either case.



Frame: Visible

Transparency: Opaque

Rotation: Fixed

Units: Autoscale

Command: `port_label (label specific ports)`

Syntax: port_label('output', 1, 'xy')

The Mask Editor contains a set of tabbed panes, each of which enables you to define a feature of the mask:

- The **Icon** pane enables you to define the block icon (see “Icon Pane” on page 13-19).
- The **Parameters** pane enables you to define and describe mask dialog box parameter prompts and name the variables associated with the parameters (see “Parameters Pane” on page 13-23).
- The **Initialization** pane enables you to specify initialization commands (see “Initialization Pane” on page 13-32).
- The **Documentation** pane enables you to define the mask type and specify the block description and the block help (see “Documentation Pane” on page 13-35).

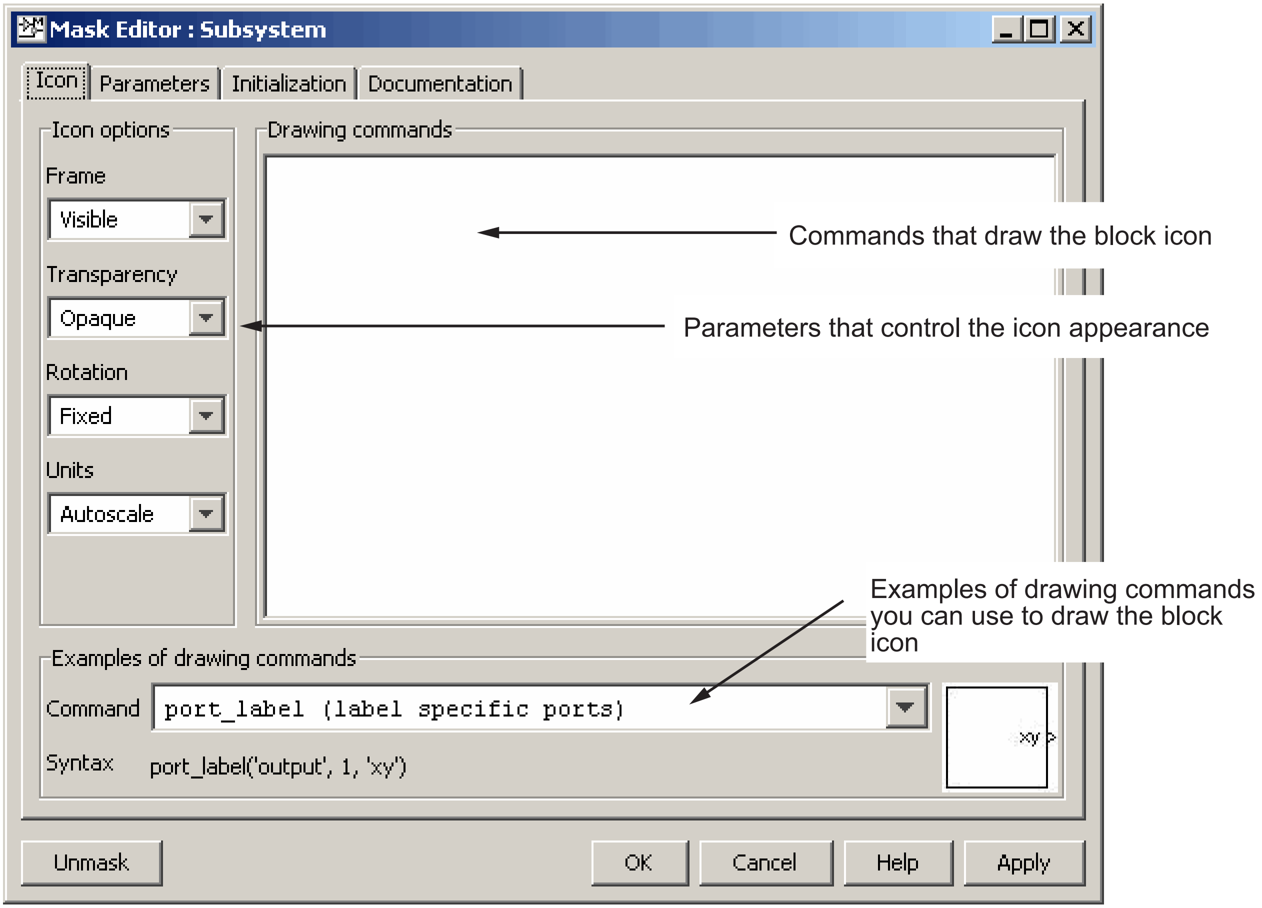
Five buttons appear along the bottom of the Mask Editor:

- The **Unmask** button deactivates the mask and closes the Mask Editor. While the model is still active, Simulink retains the mask information so that you can reactivate it. To reactivate the mask, select the block and choose **Mask Subsystem**. The Mask Editor opens, displaying the previous settings. When you close the model, Simulink discards the inactive mask information. If you want the mask information after this, you will need to recreate it the next time you open the model.
- The **OK** button applies the mask settings on all panes and closes the Mask Editor.
- The **Cancel** button closes the Mask Editor without applying any changes made since you last clicked the **Apply** button.
- The **Help** button displays the contents of this section.
- The **Apply** button creates or changes the mask using the information that appears on all masking panes. The Mask Editor remains open.

To see the system under the mask without unmasking it, select the Subsystem block, then select **Look Under Mask** from the **Edit** menu. This command opens the subsystem. The block’s mask is not affected.

Icon Pane

The Mask Editor's **Icon** pane enables you to create icons that can contain descriptive text, state equations, images, and graphics.



Frame: Visible

Transparency: Opaque

Rotation: Fixed

Units: Autoscale

Command: `port_label (label specific ports)`

Syntax: port_label('output', 1, 'xy')

Commands that draw the block icon

Parameters that control the icon appearance

Examples of drawing commands you can use to draw the block icon

The **Icon** pane contains the following controls.

Drawing commands

This field allows you to enter commands that draw the block's icon. Simulink provides a set of commands that can display text, one or more plots, or show a transfer function (see "Mask Icon Drawing Commands" in the online Simulink reference). You must use only these commands to draw your icon. Simulink executes the drawing commands in the order in which they appear in this field. Drawing commands have access to all variables in the mask workspace. If Simulink cannot successfully execute the drawing commands, the icon displays three question marks.

This example demonstrates how to create an improved icon for the `mx + b` sample masked subsystem discussed earlier in this section. First you must enter the following initialization commands to define the data that enables the drawing command to produce an accurate icon regardless of the shape of the block:

```
pos = get_param(gcb, 'Position');
width = pos(3) - pos(1); height = pos(4) - pos(2);
x = [0, width];
if (m >= 0), y = [0, (m*width)]; end
if (m < 0),  y = [height, (height + (m*width))]; end
```

The drawing command that generates this icon is `plot(x,y)`.

Simulink executes the drawing commands when you

- Load the model
- Run or update the block diagram
- Apply any changes made in the mask parameter dialog box, either by clicking **Apply** or **OK**.
- Apply any changes made in the Mask Editor, either by clicking **Apply** or **OK**
- Make changes to the block diagram that affect the appearance of the block, such as rotating the block
- Copy the masked block within the same model or between different models

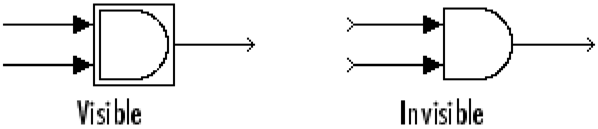
Examples of drawing commands

This panel illustrates the usage of the various icon drawing commands supported by Simulink. To determine the syntax of a command, select the command from the **Command** list. Simulink displays an example of the selected command at the bottom of the panel and the icon produced by the command to the right of the list.

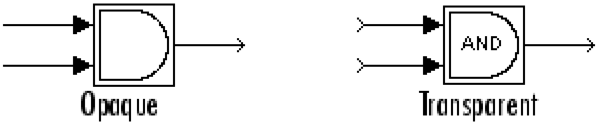
Icon options

These controls allow you to specify the following attributes of the block icon.

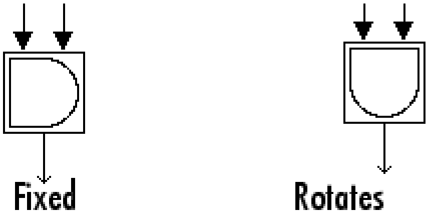
Frame. The icon frame is the rectangle that encloses the block. You can choose to show or hide the frame by setting the **Frame** parameter to `Visible` or `Invisible`. The default is to make the icon frame visible. For example, this figure shows visible and invisible icon frames for an AND gate block.



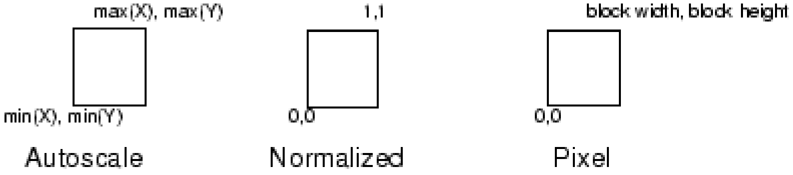
Transparency. The icon can be set to `Opaque` or `Transparent`, either hiding or showing what is underneath the icon. `Opaque`, the default, covers information Simulink draws, such as port labels. This figure shows opaque and transparent icons for an AND gate block. Notice the text on the transparent icon.



Rotation. When the block is rotated or flipped, you can choose whether to rotate or flip the icon or to have it remain fixed in its original orientation. The default is not to rotate the icon. The icon rotation is consistent with block port rotation. This figure shows the results of choosing `Fixed` and `Rotates` icon rotation when the AND gate block is rotated.

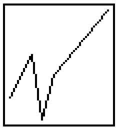


Units. This option controls the coordinate system used by the drawing commands. It applies only to `plot` and `text` drawing commands. You can select from among these choices: `Autoscale`, `Normalized`, and `Pixel`.



- `Autoscale` scales the icon to fit the block frame. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors:

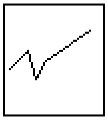
```
X = [0 2 3 4 9]; Y = [4 6 3 5 8];
```



The lower-left corner of the block frame is (0,3) and the upper-right corner is (9,8). The range of the *x*-axis is 9 (from 0 to 9), while the range of the *y*-axis is 5 (from 3 to 8).

- `Normalized` draws the icon within a block frame whose bottom-left corner is (0,0) and whose top-right corner is (1,1). Only X and Y values between 0 and 1 appear. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors:

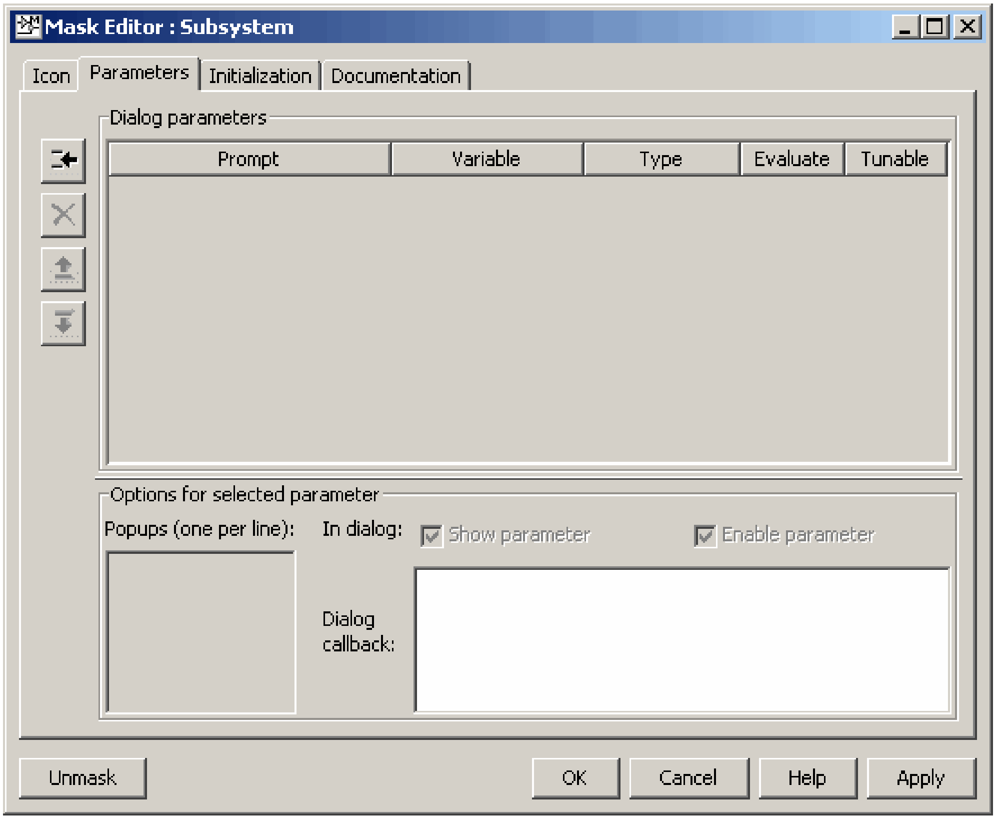
```
X = [.0 .2 .3 .4 .9]; Y = [.4 .6 .3 .5 .8];
```



- `Pixel` draws the icon with X and Y values expressed in pixels. The icon is not automatically resized when the block is resized. To force the icon to resize with the block, define the drawing commands in terms of the block size.

Parameters Pane

The **Parameters** pane allows you to create and modify masked subsystem parameters (mask parameters, for short) that determine the behavior of the masked subsystem.



The **Parameters** pane contains the following elements:

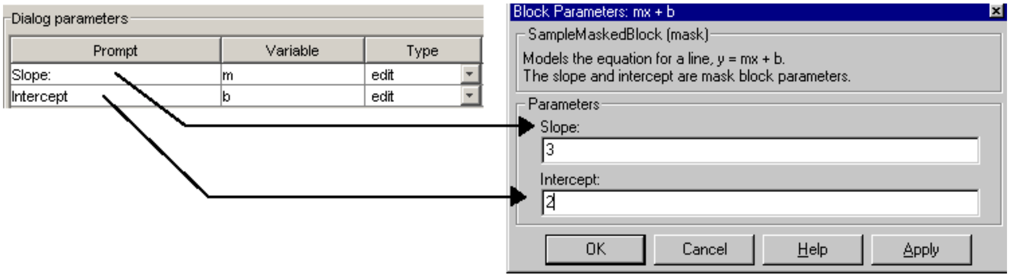
- The **Dialog parameters** panel allows you to select and change the major properties of the mask's parameters (see "Dialog Parameters Panel" on page 13-24).
- The **Options for selected parameter** panel allows you to set additional options for the parameter selected in the **Dialog parameters** panel.

- The buttons on the left side of the **Parameters** pane allow you to add, delete, and change the order of appearance of parameters on the mask's parameter dialog box (see "Dialog Parameters Panel" on page 13-24).

Dialog Parameters Panel

Lists the mask's parameters in tabular form. Each row displays the major attributes of one of the mask's parameters.

Prompt. Text that identifies the parameter on a masked subsystem's dialog box.



Dialog parameters

Prompt	Variable	Type
Slope:	m	edit
Intercept	b	edit

Block Parameters: mx + b

SampleMaskedBlock (mask)

Models the equation for a line, y = mx + b.
The slope and intercept are mask block parameters.

Parameters

Slope: 3

Intercept: 2

Variable. Name of the variable that stores the parameter's value in the mask's workspace (see "Mask Workspace" on page 13-4). You can use this variable as the value of parameters of blocks inside the masked subsystem, thereby allowing the user to set the parameters via the mask dialog box.

Note The set_param and get_param commands are insensitive to case differences in mask variable names. For example, suppose a model named MyModel contains a masked subsystem named A that defines a mask variable named Volume. Then, the following line of code returns the value of the Volume parameter.

```
get_param(MyModel/A, 'voLUME')
```

However, case does matter when using a mask variable as the value of a block parameter inside the masked subsystem. For example, suppose a Gain block inside the masked subsystem A specifies VOLUME as the value of its Gain parameter. This variable name does not resolve in the masked subsystem's workspace, as it contains a mask variable named Volume. If the base workspace does not contain a variable named VOLUME, simulating MyModel produces an error.

Type. Type of control used to edit the value of this parameter. The control appears on the mask's parameter dialog box following the parameter prompt. The button that follows the type name in the **Parameters** pane pops up a list of the controls supported by Simulink (see "Control Types" on page 13-29). To change the current control type, select another type from the list.

Evaluate. If checked, this option causes Simulink to evaluate the expression entered by the user before it is assigned to the variable. Otherwise, Simulink treats the expression itself as a string value and assigns it to the variable. For example, if the user enters the expression gain in an edit field and the **Evaluate** option is checked, Simulink evaluates gain and assigns the result to the variable. Otherwise, Simulink assigns the string 'gain' to the variable. See "Check Box Control" on page 13-30 and "Pop-Up Control" on page 13-30 for information on how this option affects evaluation of the parameters.

If you need both the string entered and the evaluated value, clear the **Evaluate** option. To get the value of a base workspace variable entered as the literal value of the mask parameter, use the MATLAB evalin command in the mask initialization code. For example, suppose the user enters the string 'gain' as the literal value of the mask parameter k where gain is the name of a base workspace variable. To obtain the value of the base workspace variable, use the following command in the mask's initialization code:

```
value = evalin('base', k)
```

Tunable. Selecting this option allows a user to change the value of the mask parameter while a simulation is running.

Note Simulink ignores this setting if the block being masked is a source block, i.e., the block has outputs but no input ports. In such a case, even if this option is selected, you cannot tune the parameter while a simulation is running. See "Changing Source Block Parameters During Simulation" on page 4-12 for more information.

Options for Selected Parameter Panel

This panel allows you to set additional options for the parameter selected in the **Dialog parameters** table.

Show parameter. The selected parameter appears on the masked block's parameter dialog box only if this option is checked (the default).

Enable parameter. Clearing this option grays the selected parameter's prompt and disables its edit control. This means that the user cannot set the value of the parameter.

Popups. This field is enabled only if the edit control for the selected parameter is a pop-up. Enter the values of the pop-up control in this field, each on a separate line.

Callback. Enter MATLAB code that you want Simulink to execute when a user applies a change to the selected parameter, i.e., selects the **Apply** or **OK** button on the mask dialog box. You can use such callbacks to create dynamic dialogs, i.e., dialogs whose appearance changes, depending on changes to control settings made by the user, e.g., enabling an edit field when a user checks a check box (see "Creating Dynamic Mask Parameter Dialog Boxes" on page 13-39 more information).

Note Callbacks are not intended to be used to alter the contents of a masked subsystem. Altering a masked subsystem's contents in a callback, for example by adding or deleting blocks or changing block parameter values, can trigger errors during model update or simulation. If you need to modify the contents of a masked subsystem, use the mask's initialization code to perform the modifications (see "Initialization Pane" on page 13-32).

The callback can create and reference variables only in the block's base workspace. If the callback needs the value of a mask parameter, it can use `get_param` to obtain the value, e.g.,

```
if str2num(get_param(gcb, 'g'))<0
 error('Gain is negative.')
end
```

Simulink executes the callback commands when you

- Open the mask parameter dialog box. Callback commands execute top down, starting with the top mask dialog parameter
- Modify a parameter value in the mask parameter dialog box then change the cursor's focus, i.e., press the **Tab** key or click into another field in the dialog

  Note The callback commands are not executed when the parameter value is modified using the `set_param` command.

- Modify the parameter value, either in the mask parameter dialog box or via a call to `set_param`, than apply the change by clicking **Apply** or **OK**. Any mask initialization commands execute after the callback commands. (See "Initialization Pane" on page 13-32.)
- Cancel any applied changes made in the mask dialog box by clicking **Cancel**.
- Hover over the masked block to see the block's data tip, when the data tip contains parameter names and values. The callback executes again when the block data tip disappears.

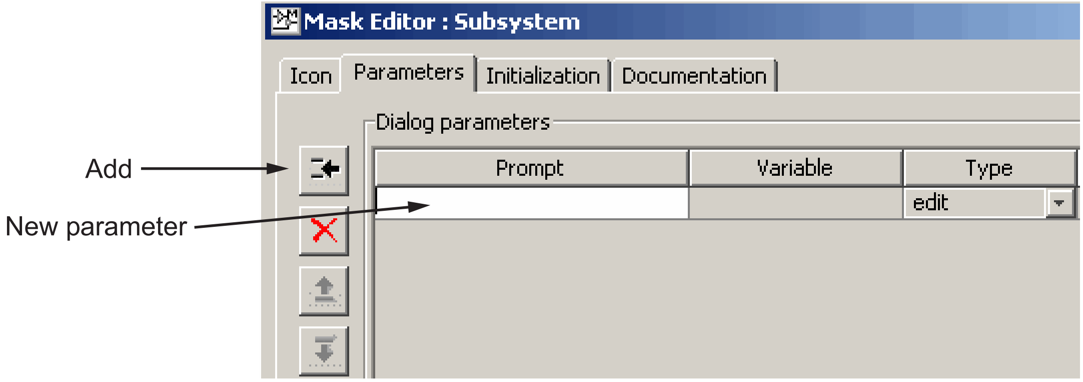
Note The callback commands do not execute if the mask dialog box is open when the block data tip appears.

For information on debugging dialog callbacks, see "Debugging Masks" on page 13-51.

Parameter Buttons

The following sections explain the purpose of the buttons that appear on the **Parameters** pane in the order of their appearance from the top of the pane.

Add Button. Adds a parameter to the mask's parameter list. The newly created parameter appears in the adjacent **Dialog parameters** table.



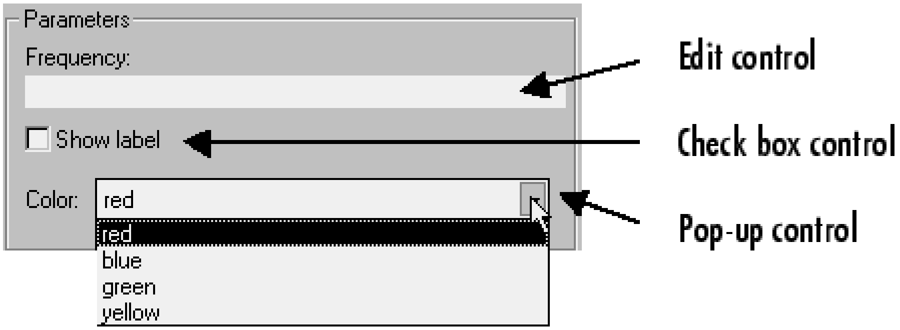
Delete Button. Deletes the parameter currently selected in the **Dialog parameters** table.

Up Button. Moves the currently selected parameter up one row in the **Dialog parameters** table. Dialog parameters appear in the mask's parameter dialog box (see "Mask Parameter Dialog Box" on page 13-3) in the same order in which they appear in the **Dialog parameters** table. This button (and the next) thus allows you to determine the order in which parameters appear on the dialog box.

Down Button. Moves the currently selected parameter down one row in the **Dialog parameters** table and hence down one position on the mask's parameter dialog box.

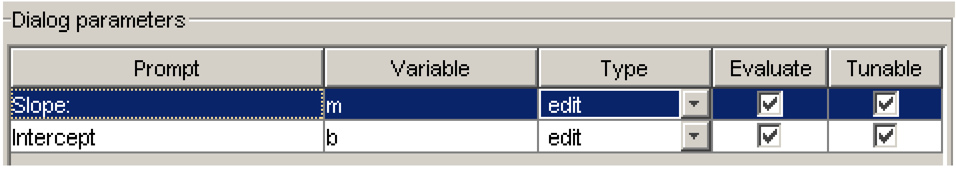
Control Types

Simulink enables you to choose how parameter values are entered or selected. You can create three styles of controls: edit fields, check boxes, and pop-up controls. For example, this figure shows the parameter area of a mask dialog box that uses all three styles of controls (with the pop-up control open).



Edit Control

An *edit field* enables the user to enter a parameter value by typing it into a field. This figure shows how the prompt for the sample edit control was defined.


Dialog parameters

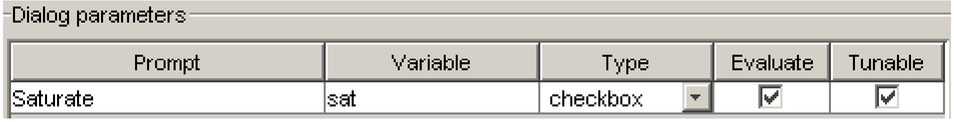
Prompt	Variable	Type	Evaluate	Tunable
Slope:	m	edit	☑	☑
Intercept	b	edit	☑	☑

The value of the variable associated with the parameter is determined by the **Evaluate** option.

Evaluate	Value
On	The result of evaluating the expression entered in the field
Off	The actual string entered in the field

Check Box Control

A *check box* enables the user to choose between two alternatives by selecting or deselecting a check box. This figure shows how the sample check box control is defined.



Dialog parameters

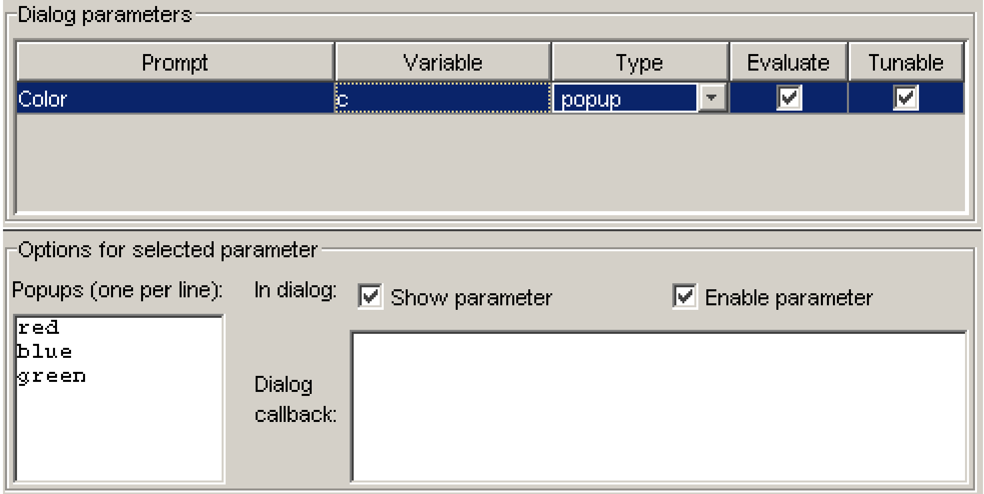
Prompt	Variable	Type	Evaluate	Tunable
Saturate	sat	checkbox	☑	☑

The value of the variable associated with the parameter depends on whether the **Evaluate** option is selected.

Control State	Evaluated Value	Literal Value
Selected	1	`'on'`
Unselected	0	`'off'`

Pop-Up Control

A *pop-up* enables the user to choose a parameter value from a list of possible values. Specify the values in the **Popups** field on the **Options for selected parameter** pane (see “Popups” on page 13-26). The following example shows a pop-up parameter.



Dialog parameters

Prompt	Variable	Type	Evaluate	Tunable
Color	c	popup	☑	☑

Options for selected parameter

Popups (one per line):

```
red
blue
green
```

In dialog: ☑ Show parameter ☑ Enable parameter

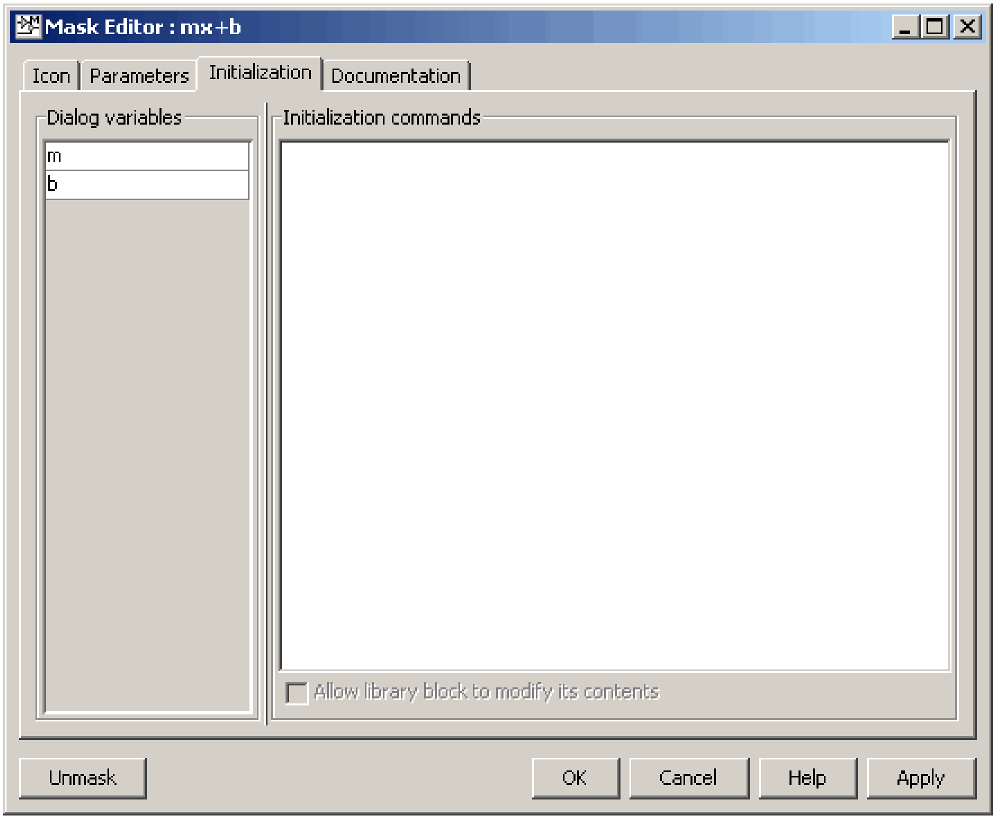
Dialog callback:

The value of the variable associated with the parameter (`Color`) depends on the item selected from the pop-up list and whether the **Evaluate** option is checked (on).

Evaluate	Value
On	Index of the value selected from the list, starting with `1`. For example, if the third item is selected, the parameter value is `3`.
Off	String that is the value selected. If the third item is selected, the parameter value is `'green'`.

Initialization Pane

The **Initialization** pane allows you to enter MATLAB commands that initialize the masked subsystem.



Simulink executes the initialization commands when you

- Load the model
- Start a simulation or update the model's block diagram
- Make changes to the block diagram that affect the appearance of the block, such as rotating the block

- Copy the masked subsystem within the same model or between different models
- Apply any changes to the block's dialog that affect the block's appearance or behavior, such as changing the value of a mask parameter on which the block's icon drawing code depends.

The **Initialization** pane includes the following controls.

Dialog variables

The **Dialog variables** list displays the names of the variables associated with the subsystem's mask parameters, i.e., the parameters defined in the **Parameters** pane. You can copy the name of a parameter from this list and paste it into the adjacent **Initialization commands** field, using the Simulink keyboard copy and paste commands. You can also use the list to change the names of mask parameter variables. To change a name, double-click the name in the list. An edit field containing the existing name appears. Edit the existing name and press **Enter** or click outside the edit field to confirm your changes.

Initialization commands

Enter the initialization commands in this field. You can enter any valid MATLAB expression, consisting of MATLAB functions and scripts, operators, and variables defined in the mask workspace. Initialization commands cannot access base workspace variables. Terminate initialization commands with a semicolon to avoid echoing results to the Command Window. For information on debugging initialization commands, see "Debugging Masks" on page 13-51.

Allow library block to modify its contents

This check box is enabled only if the masked subsystem resides in a library. Checking this option allows the block's initialization code to modify the contents of the masked subsystem, i.e., it lets the code add or delete blocks and set the parameters of those blocks. Otherwise, Simulink generates an error when a masked library block tries to modify its contents in any way. To set this option at the MATLAB prompt, select the self-modifying block and enter the following command.

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

Then save the block.

Initialization Command Limitations

Mask initialization commands must observe the following rules:

- Do not use initialization code to create dynamic mask dialogs, i.e., dialogs whose appearance or control settings change depending on changes made to other control settings. Instead, use the mask callbacks provided specifically for this purpose (see "Creating Dynamic Mask Parameter Dialog Boxes" on page 13-39 for more information).
- Avoid prefacing variable names in initialization commands with `L_` and `M_` to prevent undesirable results. Simulink reserves these specific prefixes for use with its own internal variable names.
- Avoid using `set_param` commands to set parameters of blocks residing in masked subsystems that reside in the masked subsystem being initialized. Trying to set parameters of blocks in lower-level masked subsystems can trigger unresolvable symbol errors if lower-level masked subsystems reference symbols defined by higher-level masked subsystems. Suppose, for example, a masked subsystem A contains masked subsystem B, which contains Gain block C, whose Gain parameter references a variable defined by B. Suppose also that subsystem A's initialization code contains the command

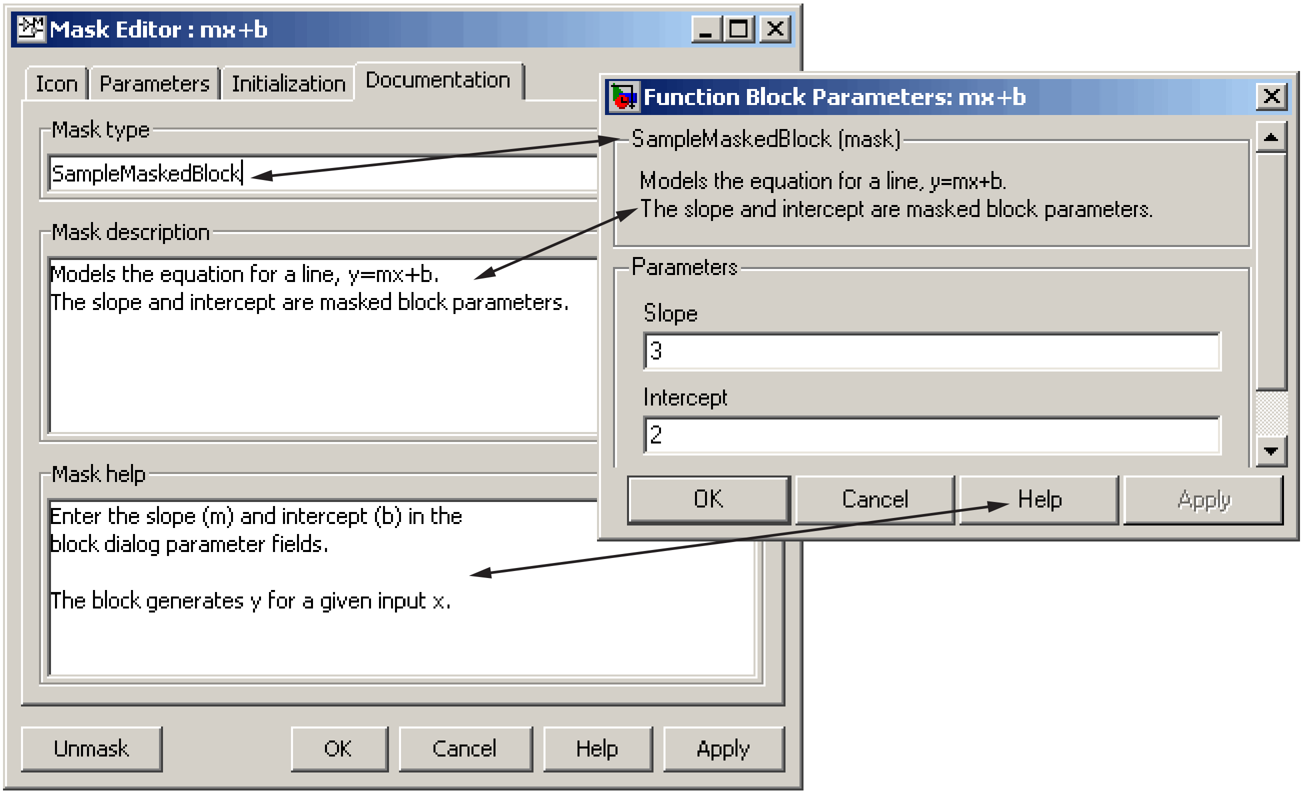
  ```
  set_param([gcb '/B/C'], 'SampleTime', '-1');
  ```

  Simulating or updating a model containing A causes an unresolvable symbol error.

Documentation Pane

The **Documentation** pane enables you to define or modify the type, description, and help text for a masked block.

This figure shows how fields on the **Documentation** pane correspond to the `mx + b` sample mask block's dialog box.



Mask Editor : mx+b

Mask type

SampleMaskedBlock

Mask description

Models the equation for a line, y=mx+b.
The slope and intercept are masked block parameters.

Mask help

Enter the slope (m) and intercept (b) in the
block dialog parameter fields.

The block generates y for a given input x.

Function Block Parameters: mx+b

SampleMaskedBlock (mask)

Models the equation for a line, y=mx+b.
The slope and intercept are masked block parameters.

Parameters

Slope

3

Intercept

2

Mask Type Field

The mask type is a block classification used only for purposes of documentation. It appears in the block's dialog box and on all Mask Editor panes for the block. You can choose any name you want for the mask type. When Simulink creates the block's dialog box, it adds "(mask)" after the mask type to differentiate masked blocks from built-in blocks.

Mask Description Field

The block description is informative text that appears in the block's dialog box in the frame under the mask type. If you are designing a system for others to use, this is a good place to describe the block's purpose or function.

Simulink automatically wraps long lines of text. You can force line breaks by using the **Enter** or **Return** key.

Block Help Field

You can provide help text that is displayed when the **Help** button is clicked on the masked block's dialog box. If you create models for others to use, this is a good place to explain how the block works and how to enter its parameters.

You can include user-written documentation for a masked block's help. You can specify any of the following for the masked block help text:

- URL specification (a string starting with `http:`, www, `file:`, `ftp:`, or `mailto:`)
- `web` command (launches a browser)
- `eval` command (evaluates a MATLAB string)
- HTML-tagged text to be displayed in a Web browser

Simulink examines the first line of the masked block help text. If Simulink detects a URL specification, for example,

```
http://www.mathworks.com
```

or

```
file:///c:/mydir/helpdoc.html
```

Simulink displays the specified file in the browser. If Simulink detects a `web` command, for example,

```
web([docroot '/My Blockset Doc/' get_param(gcb,'MaskType')...
'.html'])
```

or an eval command, for example,

```
eval('!Word My_Spec.doc')
```

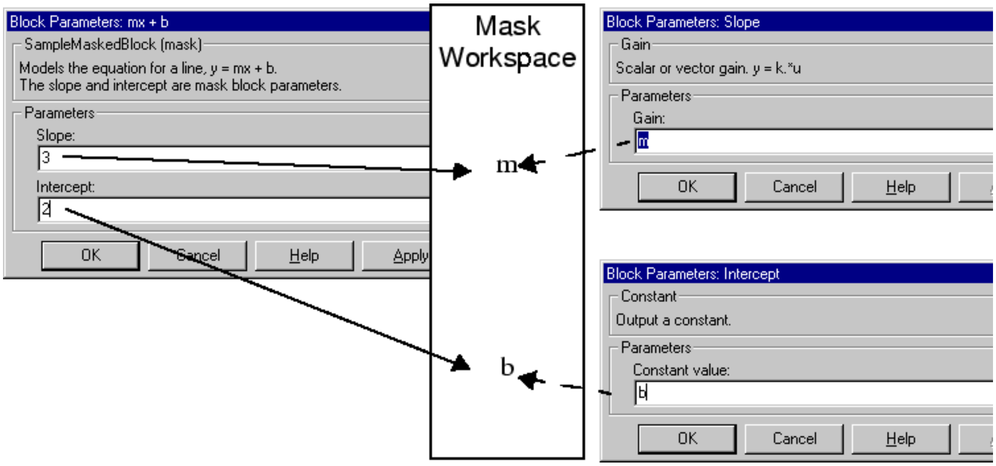
Simulink executes the specified command. Otherwise, Simulink displays the contents of the **Block Help** field, which can include HTML tags, in the browser.

Note, if you enter HTML-tagged text, Simulink copies that text into a temporary directory and displays it from that temporary directory. If you want to include an image (for example, with the img tag) with that text, you need to place the image file in that temporary directory. (You can use tempdir to find the temporary directory for your system.) Alternatively, you can save the HTML-tagged text into an HTML file (such as hello.html) in the current directory and display that file directly (for example, web('hello.html', '-helpbrowser')). This method enables you to place referenced image files in the same directory as the HTML file.

Linking Mask Parameters to Block Parameters

The variables associated with mask parameters allow you to link mask parameters with block parameters. This in turn allows a user to use the mask to set the values of parameters of blocks inside the masked subsystem.

To link the parameters, open the block's parameter dialog box and enter an expression in the block parameter's value field that uses the mask parameter. The `mx + b` masked subsystem, described earlier in this chapter, uses this approach to link the Slope and Intercept mask parameters to corresponding parameters of a Gain and Constant block, respectively, that reside in the subsystem.



Block Parameters: mx + b

SampleMaskedBlock (mask)

Models the equation for a line, y = mx + b.
The slope and intercept are mask block parameters.

Parameters

Slope: 3

Intercept: 2

Block Parameters: Slope

Gain

Scalar or vector gain. y = k.*u

Parameters

Gain: m

Block Parameters: Intercept

Constant

Output a constant.

Parameters

Constant value: b

You can use a masked block's initialization code to link mask parameters indirectly to block parameters. In this approach, the initialization code creates variables in the mask workspace whose values are functions of the mask parameters and that appear in expressions that set the values of parameters of blocks concealed by the mask.

Creating Dynamic Mask Parameter Dialog Boxes

Simulink allows you to create dialogs for masked blocks whose appearance changes in response to user input. Features of masked dialog boxes that can change in this way include

- Visibility of parameter controls

  Changing a parameter can cause the control for another parameter to appear or disappear. The dialog expands or shrinks when a control appears or disappears, respectively.

- Enabled state of parameter controls

  Changing a parameter can cause the control for another parameter to be enabled or disabled for input. Simulink grays a disabled control to indicate visually that it is disabled.

- Parameter values

  Changing a parameter can cause related parameters to be set to appropriate values.

Creating a dynamic masked dialog entails using the mask editor in combination with the `set_param` command. Specifically, you use the Mask Editor to define the dialog's parameters, both static and dynamic. For each dynamic parameter, you enter a callback function that defines the dialog's response to changes to that parameter (see "Callback" on page 13-26). The callback function can in turn use the `set_param` command to set mask dialog parameters that affect the appearance and settings of other controls on the dialog (see "Setting Masked Block Dialog Parameters" on page 13-39). Finally, you save the model or library containing the masked subsystem to complete the creation of the dynamic masked dialog.

Setting Masked Block Dialog Parameters

Simulink defines a set of masked block parameters that define the current state of the masked block's dialog. You can use the mask editor to inspect and set many of these parameters. The Simulink `get_param` and `set_param` commands also let you inspect and set mask dialog parameters. The advantage? The `set_param` command allows you to set parameters and hence change a dialog's appearance while the dialog is open. This in turn allows you to create dynamic masked dialogs.

For example, you can use the set_param command in mask callback functions to be invoked when a user changes the values of user-defined parameters. The callback functions in turn can use set_param commands to change the values of the masked dialog's predefined parameters and hence its state, for example, to hide, show, enable, or disable a user-defined parameter control.

The following example creates a mask dialog with two parameters. The first parameter is a pop-up menu that selects one of three gain values: 2, 5, or User-defined. The selection in this pop-up menu determines the visibility of an edit field used to specify the user-defined gain. See Predefined Masked Dialog Parameters for more information on the syntax and use of the various masked dialog parameters used in this example.

1. Mask a subsystem as described in steps one and two in Masking a Subsystem.

2. Select the **Parameters** pane on the Mask Editor.

3. Add a parameter.

   - Enter Gain: in the **Prompt** field
   - Enter gain in the **Variable** field
   - Select popup in the **Type** field

4. Enter the following three values in the **Popups (one per line)** field:

   ```
   2
   5
   User-defined
   ```

5. Enter the following code in the **Dialog callback** field:

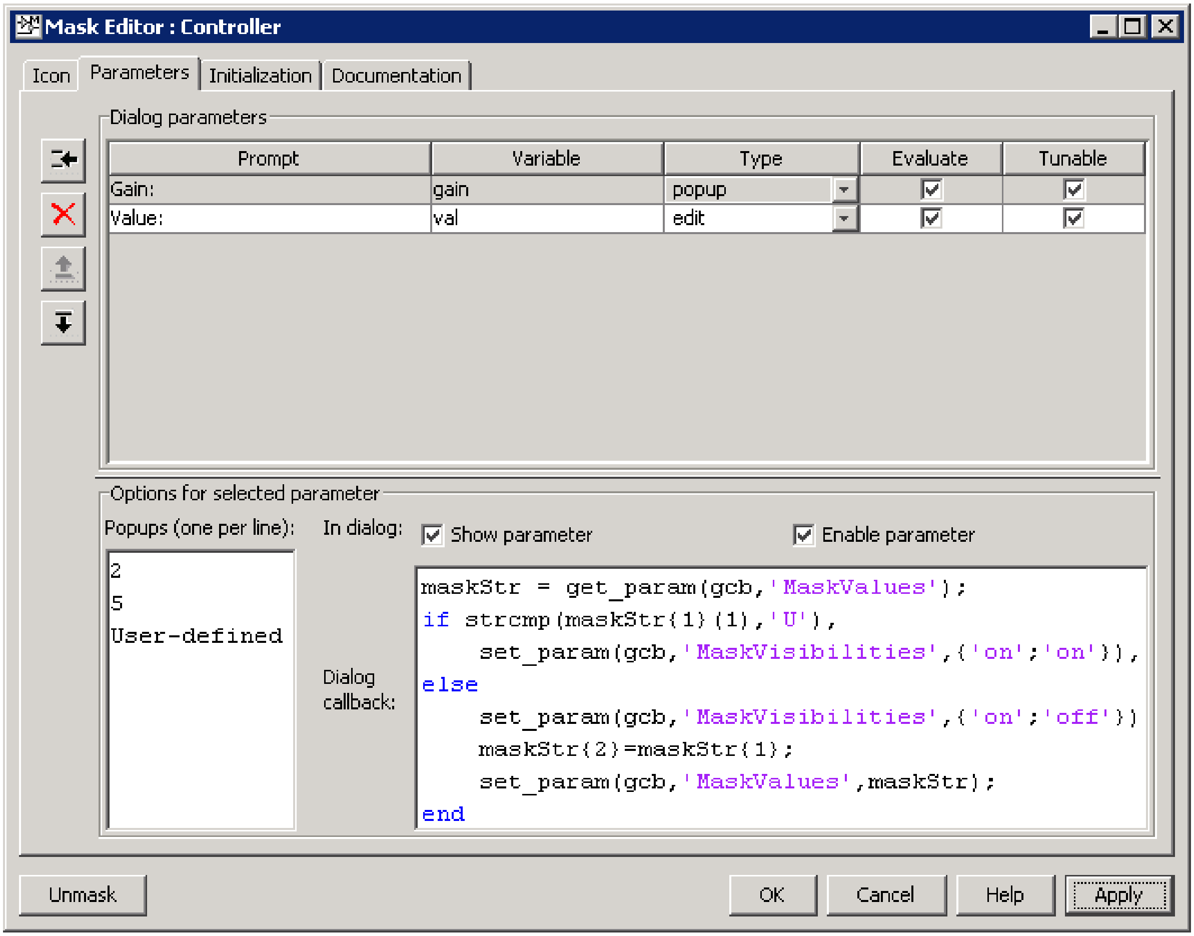
   ```
   % Get the mask parameter values. This is a cell
   %   array of strings.
   maskStr = get_param(gcb,'MaskValues');

   % The pop-up menu is the first mask parameter.
   %   Check the value selected in the pop-up
   ```

```
if strcmp(maskStr{1}(1),'U'),

    % Set the visibility of both parameters on when
    %   User-defined is selected in the pop-up.

    set_param(gcb,'MaskVisibilities',{'on';'on'}),

else

    % Turn off the visibility of the Value field
    %   when User-defined is not selected.

    set_param(gcb,'MaskVisibilities',{'on';'off'}),

    % Set the string in the Values field equal to the
    % string selected in the Gain pop-up menu.

    maskStr{2}=maskStr{1};
    set_param(gcb,'MaskValues',maskStr);
end
```

6 Add a second parameter.

- Enter `Value:` in the **Prompt** field
- Enter `val` in the **Variable** field
- Uncheck **Show parameter** in the **Options for selected parameter** group. This turns the visibility of this parameter off, by default.

7 Select **Apply** on the Mask Editor. The Mask Editor now looks like this when the `gain` parameter is selected and comments are removed from the mask callback code:



Prompt	Variable	Type	Evaluate	Tunable
Gain:	gain	popup	☑	☑
Value:	val	edit	☑	☑

Popups (one per line):

```
2
5
User-defined
```

In dialog: ☑ Show parameter ☑ Enable parameter

Dialog callback:

```
maskStr = get_param(gcb,'MaskValues');
if strcmp(maskStr{1}(1),'U'),
    set_param(gcb,'MaskVisibilities',{'on';'on'}),
else
    set_param(gcb,'MaskVisibilities',{'on';'off'})
    maskStr{2}=maskStr{1};
    set_param(gcb,'MaskValues',maskStr);
end
```

Double-clicking on the new masked subsystem opens the Mask Parameters dialog box. Selecting `2` or `5` for the **Gain** parameter hides the **Value** parameter, while selecting `User-defined` makes the **Value** parameter visible. Note that any blocks in the masked subsystem that need the gain value should reference the mask variable `val` as the `set_param` in the `else` code assures that `val` contains the current value of the gain when `2` or `5` is selected in the popup.

Predefined Masked Dialog Parameters

Simulink associates the following predefined parameters with masked dialogs.

MaskCallbacks

The value of this parameter is a cell array of strings that specify callback expressions for the dialog's user-defined parameter controls. The first cell defines the callback for the first parameter's control, the second for the second parameter control, etc. The callbacks can be any valid MATLAB expressions, including expressions that invoke M-file commands. This means that you can implement complex callbacks as M-files.

You can use either the mask editor or the MATLAB command line to specify mask callbacks. To use the mask editor to enter a callback for a parameter, enter the callback in the **Callback** field for the parameter.

The easiest way to set callbacks for a mask dialog at the MATLAB command is to first select the corresponding masked dialog in a model or library window and then to issue a `set_param` command at the MATLAB command line. For example, the following code

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

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

MaskDescription

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

MaskDisplay

The value of this parameter is string that specifies the drawing commands for the block's icon.

MaskEnables

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

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

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

would disable the third control of the currently open masked block's dialog. Simulink colors disabled controls gray to indicate visually that they are disabled.

MaskInitialization

The value of this parameter is string that specifies the initialization commands for the mask workspace.

MaskPrompts

The value of this parameter is a cell array of strings that specify prompts for user-defined parameters. The first cell defines the prompt for the first parameter, the second for the second parameter, etc.

MaskType

The value of this parameter is the mask type of the block associated with this dialog.

MaskValues

The value of this parameter is a cell array of strings that specify the values of user-defined parameters for this dialog. The first cell defines the value for the first parameter, the second for the second parameter, etc.

MaskVisibilities

The value of this parameter is a cell array of strings that specify the visibility of the user-defined parameter controls for this dialog. The first cell defines the visibility of the control for the first parameter, the second for the second parameter, etc. A value of `'on'` indicates that the corresponding control is visible; a value of `'off'` indicates that the control is hidden.

You can hide or show user-defined parameter controls dynamically by setting this parameter in the callback for a control. For example, the following command in a callback

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

would hide the control for the currently selected block's second user-defined mask parameter. Simulink expands or shrinks a dialog to show or hide a control, respectively.

Note For a full list of predefined masked block parameters see the Mask Parameters reference page.

Masking Library Blocks

You mask blocks in a library (see “Working with Block Libraries” on page 4-35) just as you would mask blocks in a Simulink model (see “Masking a Block” on page 13-14). Masking a library block allows you to hide the contents of the block and create a custom mask parameter dialog box for every copy of the library block. See the following sections for more information on masking blocks in a library.

- “Specifying Default Values for Library Block Mask Parameters” on page 13-46
- “Creating Self-Modifying Masks for Library Blocks” on page 13-46

Specifying Default Values for Library Block Mask Parameters

When you create a mask for a library block, the edit fields on the mask parameter dialog box, by default, have a value of zero, check boxes are not selected, and drop-down lists select the first item in the list. To change default parameter values in a masked library block:

1. Unlock the library (see “Modifying a Library” on page 4-37).
2. Double-click the block to access the mask parameter dialog box.
3. Fill in the desired default values or change check box or drop-down list settings, then apply the changes and close the dialog box.
4. Save the library.

When the block is copied into a model and opened, the default values appear on the block’s dialog box.

Creating Self-Modifying Masks for Library Blocks

You can create masked library blocks that can modify their structural contents. These self-modifying masks allow you to

- Modify the contents of a masked subsystem based on parameters in the mask parameter dialog box or when the subsystem is initially dragged from the library into a new model.
- Vary the number of ports on a multiport S-function that resides in a library.

Creating Self-Modifying Masks Using the Mask Editor

To create a self-modifying mask using the Mask Editor:

1. Unlock the library (see “Modifying a Library” on page 4-37).
2. Select the block in the library.
3. Select **Edit Mask** from the model editor’s **Edit** menu or from the block’s context menu. (Right-click the block to display its context menu.) The Mask Editor opens.
4. In the Mask Editor’s **Initialization** pane, select the **Allow library block to modify its contents** option.
5. Click **Apply** to apply the change or **OK** to apply the change and dismiss the Mask Editor.

Creating Self-Modifying Masks from the Command Line

To create a self-modifying mask from the command line:

1. Unlock the library using the following command:

   ```
   set_param(gcs,'Lock','off')
   ```

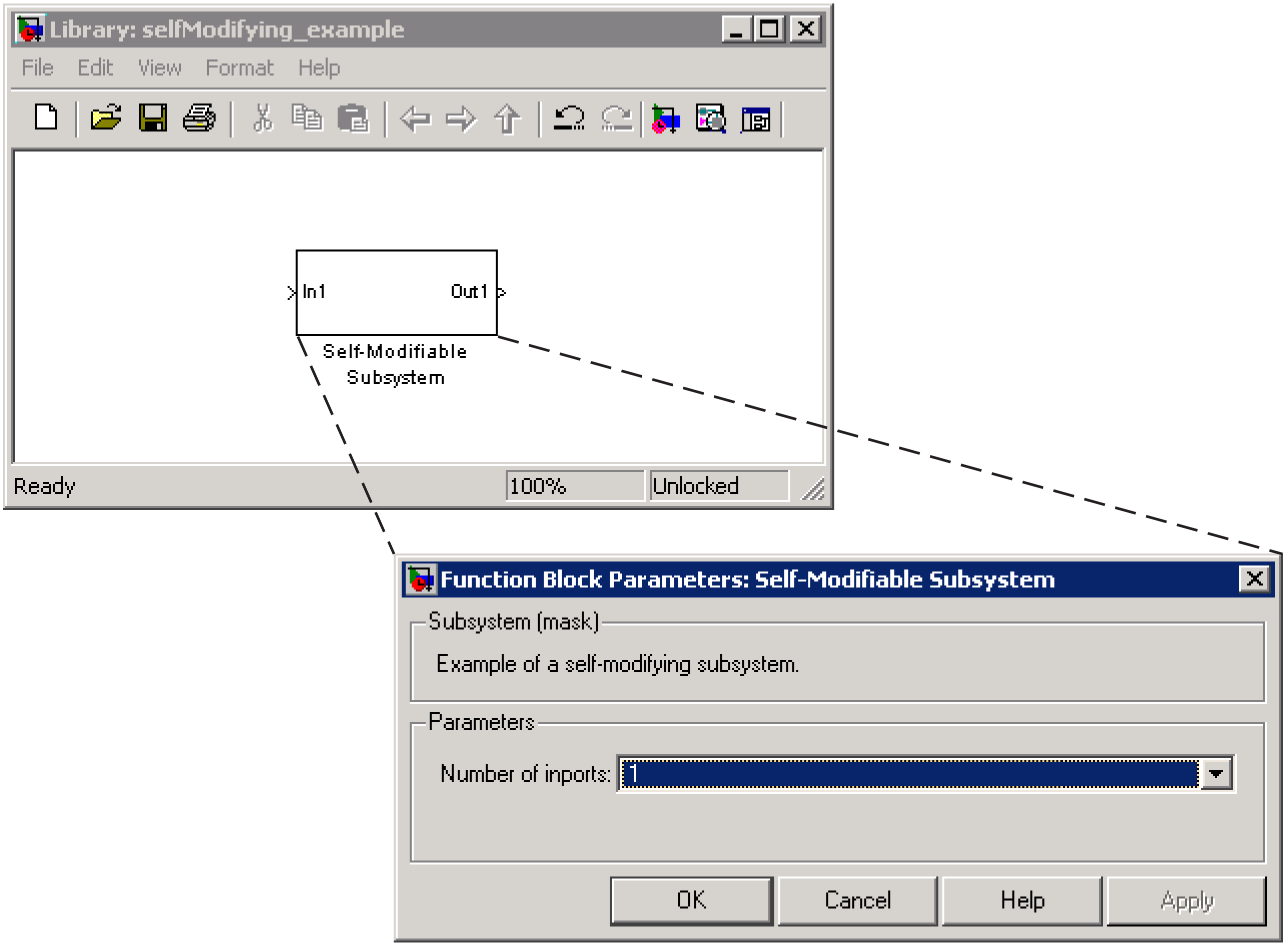
2. Specify that the block is self-modifying by using the following command:

   ```
   set_param(block_name,'MaskSelfModifiable','on')
   ```

   where `block_name` is the full path to the block in the library.

Self-Modifying Mask Example

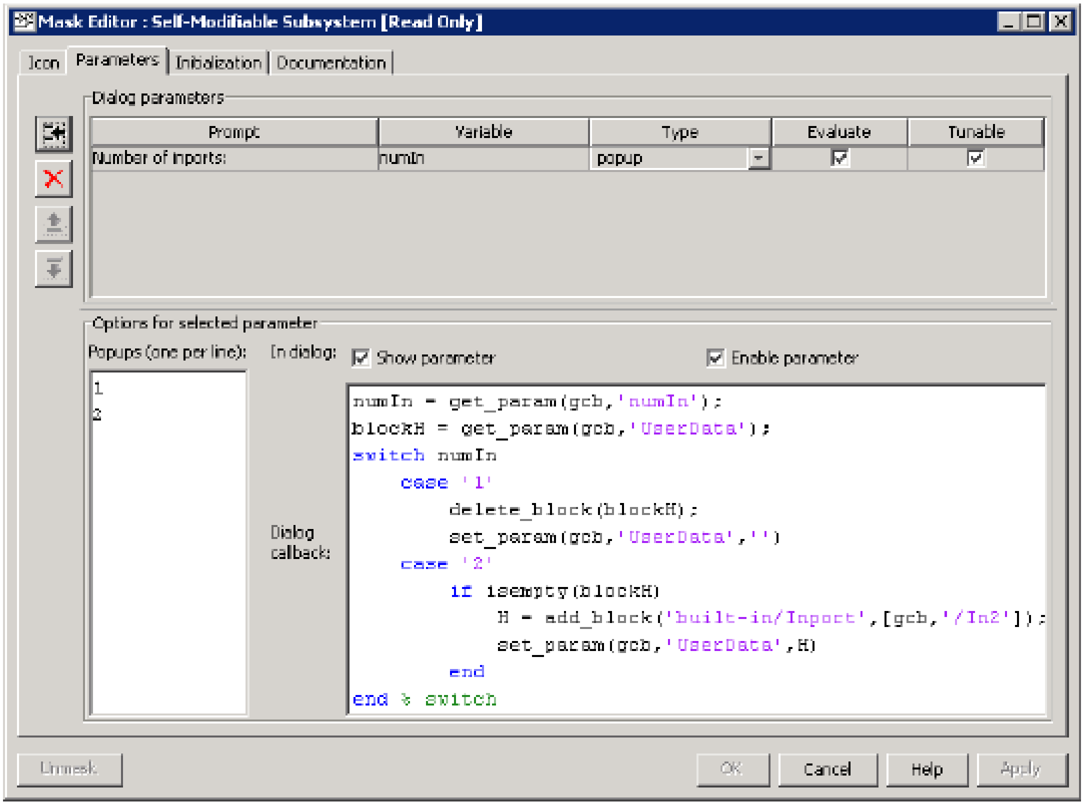
The library `selfModifying_example.mdl` contains a masked subsystem that modifies its number of input ports based on a selection made in the subsystem’s mask parameter dialog box.


Library: selfModifying_example
File Edit View Format Help
In1
Out1
Self-Modifiable
Subsystem
Ready
100%
Unlocked
Function Block Parameters: Self-Modifiable Subsystem
Subsystem (mask)
Parameters
OK
Cancel
Help
Apply

Example of a self-modifying subsystem.

Number of inports: 1

Select the subsystem then select **View Mask** from the model editor's **Edit** menu or from the block's context menu. (Right-click the block to display its context menu.) The Mask Editor opens. The Mask Editor's **Parameters** pane defines one mask parameter variable `numIn` that stores the value for the **Number of inports** option. This mask parameter's dialog callback adds or removes Input ports inside the masked subsystem based on the selection made in the **Number of inports** list.


Mask Editor : Self-Modifiable Subsystem [Read Only]
Icon
Parameters
Initialization
Documentation
Dialog parameters
Options for selected parameter
Unmask
OK
Cancel
Help
Apply

Prompt	Variable	Type	Evaluate	Tunable
Number of inports:	numIn	popup	☑	☑

Popups (one per line):

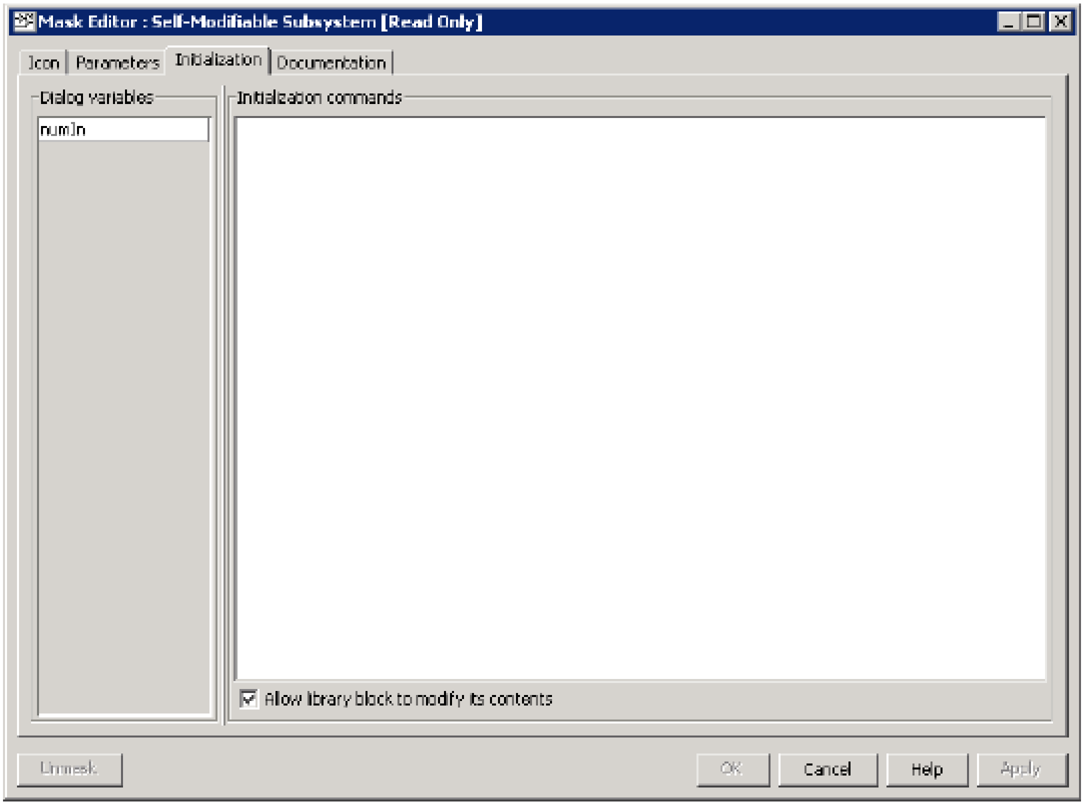
1
2

In dialog: ☑ Show parameter ☑ Enable parameter

Dialog callback:

```
numIn = get_param(gcb,'numIn');
blockH = get_param(gcb,'UserData');
switch numIn
    case '1'
        delete_block(blockH);
        set_param(gcb,'UserData','')
    case '2'
        if isempty(blockH)
            H = add_block('built-in/Inport',[gcb,'/In2']);
            set_param(gcb,'UserData',H)
        end
end % switch
```

To allow the dialog callback to function properly, the **Allow library block to modify its contents** option on the Mask Editor's **Initialization** pane is selected. If this option was not selected, copies of the library block could not modify their structural contents and changing the selection in the **Number of inports** list would produce an error.



numIn

☑ Allow library block to modify its contents

Debugging Masks

You can use MATLAB tools to debug mask initialization commands and dialog callbacks either in the Mask Editor or the MATLAB Editor/Debugger.

Note You cannot debug icon drawing commands using the MATLAB Editor/Debugger. Use the syntax examples provided in the Mask Editor's **Icon** pane to help solve errors in the icon drawing commands.

For more information on debugging Masks, see:

- "Debugging Masks Using the Mask Editor" on page 13-51
- "Debugging Masks Using the MATLAB Editor/Debugger" on page 13-51

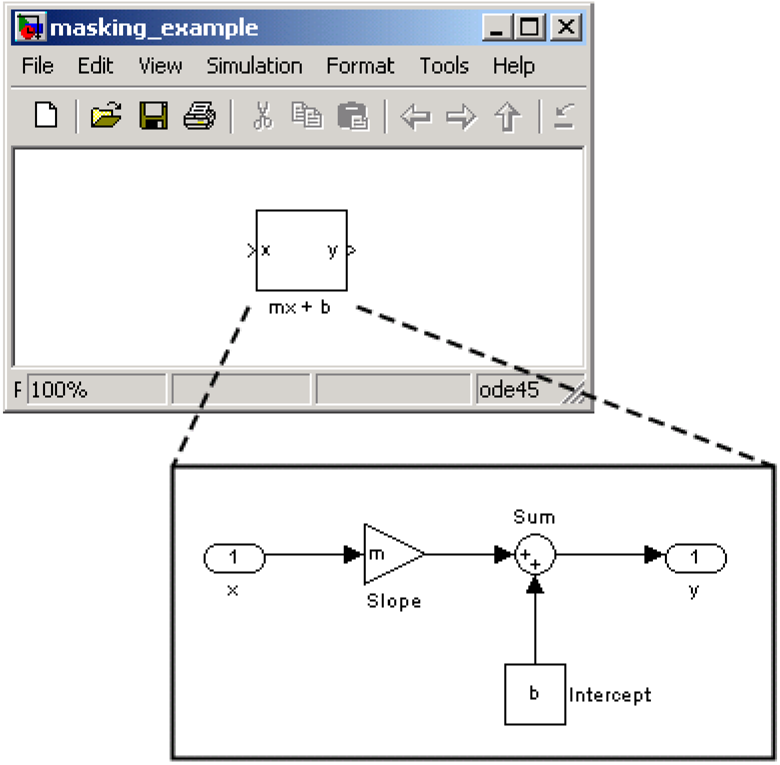
Debugging Masks Using the Mask Editor

You can debug initialization commands and parameter callbacks entered directly into the Mask Editor in these ways:

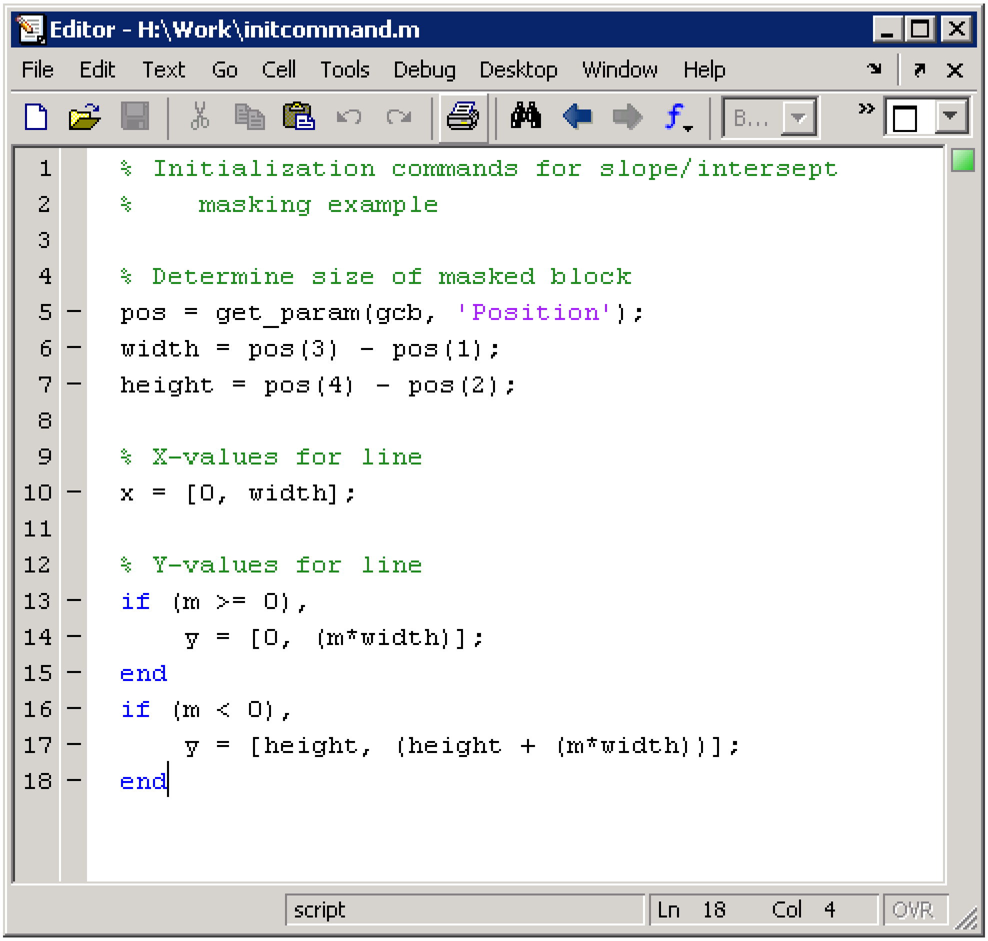
- Remove the terminating semicolon from a command to echo its results to the MATLAB Command Window.
- Place a `keyboard` command in the code to stop execution and give control to the keyboard. For more information, see the help text for the `keyboard` command.

Debugging Masks Using the MATLAB Editor/Debugger

You can debug initialization commands and parameter callbacks written in M-files using the MATLAB Editor/Debugger. For example, consider masking a subsystem that calculates the equation of a line. See "Masked Subsystem Example" on page 13-6 for information on creating this example.

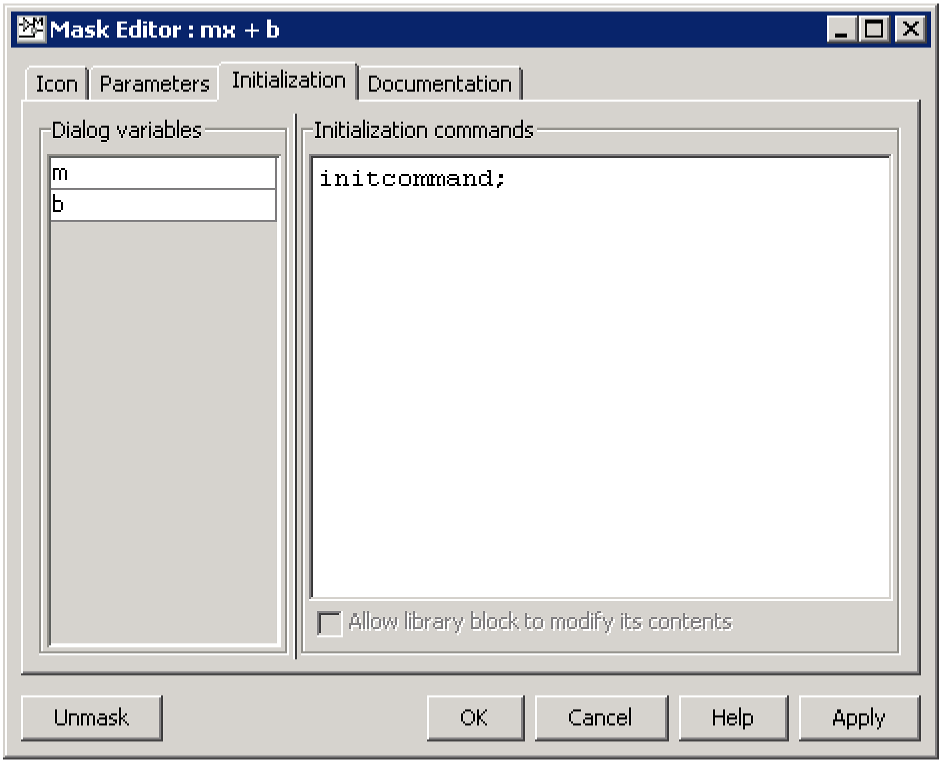


1 Write an M-file script `initcommand.m` that contains initialization commands to calculate the `x` and `y` values of the line.


Editor - H:\Work\initcommand.m
File Edit Text Go Cell Tools Debug Desktop Window Help
B...
script
Ln 18 Col 4
OVR

```
% Initialization commands for slope/intersept
%    masking example

% Determine size of masked block
pos = get_param(gcb, 'Position');
width = pos(3) - pos(1);
height = pos(4) - pos(2);

% X-values for line
x = [0, width];

% Y-values for line
if (m >= 0),
    y = [0, (m*width)];
end
if (m < 0),
    y = [height, (height + (m*width))];
end
```

2 Enter the name of the script into the **Initialization commands** field in the Mask Editor's **Initialization** pane.



m
b

```
initcommand;
```

3 Use the MATLAB Editor/Debugger to place breakpoints in the M-file and step through the code. See "Editing and Debugging M-Files" in MATLAB Desktop Tools and Development Environment for more information on editing and debugging M-files.

Note Simulink catches errors in parameter callbacks and initialization commands. To stop execution when an error occurs, you must issue the following command at the MATLAB command prompt:

```
dbstop if caught error
```

You can view the contents of the mask workspace while debugging M-file initialization commands. When debugging M-file parameter callbacks, you can access only the block's base workspace. If you need the value of a mask parameter, use the `get_param` command.

Simulink Debugger

The following sections tell you how to use the Simulink debugger to pinpoint bugs in a model:

Introduction

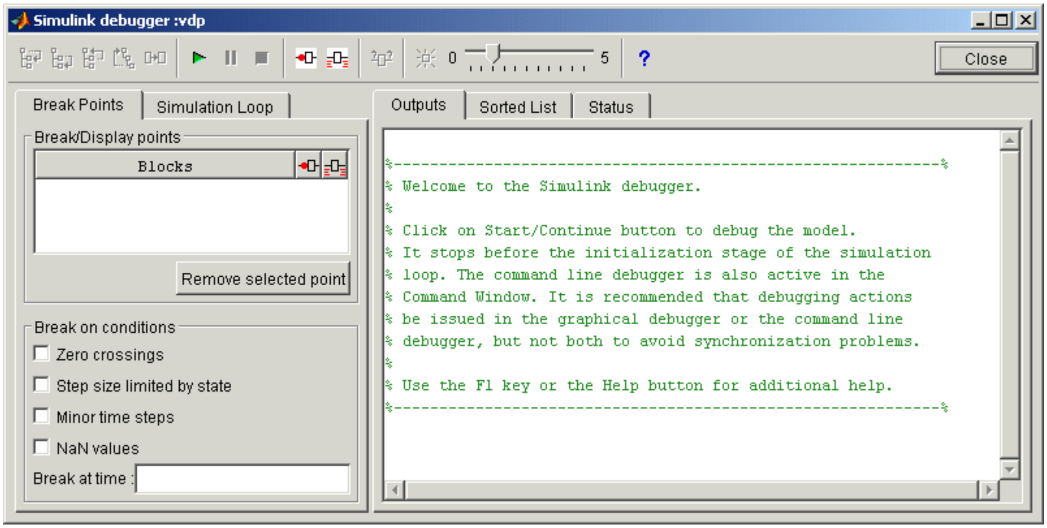
The Simulink debugger allows you to run a simulation method by method, stopping the simulation after each method, to examine the results of executing that method. This allows you to pinpoint problems in your model to specific blocks, parameters, or interconnections.

Note Methods are functions that Simulink uses to solve a model at each time step during the simulation. Blocks are made up of multiple methods. “Block execution” in this documentation is shorthand notation for “block methods execution.” Block diagram execution is a multi-step operation that requires execution of the different block methods in all the blocks in a diagram at various points during the process of solving a model at each time step during simulation, as specified by the simulation loop.

The Simulink debugger has both a graphical and a command-line user interface. The graphical interface allows you to access the debugger’s most commonly used features. The command-line interface gives you access to all the debugger’s capabilities. If both interfaces enable you to perform a task, the documentation shows you first how to use the graphical interface, then the command-line interface, to perform the task.

Using the Debugger’s Graphical User Interface

Select **Simulink Debugger** from a model window’s **Tools** menu to display the Simulink debugger’s graphical interface.



□ Zero crossings

□ Step size limited by state

□ Minor time steps

□ NaN values

Break at time :

```
%--------------------------------------------------------------%
% Welcome to the Simulink debugger.
%
% Click on Start/Continue button to debug the model.
% It stops before the initialization stage of the simulation
% loop. The command line debugger is also active in the
% Command Window. It is recommended that debugging actions
% be issued in the graphical debugger or the command line
% debugger, but not both to avoid synchronization problems.
%
% Use the F1 key or the Help button for additional help.
%--------------------------------------------------------------%
```

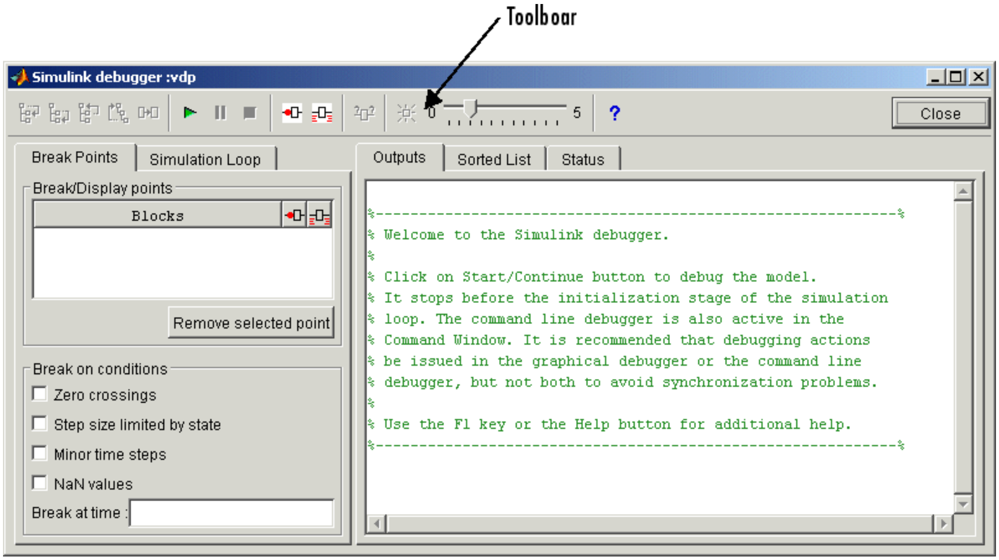
The following topics describe the major components of the debugger’s graphical user interface:

- “Toolbar” on page 14-4
- “Breakpoints Pane” on page 14-6
- “Simulation Loop Pane” on page 14-7
- “Outputs Pane” on page 14-9
- “Sorted List Pane” on page 14-10
- “Status Pane” on page 14-11

Note The debugger's graphical user interface does not display state or solver information. The debugger's command line interface does provide this information. See "Displaying System States" on page 14-37 and "Displaying Solver Information" on page 14-37.

Toolbar

The debugger toolbar appears at the top of the debugger window.


Toolbar

Simulink debugger :vdp

☐ Zero crossings

☐ Step size limited by state

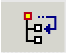
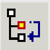
☐ Minor time steps

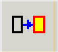
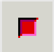
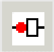
☐ NaN values

Break at time :

```
%--------------------------------------------------------------%
% Welcome to the Simulink debugger.
%
% Click on Start/Continue button to debug the model.
% It stops before the initialization stage of the simulation
% loop. The command line debugger is also active in the
% Command Window. It is recommended that debugging actions
% be issued in the graphical debugger or the command line
% debugger, but not both to avoid synchronization problems.
%
% Use the F1 key or the Help button for additional help.
%--------------------------------------------------------------%
```

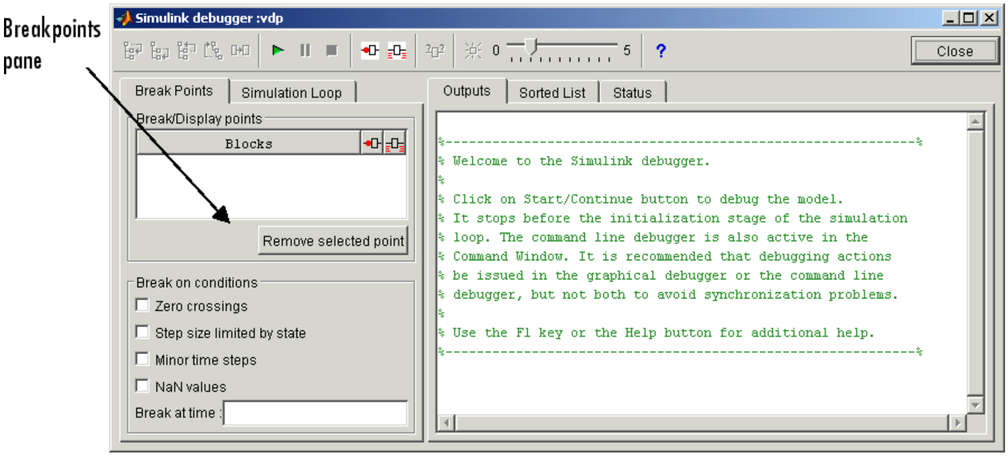
From left to right, the toolbar contains the following command buttons:

Button	Purpose
	Step into next method (see "Stepping Commands" on page 14-22 for more information on this and the following stepping commands).
	Step over next method.
	Step out of current method.

Button	Purpose
	Step to first method at start of next time step.
	Step to next block method.
	Start or continue the simulation.
	Pause the simulation.
	Stop the simulation.
	Break before the selected block.
	Display inputs and outputs of the selected block when executed (same as `trace gcb`).
	Display current inputs and outputs of selected block (same as `probe gcb`).
	Toggle animation mode on or off (see "Animation Mode" on page 14-23). The slider next to this button controls the animation rate.
?	Display help for the debugger.
Close	Close the debugger.

Breakpoints Pane

To display the **Breakpoints** pane, select the **Break Points** tab on the debugger window.



Breakpoints pane

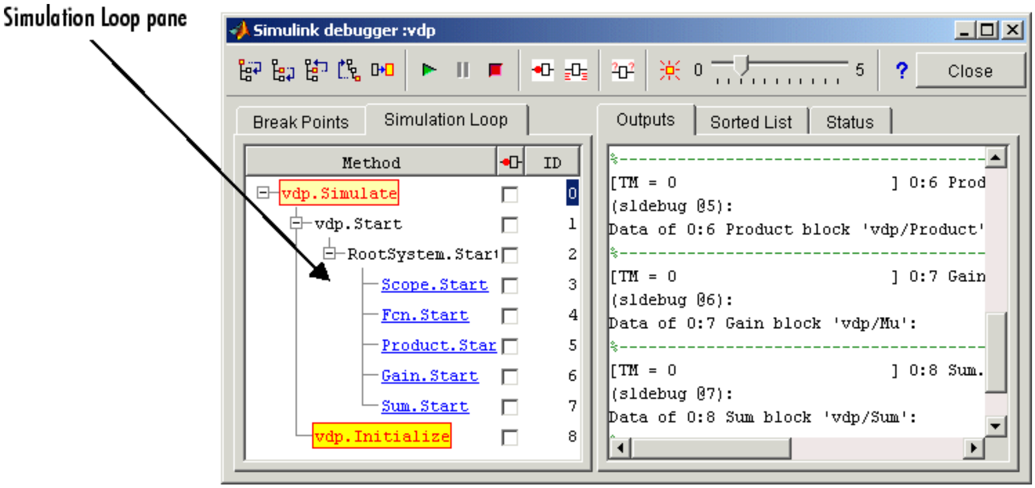
```
%----------------------------------------------------------------%
% Welcome to the Simulink debugger.
%
% Click on Start/Continue button to debug the model.
% It stops before the initialization stage of the simulation
% loop. The command line debugger is also active in the
% Command Window. It is recommended that debugging actions
% be issued in the graphical debugger or the command line
% debugger, but not both to avoid synchronization problems.
%
% Use the F1 key or the Help button for additional help.
%----------------------------------------------------------------%
```

The **Breakpoints** pane allows you to specify block methods or conditions at which to stop a simulation. See “Setting Breakpoints” on page 14-28 for more information.

Note The debugger grays out and disables the **Breakpoints** pane when its animation mode is selected (see “Animation Mode” on page 14-23). This prevents you from setting breakpoints and indicates that animation mode ignores existing breakpoints.

Simulation Loop Pane

To display the **Simulation Loop** pane, select the **Simulation Loop** tab on the debugger window.



```
Method                      ID
⊟ vdp.Simulate          □   0
  ⊟ vdp.Start           □   1
    ⊟ RootSystem.Start  □   2
        Scope.Start     □   3
        Fcn.Start       □   4
        Product.Star    □   5
        Gain.Start      □   6
        Sum.Start       □   7
    vdp.Initialize      □   8
```

```
%----------------------------------------
[TM = 0                    ] 0:6 Prod
(sldebug @5):
Data of 0:6 Product block 'vdp/Product'
%----------------------------------------
[TM = 0                    ] 0:7 Gain
(sldebug @6):
Data of 0:7 Gain block 'vdp/Mu':
%----------------------------------------
[TM = 0                    ] 0:8 Sum.
(sldebug @7):
Data of 0:8 Sum block 'vdp/Sum':
```

The Simulation Loop pane contains three columns:

- Method
- Breakpoints
- ID

Method Column

The **Method** column lists the methods that have been called thus far in the simulation as a method tree with expandable/collapsible nodes. Each node of the tree represents a method that calls other methods. Expanding a node shows the methods that the block method calls. Block method names are hyperlinks. Clicking a block method name highlights the corresponding block in the model diagram. Block method names are underlined to indicate that they are hyperlinks.

Whenever the simulation stops, the debugger highlights the name of the method where the simulation has stopped as well as the methods that directly or indirectly invoked it. The highlighted method names visually indicate the current state of the simulator's method call stack.

Breakpoints Column

The breakpoints column consists of check boxes. Selecting a check box sets a breakpoint at the method whose name appears to the left of the check box. See "Setting Breakpoints from the Simulation Loop Pane" on page 14-30 for more information.

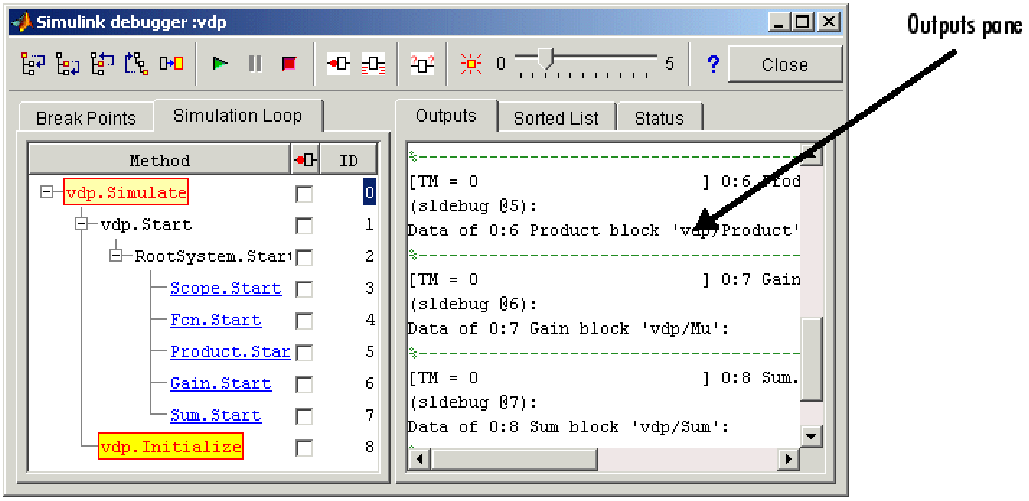
Note The debugger grays out and disables this column when its animation mode is selected (see "Animation Mode" on page 14-23). This prevents you from setting breakpoints and indicates that animation mode ignores existing breakpoints.

ID Column

The ID column lists the IDs of the methods listed in the **Methods** column. See "Method ID" on page 14-12 for more information.

Outputs Pane

To display the **Outputs** pane, select the **Outputs** tab on the debugger window.



Method		ID
vdp.Simulate	□	0
vdp.Start	□	1
RootSystem.Start	□	2
Scope.Start	□	3
Fcn.Start	□	4
Product.Star	□	5
Gain.Start	□	6
Sum.Start	□	7
vdp.Initialize	□	8

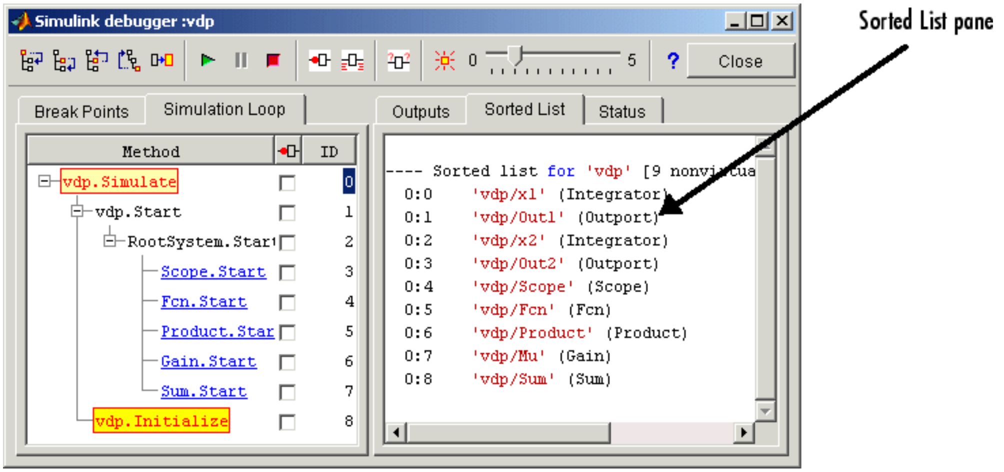
```
%----------------------------------------
[TM = 0                    ] 0:6 Prod
(sldebug @5):
Data of 0:6 Product block 'vdp/Product'
%----------------------------------------
[TM = 0                    ] 0:7 Gain
(sldebug @6):
Data of 0:7 Gain block 'vdp/Mu':
%----------------------------------------
[TM = 0                    ] 0:8 Sum.
(sldebug @7):
Data of 0:8 Sum block 'vdp/Sum':
```

Outputs pane

The Outputs pane displays the same debugger output that would appear in the MATLAB Command Window, if the debugger were running in command-line mode. The output includes the debugger command prompt and the inputs, outputs, and states of the block at whose method the simulation is currently paused (see “Block Data Output” on page 14-21). The command prompt displays current simulation time and the name and index of the method in which the debugger is currently stopped (see “Block ID” on page 14-12).

Sorted List Pane

To display the **Sorted List** pane, select the **Sorted List** tab on the debugger window.



Method	□	ID
vdp.Simulate	□	0
vdp.Start	□	1
RootSystem.Start	□	2
Scope.Start	□	3
Fcn.Start	□	4
Product.Star	□	5
Gain.Start	□	6
Sum.Start	□	7
vdp.Initialize	□	8

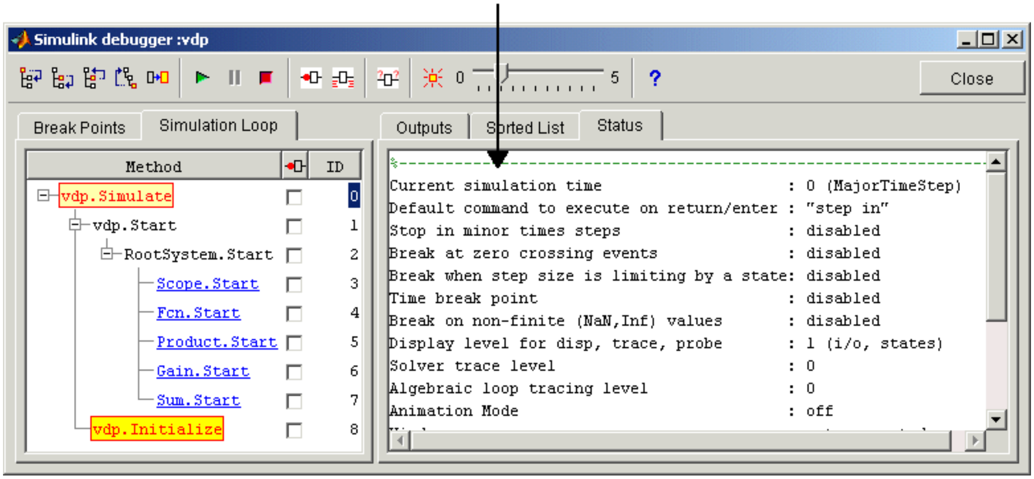
```
---- Sorted list for 'vdp' [9 nonvirtua
  0:0     'vdp/x1' (Integrator)
  0:1     'vdp/Out1' (Outport)
  0:2     'vdp/x2' (Integrator)
  0:3     'vdp/Out2' (Outport)
  0:4     'vdp/Scope' (Scope)
  0:5     'vdp/Fcn' (Fcn)
  0:6     'vdp/Product' (Product)
  0:7     'vdp/Mu' (Gain)
  0:8     'vdp/Sum' (Sum)
```

The **Sorted List** pane displays the sorted lists for the model being debugged. See “Displaying a Model’s Sorted Lists” on page 14-38 for more information.

Status Pane

To display the **Status** pane, select the **Status** tab on the debugger window.

Status pane



Method		ID
vdp.Simulate	☐	0
vdp.Start	☐	1
RootSystem.Start	☐	2
Scope.Start	☐	3
Fcn.Start	☐	4
Product.Start	☐	5
Gain.Start	☐	6
Sum.Start	☐	7
vdp.Initialize	☐	8

```
%----------------------------------------------------------------------
Current simulation time                      : 0 (MajorTimeStep)
Default command to execute on return/enter : "step in"
Stop in minor times steps                    : disabled
Break at zero crossing events                : disabled
Break when step size is limiting by a state: disabled
Time break point                             : disabled
Break on non-finite (NaN,Inf) values         : disabled
Display level for disp, trace, probe         : 1 (i/o, states)
Solver trace level                           : 0
Algebraic loop tracing level                 : 0
Animation Mode                               : off
```

The **Status** pane displays the values of various debugger options and other status information.

Using the Debugger's Command-Line Interface

In command-line mode, you control the debugger by entering commands at the debugger command line in the MATLAB Command Window. The debugger accepts abbreviations for debugger commands. See "Simulink Debugger Commands — Alphabetical List" for a list of command abbreviations and repeatable commands. You can repeat some commands by entering an empty command (i.e., by pressing the **Enter** key) at the MATLAB command line.

For more information, see:

- "Method ID" on page 14-12
- "Block ID" on page 14-12
- "Accessing the MATLAB Workspace" on page 14-12

Method ID

Some Simulink commands and messages use method IDs to refer to methods. A method ID is an integer assigned to a method the first time it is invoked in a simulation. The debugger assigns method indexes sequentially, starting with 0 for the first method invoked in a debugger session.

Block ID

Some Simulink debugger commands and messages use block IDs to refer to blocks. Simulink assigns block IDs to blocks while generating the model's sorted lists during the compilation phase of the simulation. A block ID has the form `sid:bid` where `sid` is an integer identifying the system that contains the block (either the root system or a nonvirtual subsystem) and `bid` is the position of the block in the system's sorted list. For example, the block index `0:1` refers to the first block in the model's root system. The `slist` command shows the block ID for each block in the model being debugged.

Accessing the MATLAB Workspace

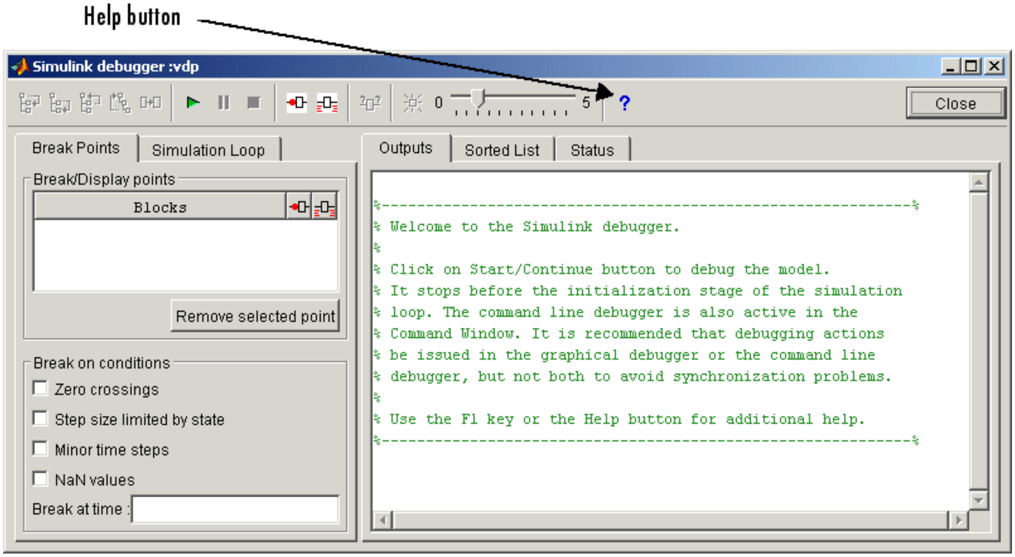
You can enter any MATLAB expression at the `sldebug` prompt. For example, suppose you are at a breakpoint and you are logging time and output of your model as `tout` and `yout`. Then the following command

```
(sldebug ...) plot(tout, yout)
```

creates a plot. You cannot display the value of a workspace variable whose name is partially or entirely the same as that of a debugger command by entering it at the debugger command prompt. You can, however, use the MATLAB `eval` command to work around this problem. For example, use `eval('s')` to determine the value of `s` rather then `s(tep)` the simulation.

Getting Online Help

You can get online help on using the debugger by clicking the **Help** button on the debugger's toolbar. Clicking the **Help** button displays help for the debugger in the MATLAB Help browser.



```
%----------------------------------------------------------------%
% Welcome to the Simulink debugger.
%
% Click on Start/Continue button to debug the model.
% It stops before the initialization stage of the simulation
% loop. The command line debugger is also active in the
% Command Window. It is recommended that debugging actions
% be issued in the graphical debugger or the command line
% debugger, but not both to avoid synchronization problems.
%
% Use the F1 key or the Help button for additional help.
%----------------------------------------------------------------%
```

In command-line mode, you can get a brief description of the debugger commands by typing `help` at the debug prompt.

Starting the Debugger

You can start the debugger either from a model window or from the MATLAB command line. To start the debugger from a model window, select **Simulink Debugger** from the model window's **Tools** menu. The debugger's graphical user interface appears (see "Using the Debugger's Graphical User Interface" on page 14-3).

To start the debugger from the MATLAB Command Window, enter either a `sim` command or the `sldebug` command. For example, either the command

```
sim('vdp',[0,10],simset('debug','on'))
```

or the command

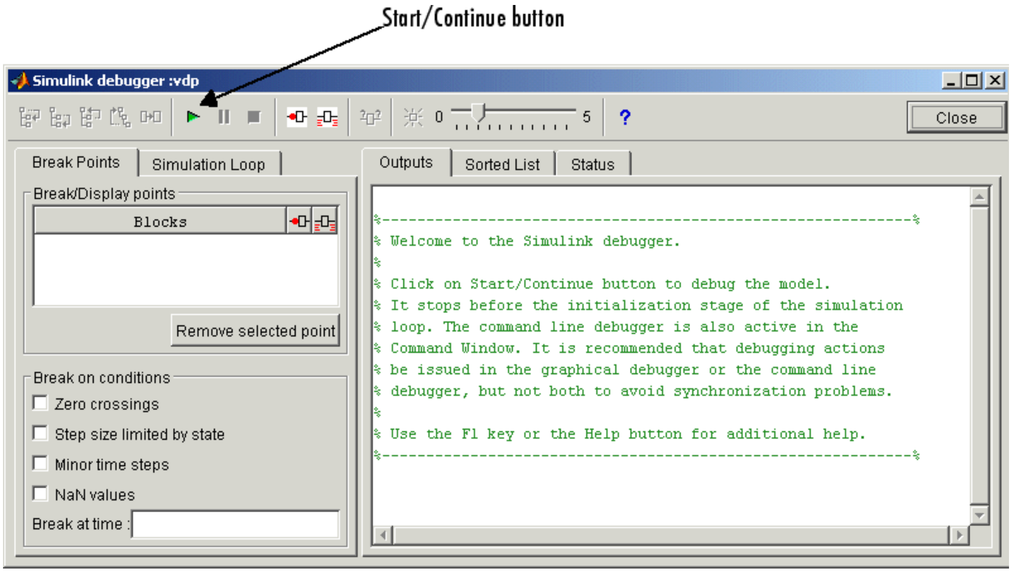
```
sldebug 'vdp'
```

loads the Simulink demo model `vdp` into memory, starts the simulation, and stops the simulation at the first block in the model's execution list.

Note When running the debugger in graphical user interface (GUI) mode, you must explicitly start the simulation. See "Starting a Simulation" on page 14-16 for more information.

Starting a Simulation

To start the simulation, click the **Start/Continue** button on the debugger's toolbar.

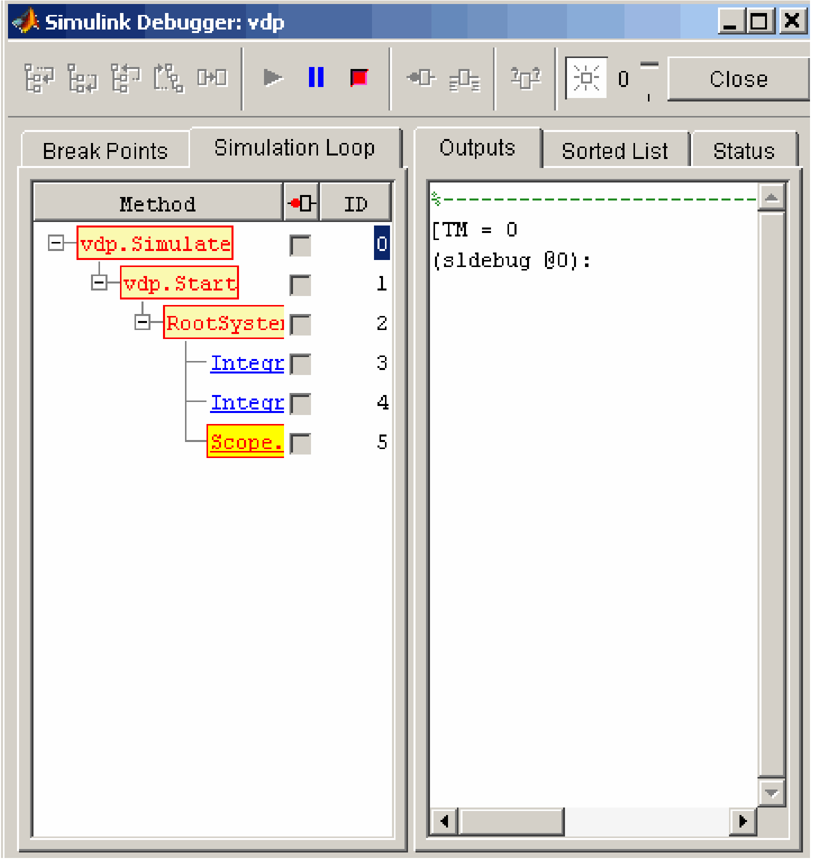


Start/Continue button

```
%-------------------------------------------------------------%
% Welcome to the Simulink debugger.
%
% Click on Start/Continue button to debug the model.
% It stops before the initialization stage of the simulation
% loop. The command line debugger is also active in the
% Command Window. It is recommended that debugging actions
% be issued in the graphical debugger or the command line
% debugger, but not both to avoid synchronization problems.
%
% Use the F1 key or the Help button for additional help.
%-------------------------------------------------------------%
```

The simulation starts and stops at the first simulation method to be executed. It displays the name of the method in its **Simulation Loop** pane and in the debug pointer on the Simulink block diagram. The debug pointer indicates on the Simulink block diagram which block method is being executed at each step. At this point, you can set breakpoints, run the simulation step by step, continue the simulation to the next breakpoint or end, examine data, or perform other debugging tasks. As the simulation progresses, the Simulink block diagram updates with debug pointers.

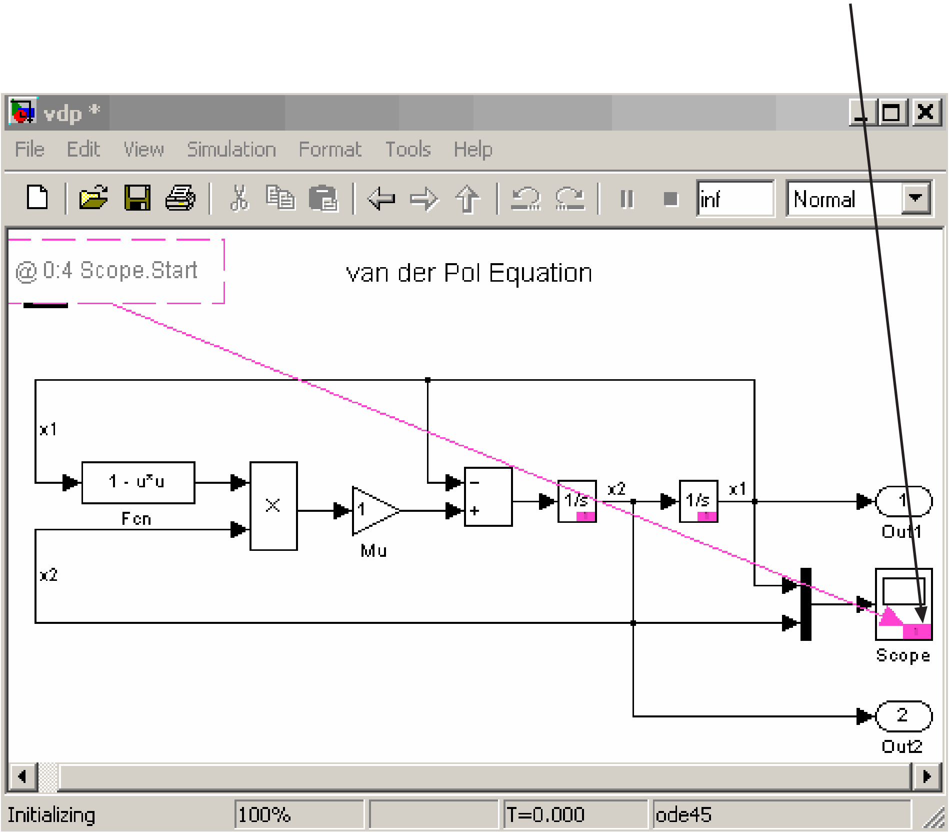
The debugger displays the name of the method in its Simulation Loop pane, as shown in the following figure:



Method		ID
vdp.Simulate	□	0
vdp.Start	□	1
RootSyste	□	2
Integr	□	3
Integr	□	4
Scope.	□	5

```
%-----------------------------
[TM = 0
(sldebug @0):
```

The debugger also displays a graphical debug pointer (see “Debug Pointer” on page 14-25) in the block diagram of the model being debugged. The debug pointer points to the first block method to be executed.



debug pointer

vdp *

@ 0:4 Scope.Start

inf

Normal

Initializing | 100% | T=0.000 | ode45

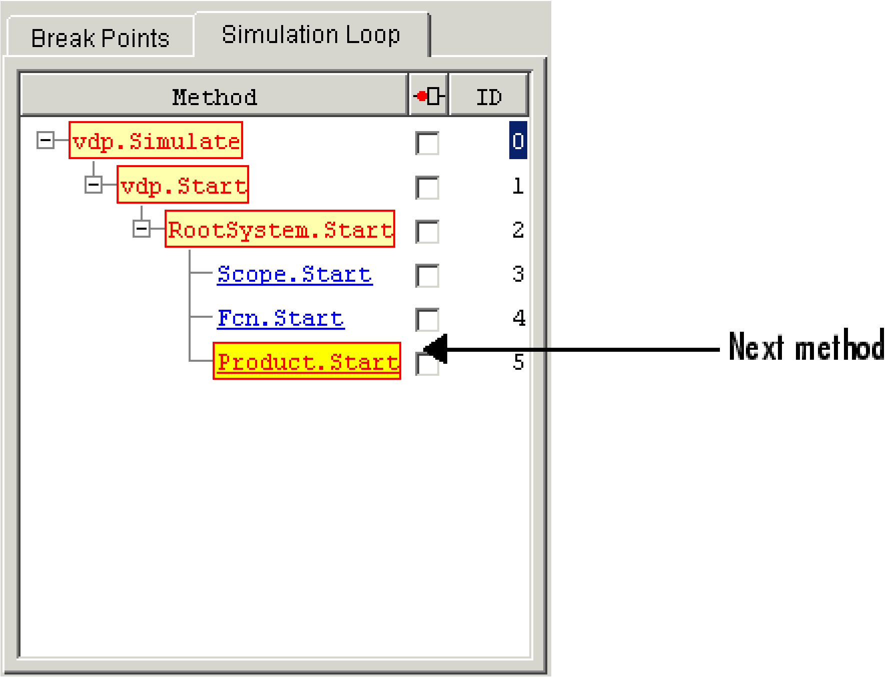
The following sections explain how to use the debugger’s graphical controls to perform these debugging tasks.

Note When you start the debugger in GUI mode, the debugger's command-line interface is also active in the MATLAB Command Window. However, you should avoid using the command-line interface, to prevent synchronization errors between the graphical and command-line interfaces.

Running a Simulation Step by Step

The Simulink debugger provides various commands that let you advance a simulation from the method where it is currently suspended (the next method) by various increments (see “Stepping Commands” on page 14-22). For example, you can advance the simulation into or over the next method, or out of the current method, or to the top of the simulation loop. After each advance, the debugger displays information that enables you to determine the point to which the simulation has advanced and the results of advancing the simulation to that point.

For example, in GUI mode, after each step command, the debugger highlights the current method call stack in the **Simulation Loop** pane. The call stack comprises the next method and the methods that invoked the next method either directly or indirectly. The debugger highlights the call stack by highlighting the names of the methods that make up the call stack in the **Simulation Loop** pane.



Method		ID
vdp.Simulate	☐	0
vdp.Start	☐	1
RootSystem.Start	☐	2
Scope.Start	☐	3
Fcn.Start	☐	4
Product.Start	☐	5

Next method

In command-line mode, you can use the `where` command to display the method call stack. If the next method is a block method, the debugger points the debug pointer at the block corresponding to the method (see "Debug Pointer" on page 14-25 for more information). If the block of the next method to be executed resides in a subsystem, the debugger opens the subsystem and points to the block in the subsystem's block diagram.

The following topics provide more information:

- "Block Data Output" on page 14-21
- "Stepping Commands" on page 14-22
- "Continuing a Simulation" on page 14-23
- "Running a Simulation Nonstop" on page 14-24
- "Debug Pointer" on page 14-25

Block Data Output

After executing a block method, the debugger prints any or all of the following block data in the debugger Output panel (in GUI mode) or in the MATLAB Command Window (in command-line mode):

- `Un = v`

  where `v` is the current value of the block's `n`th input.

- `Yn = v`

  where `v` is the current value of the block's `n`th output.

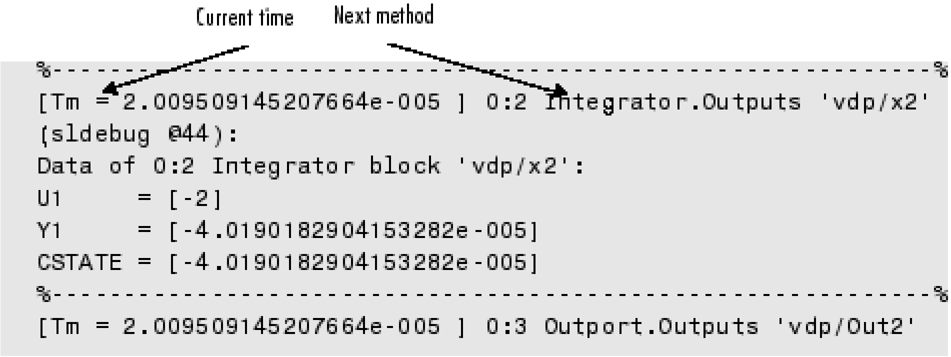
- `CSTATE = v`

  where `v` is the value of the block's continuous state vector.

- `DSTATE = v`

  where `v` is the value of the blocks discrete state vector.

The debugger also displays the current time, the ID and name of the next method to be executed, and the name of the block to which the method applies in the MATLAB Command Window. The following example illustrates typical debugger outputs after a step command.



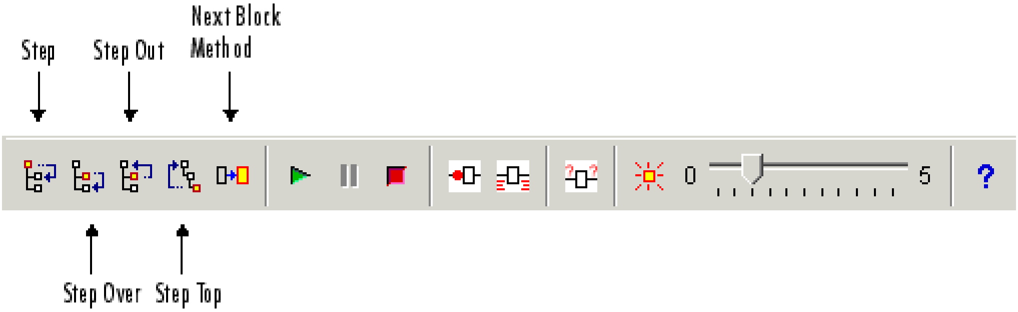
```
%----------------------------------------------------------------%
[Tm = 2.009509145207664e-005 ] 0:2 Integrator.Outputs 'vdp/x2'
(sldebug @44):
Data of 0:2 Integrator block 'vdp/x2':
U1     = [-2]
Y1     = [-4.0190182904153282e-005]
CSTATE = [-4.0190182904153282e-005]
%----------------------------------------------------------------%
[Tm = 2.009509145207664e-005 ] 0:3 Outport.Outputs 'vdp/Out2'
```

Stepping Commands

Command-line mode provides the following commands for advancing a simulation incrementally:

Command	Advances the simulation...
`step [in into]`	Into the next method, stopping at the first method in the next method or, if the next method does not contain any methods, at the end of the next method
`step over`	To the method that follows the next method, executing all methods invoked directly or indirectly by the next method
`step out`	To the end of the current method, executing any remaining methods invoked by the current method
`step top`	To the first method of the next time step (i.e., the top of the simulation loop)
`step blockmth`	To the next block method to be executed, executing all intervening model- and system-level methods
`next`	Same as `step over`

Buttons in the debugger toolbar allow you to access these commands in GUI mode.



Clicking a button has the same effect as entering the corresponding command at the debugger command line.

Continuing a Simulation

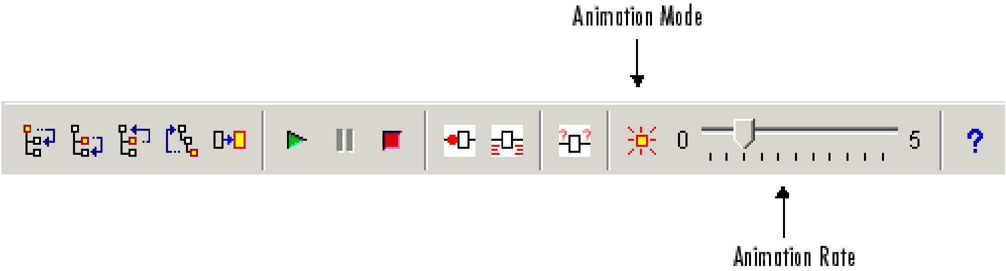
In GUI mode, the **Stop** button turns red when the debugger suspends the simulation for any reason. To continue the simulation, click the **Start/Continue** button. In command-line mode, enter `continue` to continue the simulation. By default, the debugger runs the simulation to the next breakpoint (see "Setting Breakpoints" on page 14-28) or to the end of the simulation, whichever comes first.

Animation Mode

In *animation mode*, the **Start/Continue** button or the `continue` command advances the simulation method by method, pausing after each method, to the first method of the next major time step. While running the simulation in animation mode, the debugger uses its debug pointer (see "Debug Pointer" on page 14-25) to indicate on the block diagram which block method is being executed at each step. The moving pointer providing a visual indication of the progress of the simulation.

Note When animation mode is enabled, the debugger does not allow you to set breakpoints and ignores any breakpoints that you set when animating the simulation.

To enable animation when running the debugger in GUI mode, click the **Animation Mode** toggle button on the debugger's toolbar.



The slider on the debugger toolbar allows you to increase or decrease the delay between method invocations and hence to slow down or speed up the animation rate. To disable animation mode when running the debugger in GUI mode, toggle the **Animation Mode** button on the toolbar.

To enable animation when running the debugger in command-line mode, enter the `animate` command at the MATLAB command line. The `animate` command's optional delay parameter allows you to specify the length of the pause between method invocations (1 second by default) and thereby accelerate or slow down the animation. For example, the command

```
animate 0.5
```

causes the animation to run at twice its default rate. To disable animation mode when running the debugger in command-line mode, enter

```
animate stop
```

at the MATLAB command line.

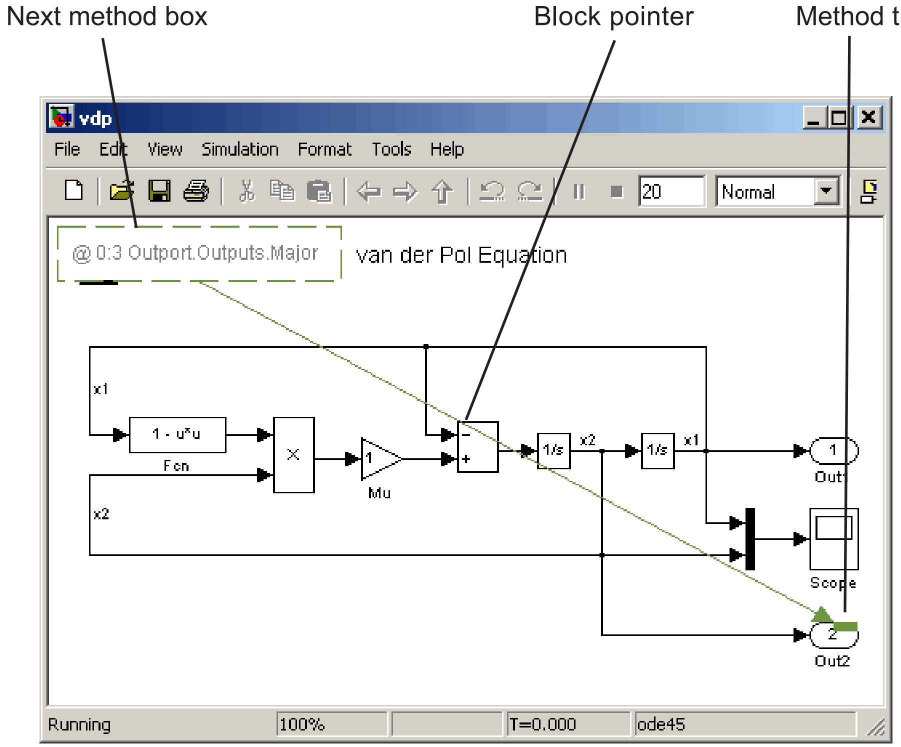
Running a Simulation Nonstop

The `run` command lets you run a simulation to the end of the simulation, skipping any intervening breakpoints. At the end of the simulation, the debugger returns you to the MATLAB command line. To continue debugging a model, you must restart the debugger.

Note The GUI mode does not provide a graphical version of the `run` command. To run the simulation to the end, you must first clear all breakpoints and then click the **Start/Continue** button.

Debug Pointer

Whenever the debugger stops the simulation at a method, it displays a debug pointer on the block diagram of the model being debugged.



The debug pointer is an annotation that indicates the next method to be executed when simulation resumes. It consists of the following elements:

- Next method box

- Block pointer
- Method tile

Next Method Box

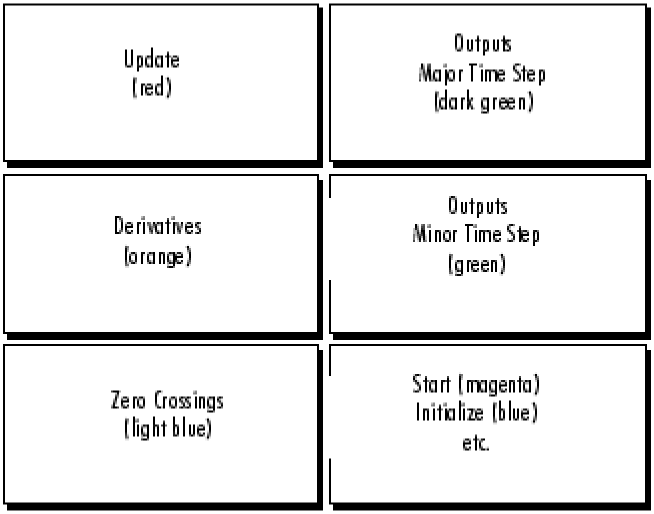
The next method box appears in the upper-left corner of the block diagram. It specifies the name and ID of the next method to be executed.

Block Pointer

The block pointer appears when the next method is a block method. It indicates the block on which the next method operates.

Method Tile

The method tile is a rectangular patch of color that appears when the next method is a block method. The tile overlays a portion of the block on which the next method executes. The color and position of the tile on the block indicate the type of the next block method as follows.



In animation mode, the tiles persist for the length of the current major time step and a number appears in each tile. The number specifies the number of times that the corresponding method has been invoked for the block thus far in the time step.

Setting Breakpoints

The Simulink debugger allows you to define stopping points in a simulation called breakpoints. You can then run a simulation from breakpoint to breakpoint, using the debugger's `continue` command. The debugger lets you define two types of breakpoints: unconditional and conditional. An unconditional breakpoint occurs whenever a simulation reaches a method that you specified previously. A conditional breakpoint occurs when a condition that you specified in advance arises in the simulation.

Breakpoints are useful when you know that a problem occurs at a certain point in your program or when a certain condition occurs. By defining an appropriate breakpoint and running the simulation via the `continue` command, you can skip immediately to the point in the simulation where the problem occurs.

For more information on setting breakpoints, see:

- "Setting Unconditional Breakpoints" on page 14-28
- "Setting Conditional Breakpoints" on page 14-31

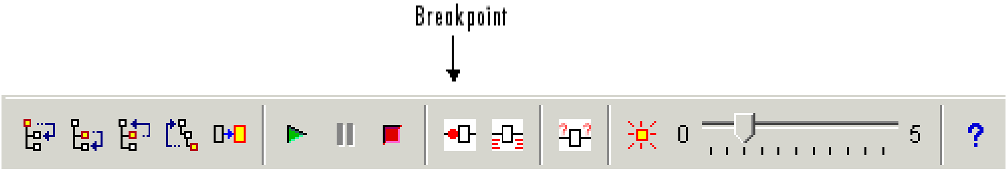
Setting Unconditional Breakpoints

You can set unconditional breakpoints from the

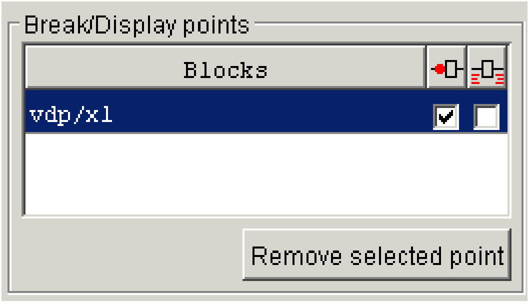
- Debugger toolbar
- **Simulation Loop** pane
- MATLAB Command Window (command-line mode only)

Setting Breakpoints from the Debugger Toolbar

To set a breakpoint on a block's methods, select the block and then click the **Breakpoint** button on the debugger toolbar. If you set a break point on a block, the debugger stops if the execution reaches any method of the block.



The debugger displays the name of the selected block in the **Break/Display points** panel of its **Breakpoints** pane.



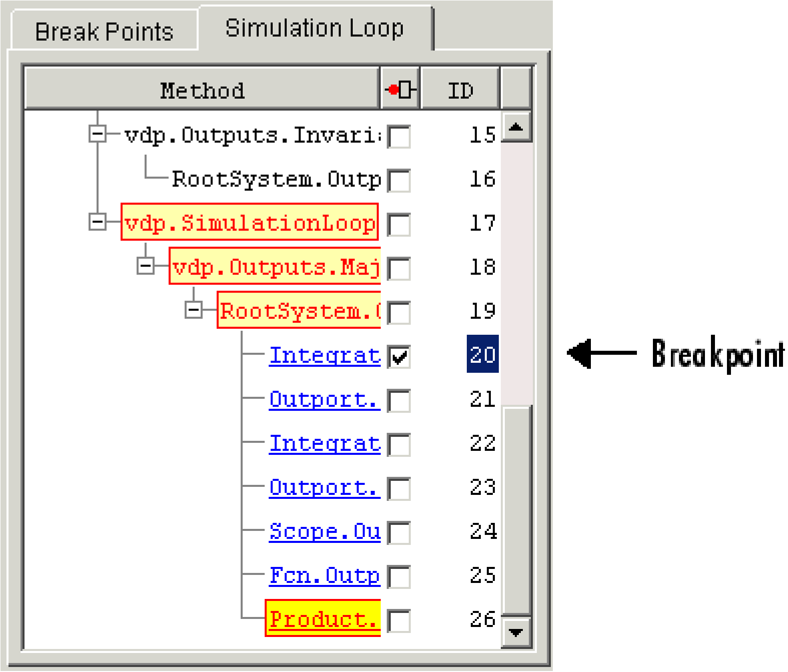
Note Clicking the **Breakpoint** button on the toolbar sets breakpoints on the invocations of a block's methods in major time steps.

You can temporarily disable the breakpoints on a block by deselecting the check box in the breakpoints column of the panel. To clear the breakpoints on a block and remove its entry from the panel, select the entry and then click the **Remove selected point** button on the panel.

Note You cannot set a breakpoint on a virtual block. A virtual block is a block whose function is purely graphical: it indicates a grouping or relationship among a model's computational blocks. The debugger warns you if you attempt to set a breakpoint on a virtual block. You can obtain a listing of a model's nonvirtual blocks, using the `slist` command (see "Displaying a Model's Nonvirtual Blocks" on page 14-40).

Setting Breakpoints from the Simulation Loop Pane

To set a breakpoint at a particular invocation of a method displayed in the Simulation Loop pane, select the check box next to the method's name in the breakpoint column of the pane.



Method		ID
vdp.Outputs.Invari:	□	15
RootSystem.Outp	□	16
vdp.SimulationLoop	□	17
vdp.Outputs.Maj	□	18
RootSystem.(	□	19
Integrat	☑	20
Outport.	□	21
Integrat	□	22
Outport.	□	23
Scope.Ou	□	24
Fcn.Outp	□	25
Product.	□	26

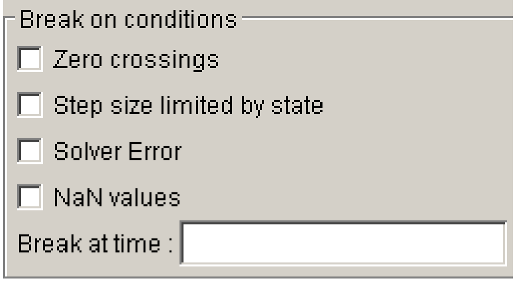
To clear the breakpoint, deselect the check box.

Setting Breakpoints from the MATLAB Command Window

In command-line mode, use the `break` and `bafter` commands to set breakpoints before or after a specified method, respectively. Use the `clear` command to clear breakpoints.

Setting Conditional Breakpoints

You can use either the **Break on conditions** controls group on the debugger's **Breakpoints** pane



Break on conditions

□ Zero crossings

□ Step size limited by state

□ Solver Error

□ NaN values

Break at time :

or the following commands (in command-line mode) to set conditional breakpoints.

Command	Causes Simulation to Stop
`tbreak [t]`	At a simulation time step
`ebreak`	At a recoverable error in the model
`nanbreak`	At the occurrence of an underflow or overflow (`NaN`) or infinite (`Inf`) value
`xbreak`	When the simulation reaches the state that determines the simulation step size
`zcbreak`	When a zero crossing occurs between simulation time steps

Setting Breakpoints at Time Steps

To set a breakpoint at a time step, enter a time in the debugger's **Break at time** field (GUI mode) or enter the time using the `tbreak` command. This causes the debugger to stop the simulation at the Outputs.Major method of the model at the first time step that follows the specified time. For example, starting `vdp` in debug mode and entering the commands

```
tbreak 2
continue
```

causes the debugger to halt the simulation at the `vdp.Outputs.Major` method of time step `2.078` as indicated by the output of the `continue` command.

```
%----------------------------------------------------------------
%
[TM = 2.078784598291364      ] vdp.Outputs.Major
(sldebug @18):
```

Breaking on Nonfinite Values

Selecting the debugger's **NaN values** option or entering the `nanbreak` command causes the simulation to stop when a computed value is infinite or outside the range of values that can be represented by the machine running the simulation. This option is useful for pinpointing computational errors in a Simulink model.

Breaking on Step-Size Limiting Steps

Selecting the **Step size limited by state** option or entering the `xbreak` command causes the debugger to stop the simulation when the model uses a variable-step solver and the solver encounters a state that limits the size of the steps that it can take. This command is useful in debugging models that appear to require an excessive number of simulation time steps to solve.

Breaking at Zero Crossings

Selecting the **Zero crossings** option or entering the `zcbreak` command causes the simulation to halt when Simulink detects a nonsampled zero crossing in a model that includes blocks where zero crossings can arise. After halting, Simulink displays the ID, type, and name of the block in which Simulink detected the zero crossing. The block ID (`s:b:p`) consists of a system index `s`, block index `b`, and port index `p` separated by colons (see "Block ID" on page 14-12).

For example, setting a zero-crossing break at the start of execution of the `zeroxing` demo model,

```
>> sldebug zeroxing
%--------------------------------------------------------------
%
[TM = 0                      ] zeroxing.Simulate
(sldebug @0): >> zcbreak
Break at zero crossing events              : enabled
```

and continuing the simulation

```
(sldebug @0): >> continue
```

results in a zero-crossing break at

```
2 Zero crossings detected at the following locations
  6  0:5:1  Saturate  'zeroxing/Saturation'
  7  0:5:2  Saturate  'zeroxing/Saturation'
ZeroCrossing Events detected. Interrupting model execution
%----------------------------------------------------------------%
[Tm = 0.4                    ] zeroxing.zc.SearchLoop
(sldebug @55): >>
```

If a model does not include blocks capable of producing nonsampled zero crossings, the command prints a message advising you of this fact.

Breaking on Solver Errors

Selecting the debugger's **Solver Errors** option or entering the `ebreak` command causes the simulation to stop if the solver detects a recoverable error in the model. If you do not set or disable this breakpoint, the solver recovers from the error and proceeds with the simulation without notifying you.

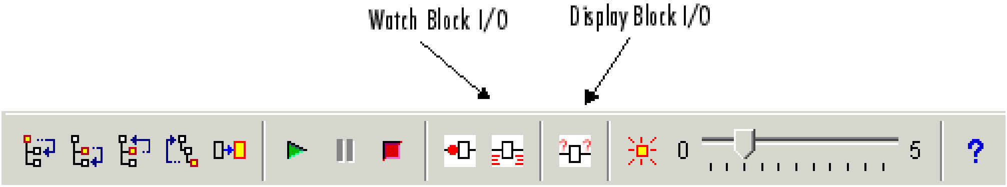
Displaying Information About the Simulation

The Simulink debugger provides a set of commands that allow you to display block states, block inputs and outputs, and other information while running a model. For more information, see:

- "Displaying Block I/O" on page 14-34
- "Displaying Algebraic Loop Information" on page 14-36
- "Displaying System States" on page 14-37
- "Displaying Solver Information" on page 14-37

Displaying Block I/O

The debugger allows you to display block I/O by clicking the appropriate buttons on the debugger toolbar



or by entering the appropriate debugger command.

Command	Displays a Block's I/O
`probe`	Immediately
`disp`	At every breakpoint any time execution stops
`trace`	Whenever the block executes

Note The two debugger toolbar buttons, Watch Block I/O () and Display Block I/O () correspond, respectively, to `trace gcb` and `probe gcb`. The `probe` and `disp` commands do not have a one to one correspondence with the debugger toolbar buttons.

Displaying I/O of Selected Block

To display the I/O of a block, select the block and click ?□? in GUI mode or enter the `probe` command in command-line mode. In the following table, the `probe gcb` command has a corresponding toolbar button. The other commands do not.

Command	Description
`probe`	Enter or exit `probe` mode. In `probe` mode, the debugger displays the current inputs and outputs of any block that you select in the model's block diagram. Typing any command causes the debugger to exit `probe` mode.
`probe gcb`	Display I/O of selected block. Same as ?□?.
`probe s:b`	Print the I/O of the block specified by system number `s` and block number `b`.

The debugger prints the current inputs, outputs, and states of the selected block in the debugger **Outputs** pane (GUI mode) or the MATLAB Command Window.

The `probe` command is useful when you need to examine the I/O of a block whose I/O is not otherwise displayed. For example, suppose you are using the `step` command to run a model method by method. Each time you step the simulation, the debugger displays the inputs and outputs of the current block. The `probe` command lets you examine the I/O of other blocks as well.

Displaying Block I/O Automatically at Breakpoints

The `disp` command causes the debugger to display a specified block's inputs and outputs whenever it halts the simulation. You can specify a block either by entering its block index or by selecting it in the block diagram and entering `gcb` as the `disp` command argument. You can remove any block from the debugger's list of display points, using the `undisp` command. For example, to remove `block 0:0`, either select the block in the model diagram and enter `undisp gcb` or simply enter `undisp 0:0`.

Note Automatic display of block I/O at breakpoints is not available in the debugger's GUI mode.

The `disp` command is useful when you need to monitor the I/O of a specific block or set of blocks as you step through a simulation. Using the `disp` command, you can specify the blocks you want to monitor and the debugger will then redisplay the I/O of those blocks on every step. Note that the debugger always displays the I/O of the current block when you step through a model block by block, using the `step` command. You do not need to use the `disp` command if you are interested in watching only the I/O of the current block.

Watching Block I/O

To watch a block, select the block and click in the debugger toolbar or enter the `trace` command. In GUI mode, if a breakpoint exists on the block, you can set a watch on it as well by selecting the check box for the block in the watch column of the **Break/Display points** pane. In command-line mode, you can also specify the block by specifying its block index in the `trace` command. You can remove a block from the debugger's list of trace points using the `untrace` command.

The debugger displays a watched block's I/O whenever the block executes. Watching a block allows you obtain a complete record of the block's I/O without having to stop the simulation.

Displaying Algebraic Loop Information

The `atrace` command causes the debugger to display information about a model's algebraic loops (see "Algebraic Loops" on page 1-26) each time they are solved. The command takes a single argument that specifies the amount of information to display.

Command	Displays for Each Algebraic Loop
`atrace 0`	No information
`atrace 1`	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error

Command	Displays for Each Algebraic Loop
atrace 2	Same as level 1
atrace 3	Level 2 plus the Jacobian matrix used to solve the loop
atrace 4	Level 3 plus intermediate solutions of the loop variable

Displaying System States

The states debug command lists the current values of the system's states in the MATLAB Command Window. For example, the following sequence of commands shows the states of the Simulink bouncing ball demo (bounce) after its first and second time steps.

```
sldebug bounce
[Tm=0                      ] **Start** of system 'bounce' outputs
(sldebug @0:0 'bounce/Position'): states
Continuous state vector (value,index,name):
  10                         0 (0:0 'bounce/Position')
  15                         1 (0:5 'bounce/Velocity')
(sldebug @0:0 'bounce/Position'): next
[Tm=0.01                   ] **Start** of system 'bounce' outputs
(sldebug @0:0 'bounce/Position'): states
Continuous state vector (value,index,name):
  10.1495095                 0 (0:0 'bounce/Position')
  14.9019                    1 (0:5 'bounce/Velocity')
```

Displaying Solver Information

The strace command allows you to pinpoint problems in solving a model's differential equations that can slow down simulation performance. Executing this command causes the debugger to display solver-related information at the MATLAB command line when you run or step through a simulation. The information includes the sizes of the steps taken by the solver, the estimated integration error resulting from the step size, whether a step size succeeded (i.e., met the accuracy requirements that the model specifies), the times at which solver resets occur, etc. If you are concerned about the time required to simulate your model, this information can help you to decide whether the solver you have chosen is the culprit and hence whether choosing another solver might shorten the time required to solve the model.

Displaying Information About the Model

In addition to providing information about a simulation, the debugger can provide you with information about the model that underlies the simulation:

- "Displaying a Model's Sorted Lists" on page 14-38
- "Displaying a Block" on page 14-39

Displaying a Model's Sorted Lists

In GUI mode, the debugger's **Sorted List** pane displays lists of blocks for a model's root system and each nonvirtual subsystem. Each list lists the blocks that the subsystems contains sorted according to their computational dependencies, alphabetical order, and other block sorting rules. In command-line mode, you can use the `slist` command to display a model's sorted lists.

```
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
  0:0    'vdp/Integrator1' (Integrator)
  0:1    'vdp/Out1' (Outport)
  0:2    'vdp/Integrator2' (Integrator)
  0:3    'vdp/Out2' (Outport)
  0:4    'vdp/Fcn' (Fcn)
  0:5    'vdp/Product' (Product)
  0:6    'vdp/Mu' (Gain)
  0:7    'vdp/Scope' (Scope)
  0:8    'vdp/Sum' (Sum)
```

These displays include the block index for each command. You can thus use them to determine the block IDs of the model's blocks. Some debugger commands accept block IDs as arguments.

Identifying Blocks in Algebraic Loops

If a block belongs to an algebraic list, the `slist` command displays an algebraic loop identifier in the entry for the block in the sorted list. The identifier has the form

```
algId=s#n
```

where s is the index of the subsystem containing the algebraic loop and n is the index of the algebraic loop in the subsystem. For example, the following entry for an Integrator block indicates that it participates in the first algebraic loop at the root level of the model.

```
0:1 'test/ss/I1' (Integrator, tid=0) [algId=0#1, discontinuity]
```

You can use the debugger's ashow command to highlight the blocks and lines that make up an algebraic loop. See "Displaying Algebraic Loops" on page 14-41 for more information.

Displaying a Block

To determine the block in a model's diagram that corresponds to a particular index, enter bshow s:b at the command prompt, where s:b is the block index. The bshow command opens the system containing the block (if necessary) and selects the block in the system's window.

Displaying a Model's Nonvirtual Systems

The systems command displays a list of the nonvirtual systems in the model being debugged. For example, the Simulink clutch demo (clutch) contains the following systems:

```
sldebug clutch
[Tm=0                    ] **Start** of system 'clutch' outputs
(sldebug @0:0 'clutch/Clutch Pedal'): systems
 0   'clutch'
 1   'clutch/Locked'
 2   'clutch/Unlocked'
```

Note The systems command does not list subsystems that are purely graphical in nature, that is, subsystems that the model diagram represents as Subsystem blocks but that Simulink solves as part of a parent system. In Simulink models, the root system and triggered or enabled subsystems are true systems. All other subsystems are virtual (that is, graphical) and hence do not appear in the listing produced by the systems command.

Displaying a Model's Nonvirtual Blocks

The slist command displays a list of the nonvirtual blocks in a model. The listing groups the blocks by system. For example, the following sequence of commands produces a list of the nonvirtual blocks in the Van der Pol (vdp) demo model.

```
sldebug vdp
[Tm=0                      ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): slist
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
  0:0    'vdp/Integrator1' (Integrator)
  0:1    'vdp/Out1' (Outport)
  0:2    'vdp/Integrator2' (Integrator)
  0:3    'vdp/Out2' (Outport)
  0:4    'vdp/Fcn' (Fcn)
  0:5    'vdp/Product' (Product)
  0:6    'vdp/Mu' (Gain)
  0:7    'vdp/Scope' (Scope)
  0:8    'vdp/Sum' (Sum)
```

Note The slist command does not list blocks that are purely graphical in nature, that is, blocks that indicate relationships or groupings among computational blocks.

Displaying Blocks with Potential Zero Crossings

The zclist command displays a list of blocks in which nonsampled zero crossings can occur during a simulation. For example, zclist displays the following list for the clutch sample model:

```
(sldebug @0:0 'clutch/Clutch Pedal'): zclist
  2:3    'clutch/Unlocked/Sign' (Signum)
  0:4    'clutch/Lockup Detection/Velocities Match' (HitCross)
  0:10   'clutch/Lockup Detection/Required Friction
           for Lockup/Abs' (Abs)
  0:11   'clutch/Lockup Detection/Required Friction for
           Lockup/ Relational Operator' (RelationalOperator)
```

```
    0:18   'clutch/Break Apart Detection/Abs' (Abs)
    0:20   'clutch/Break Apart Detection/Relational Operator'
             (RelationalOperator)
    0:24   'clutch/Unlocked' (SubSystem)
    0:27   'clutch/Locked' (SubSystem)
```

Displaying Algebraic Loops

The ashow command highlights a specified algebraic loop or the algebraic loop that contains a specified block. To highlight a specified algebraic loop, enter ashow s#n, where s is the index of the system (see "Identifying Blocks in Algebraic Loops" on page 14-38) that contains the loop and n is the index of the loop in the system. To display the loop that contains the currently selected block, enter ashow gcb. To show a loop that contains a specified block, enter ashow s:b, where s:b is the block's index. To clear algebraic-loop highlighting from the model diagram, enter ashow clear.

Displaying Debugger Status

In GUI mode, the debugger displays the settings of various debug options, such as conditional breakpoints, in its **Status** panel. In command-line mode, the status command displays debugger settings. For example, the following sequence of commands displays the initial debug settings for the vdp model:

```
sim('vdp',[0,10],simset('debug','on'))
[Tm=0                     ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): status
  Current simulation time: 0 (MajorTimeStep)
  Last command: ""
  Stop in minor times steps is disabled.
  Break at zero crossing events is disabled.
  Break when step size is limiting by a state is disabled.
  Break on non-finite (NaN,Inf) values is disabled.
  Display of integration information is disabled.
  Algebraic loop tracing level is at 0.
```

15

Simulink Accelerator

Simulink Accelerator is a MathWorks product that accelerates the simulation of Simulink models. Simulink Accelerator comes with the Simulink Profiler, a tool that collects, analyzes, and displays simulation performance data. These tools enable you to minimize the time needed to simulate Simulink models. You must install the Simulink Accelerator product on your system to use either tool. See the following sections for information on using these tools.

Simulink Accelerator

Simulink Accelerator speeds up the execution of Simulink models. Simulink Accelerator uses portions of Real-Time Workshop, a MathWorks product that automatically generates C code from Simulink models, and your C compiler to create an executable. Note that although Simulink Accelerator takes advantage of Real-Time Workshop technology, Real-Time Workshop is not required to run it. Also, if you do not have a C compiler installed on your Windows PC, you can use the `lcc` compiler provided by The MathWorks.

Note You must have the Simulink Accelerator product installed on your system to use Simulink Accelerator. Simulink Accelerator uses Real-Time Workshop technology to generate the code used to accelerate the model. However, the generated code is suitable only for acceleration of the model. If you want to generate code for other purposes, you must use Real-Time Workshop.

Simulink Accelerator Limitations

Simulink Accelerator has the following limitations.

- Simulink Accelerator does not support models with algebraic loops. If Simulink Accelerator detects an algebraic loop in your model, it halts the simulation and displays an error message.
- Simulink Accelerator does not support models that pass array parameters to M-file S-functions that are not numeric, logical, or character arrays, are sparse arrays, or that have more than two dimensions.
- Simulink Accelerator uses up to 32 bits to hold a signal, state or parameter value.

Word Size (Bits)	Data Type
1 to 8	`int8` `uint8`
9 to 16	`int16` `uint16`
17 to 32	`int32` `uint32`

See "Working with Data Types" on page 7-2 for information regarding Simulink data types.

How Simulink Accelerator Works

Simulink Accelerator works by creating and compiling C code that takes the place of the interpretive code that Simulink uses when in normal mode (that is, when Simulink is not in accelerator mode). Simulink Accelerator generates the C code from your Simulink model and invokes the MATLAB `mex` function to compile and dynamically link the generated code to Simulink.

Note The MATLAB `mex` function uses the `lcc` compiler by default to compile the code generated by Simulink Accelerator. An optimizing compiler, such as Microsoft Visual C/C++ 7.1, can produce faster compiled code and hence accelerate simulation still further. If you have such a compiler, you can configure the `mex` command to use it. See the online help for `mex` for more information.

Simulink Accelerator removes much of the computational overhead required by Simulink models when in normal mode. It works by replacing blocks that are designed to handle any possible configuration in Simulink with compiled versions customized to your particular model's configuration. Through this method, Simulink Accelerator is able to achieve substantial improvements in performance for larger Simulink models. The performance gains are tied to the size and complexity of your model. In general, as size and complexity grow, so do gains in performance. Typically, you can expect a 2X-to-6X gain in performance for models that use built-in Simulink blocks.

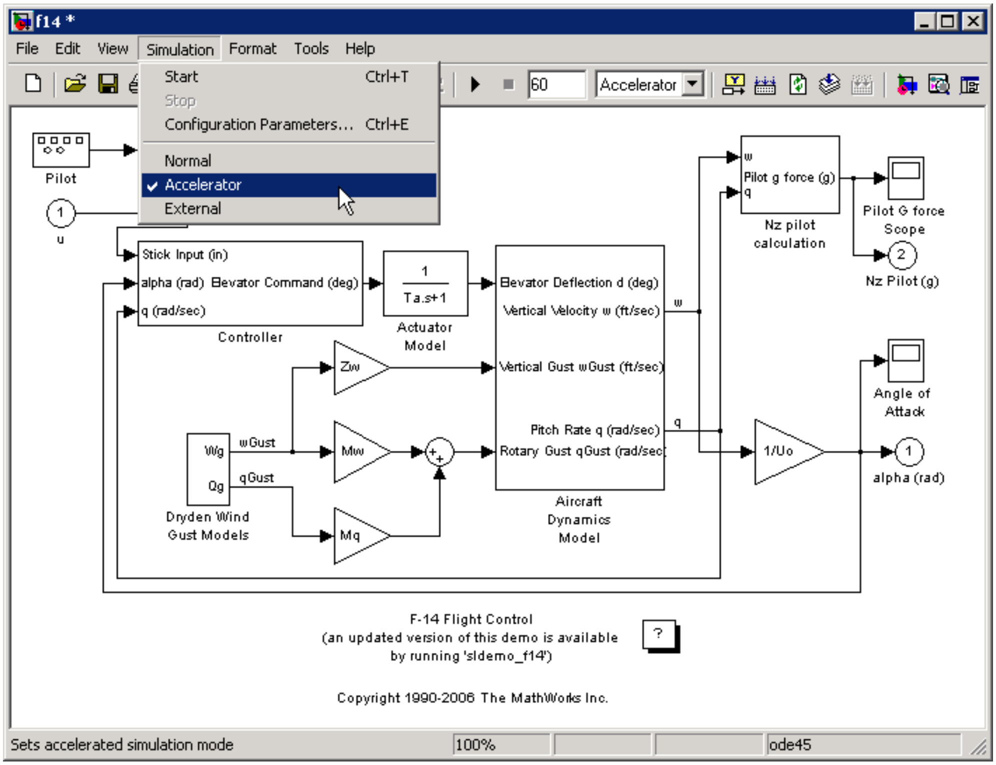
Note Blocks such as the Quantizer block might exhibit slight differences in output on some systems because of slight differences in the numerical precision of the interpreted and compiled versions of the model.

Running Simulink Accelerator

To run a model with Simulink Accelerator, first select Simulink Accelerator and then start the simulation.

1 From the **Simulation** menu, select **Accelerator**.

   Alternatively, you can select **Accelerator** from the model editor's toolbar.



2 From the **Simulation** menu, select **Start** to begin the simulation. Simulink Accelerator then generates and compiles the C code.

The information displayed during compilation is determined by the `AccelVerboseBuild` command (see "Customizing the Simulink Accelerator Build Process" on page 15-13).

Simulink Accelerator does the following after the C code is generated :

- Places the generated code in a subdirectory called `modelname_accel_rtw` (for example, `f14_accel_rtw`)
- Places a compiled MEX-file in the current working directory
- Runs the compiled model

Note If your code does not compile, the most likely reason is that you have not set up the `mex` command correctly. Run `mex -setup` at the MATLAB prompt and select your C compiler from the list shown during setup.

Handling Changes in Model Structure

After you use Simulink Accelerator to simulate a model, the MEX-file containing the compiled version of the model remains available for use in later simulations. Even if you exit MATLAB, you can reuse the MEX-file in later MATLAB or Simulink sessions.

If you alter the structure of your Simulink model, for example, by adding or deleting blocks, Simulink Accelerator automatically regenerates the C code and updates (overwrites) the existing MEX-file.

Examples of model structure changes that require Simulilnk Accelerator to rebuild include

- Changing the solver type, for example from `Variable-step` to `Fixed-step`
- Adding or deleting blocks or connections between blocks
- Changing the values of nontunable block parameters, for example, the **Initial seed** parameter of the Random Number block (see “Tunable Parameters” on page 1-9 for more information)
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states in the model
- Changing function in the Trigonometric Function block

- Changing the signs used in a Sum block
- Adding a Target Language Compiler (TLC) file to inline an S-function

Note If Simulink Accelerator regenerates the simulation target for a model at the beginning of every simulation for no apparent reason, you can determine why by capturing and comparing a checksum that Simulink uses to determine whether the code needs to be regenerated. For more information, see "Determining Why Simulink Accelerator Rebuilds" on page 15-15.

Simulink Accelerator displays a warning when you attempt any impermissible model changes during simulation. The warning does not stop the current simulation. To make the model alterations, stop the simulation, make the changes, and restart.

Some changes are permitted in the middle of simulation. Simple changes like adjusting the value of a Gain block do not cause a warning. When in doubt, try to make the change. If you do not see a warning, Simulink Accelerator accepted the change.

Note Simulink Accelerator does not display warnings that blocks generate *during* simulation. Examples include divide-by-zero and integer overflow. This is a different set of warnings from those discussed previously.

Increasing Performance of Accelerator Mode

In general, Simulink Accelerator creates code optimized for speed with most blocks available in Simulink. There are situations, however, where you can further improve performance by adjusting your simulation or being aware of Simulink Accelerator behavior. These include

- **Configuration Parameters** dialog box — To increase performance:
  - Disable **Solver data inconsistency** on the **Solver Diagnostics** pane.
  - Disable **Array bounds exceeded** on the **Data Validity Diagnostics** pane.

  - Set **Signal storage reuse** on the **Optimization** pane.

- Stateflow — Simulink Accelerator is fully compatible with Stateflow, but it does not improve the performance of the Stateflow portions of models. Disable Stateflow debugging and animation to increase performance of models that include Stateflow blocks.
- User-written S-functions — Simulink Accelerator cannot improve simulation speed for S-functions unless you inline them using the Target Language Compiler. *Inlining* refers to the process of creating TLC files that direct Real-Time Workshop to create C code for the S-function. This eliminates unnecessary calls to the Simulink application program interface (API).

  For information on how to inline S-functions, consult the Real-Time Workshop Target Language Compiler Reference Guide, which is available on the MathWorks Web site, at `www.mathworks.com`.
- S-functions supplied by Simulink and blocksets — Although Simulink Accelerator is compatible with all the blocks provided with Simulink and blocksets, it does not improve the simulation speed for M-file or C-MEX S-Function blocks that do not have an associated inlining TLC file.
- Logging large amounts of data — If you use Workspace I/O, To Workspace, To File, or Scope blocks, large amounts of data will slow Simulink Accelerator down. Try using decimation or limiting outputs to the last *N* data points.
- Large models — In both accelerator and normal mode, Simulink can take significant time to initialize large models. Simulink Accelerator speedup can be minimal if run-times (from start to finish of a single simulation) are small.

Blocks That Do Not Show Speed Improvements

The Simulink Accelerator speeds up execution only of blocks from Simulink and Signal Processing Blockset. Further, Simulink Accelerator does not improve the performance of some blocks in Simulink. The following sections list these blocks.

Simulink Blocks

- Display

- Embedded MATLAB Function
- From File
- From Workspace
- Inport (root level only)
- MATLAB Fcn
- Outport (root level only)
- Scope
- To File
- To Workspace
- Transport Delay
- Variable Transport Delay
- XY Graph

User-Written S-Function Blocks

In addition, Simulink Accelerator does not speed up user-written S-Function blocks unless you inline them using the Target Language Compiler and set `SS_OPTION_USE_TLC_WITH_ACCELERATOR` in the S-function itself. See "Controlling S-Function Execution" on page 15-14 for more information.

Using Simulink Accelerator with the Simulink Debugger

If you have large and complex models that you need to debug, Simulink Accelerator can shorten the length of your debugging sessions. For example, if you need to set a time break that is very large, you can use Simulink Accelerator to reach the breakpoint more quickly.

To run the Simulink debugger while in accelerator mode:

1. Select **Accelerator** from the **Simulation** menu, then enter

   ```
   sldebug modelname
   ```

   at the MATLAB command prompt.

2 At the debugger prompt, set a time break:

```
tbreak 10000
continue
```

3 Once you reach the breakpoint, use the debugger command `emode` (execution mode) to toggle between accelerator and normal mode.

Note that you must switch to normal mode to step the simulation by blocks. You must also switch to normal mode to use the following debug commands:

- `trace`
- `break`
- `zcbreak`
- `nanbreak`

For more information on the Simulink debugger, see Chapter 14, “Simulink Debugger”.

Interacting with Simulink Accelerator Programmatically

Using three commands, `set_param`, `sim`, and `accelbuild`, you can control the execution of your model from the MATLAB prompt or from M-files. This section describes the syntax for these commands and the options available.

Controlling the Simulation Mode

You can control the simulation mode from the MATLAB prompt using

```
set_param(gcs,'simulationmode','mode')
```

or

```
set_param(modelname,'simulationmode','mode')
```

You can use `gcs` (“get current system”) to set parameters for the currently active model (i.e., the active model window) and `modelname` if you want to specify the model name explicitly. The simulation mode can be normal, accelerator, or external.

Simulating an Accelerated Model

You can also simulate an accelerated model using

```
sim(gcs);   % Blocks the MATLAB prompt until simulation complete
```

or

```
set_param(gcs,'simulationcommand','start'); % Returns to the
                                             % MATLAB prompt
                                             % immediately
```

Again, you can substitute the modelname for gcs if you prefer to specify the model explicitly.

Building Simulink Accelerator MEX-Files Independent of Simulation

You can build the Simulink Accelerator MEX-file without actually simulating the model by using the accelbuild command, for example,

```
accelbuild f14
```

Creating the Simulink Accelerator MEX-files in batch mode using accelbuild allows you to build the C code and executables prior to running your simulations. When you use Simulink Accelerator interactively at a later time, it does not need to generate or compile MEX-files at the start of the accelerated simulations.

You can use the accelbuild command to specify build options, such as including debug information in the generated MEX-file.

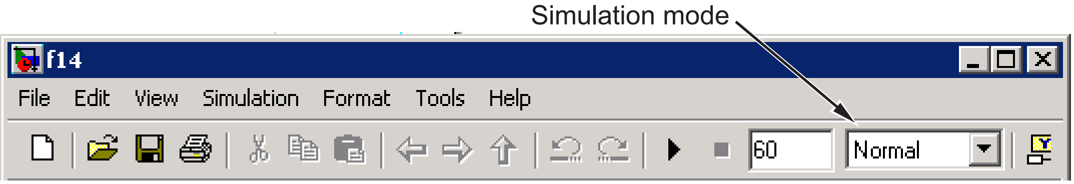
```
accelbuild f14 OPT_OPTS=-g
```

Refer to your compiler's documentation for more information about its particular build options.

Comparing Performance

If you need to quantify the improvement in simulation speed that you get using Simulink Accelerator, you can use the tic, toc, and sim commands. These commands provide a comparison of the performance of Simulink Accelerator to Simulink in normal mode. For example:

1 Open the f14 model. By default, the simulation is set to normal mode.



2 Measure the performance of the model in normal simulation mode by typing the following code at the MATLAB command prompt:

```
tic,[t,x,y]=sim('f14',10000);toc
```

The tic and toc commands work together to record and return the elapsed time. The elapsed time provides a measure of how long the model takes to simulate.

```
Elapsed time is 17.789364 seconds.
```

3 Change the simulation mode by selecting Accelerator from the **Simulation mode** menu in the model editor's toolbar. You can alternatively select **Simulation > Accelerator** from the model editor's menu bar.

4 Before measuring the performance of the model, you must run the simulation once using Simulink Accelerator to build an executable for the model. Simulink Accelerator uses this executable in subsequent simulations as long as the model remains structurally unchanged. To run the model, click the **Start** button on the model editor's toolbar. Simulink Accelerator displays the code generation steps in the MATLAB Command Window.

5 Rerun the model at the MATLAB command prompt using the following command:

```
tic,[t,x,y]=sim('f14',10000);toc
```

The resulting elapsed time shows the improvement achieved by Simulink Accelerator.

```
Elapsed time is 12.419914 seconds.
```

Note Simulink Accelerator checks at the beginning of each simulation to determine whether it must regenerate the executable. This step adds a small overhead to the run-time and, as a result, the normal mode simulation could be faster for models with very short simulation times.

Customizing the Simulink Accelerator Build Process

Typically, no customization is necessary for the Simulink Accelerator build process. However, you can use the following parameters to control different portions of the build process and the information displayed by the compiler.

Use this parameter	To
`AccelMakeCommand`	Specify a custom make command
`AccelSystemTargetFile`	Specify the Simulink Accelerator target for your model
`AccelTemplateMakeFile`	Specify the make file for your model
`AccelVerboseBuild`	Determine how much information is displayed when Simulink Accelerator generates and compiles code

Proper use of these parameters requires an understanding of how Real-Time Workshop generates code. For a description, see the *Real-Time Workshop User's Guide*, which is available on the MathWorks Web site, `http://www.mathworks.com`.

See "Simulink Accelerator" on page 15-2 for details of how the Simulink Accelerator operates.

For a description of the these parameters and the values they take see “Model Parameters”.

AccelVerboseBuild is 'off' by default, which produces a short form listing in the MATLAB Command Window. Set it 'on' to display progress information during code generation.

For example, the following causes output information to be displayed when compiling 'my_model'.

```
set_param('my_model', 'AccelVerboseBuild', 'on')
```

Controlling S-Function Execution

Inlining S-functions using the Target Language Compiler increases performance when used with Simulink Accelerator. By default, however, Simulink Accelerator ignores an inlining TLC file for an S-function, even though the file exists.

One example of why this default was chosen is a device driver S-Function block for an I/O board. The S-function TLC file is typically written to access specific hardware registers of the I/O board. Because Simulink Accelerator is not running on a target system, but rather is a simulation on the host system, it must avoid using the inlined TLC file for the S-function.

Another example is when the TLC file and MEX-file versions of an S-function are not compatible in their use of work vectors, parameters, and/or initialization.

If your inlined S-function is not complicated by these issues, you can direct Simulink Accelerator to use the TLC file instead of the S-function MEX-file by specifying SS_OPTION_USE_TLC_WITH_ACCELERATOR in the mdlInitializeSizes function of the S-function. When set, Simulink Accelerator uses the inlining TLC file and full performance increases are realized. For example:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* Code deleted */
ssSetOptions(S, SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}
```

Determining Why Simulink Accelerator Rebuilds

Sometimes, for no apparent reason, Simulink Accelerator regenerates the simulation target for a model at the beginning of every simulation. You can determine why by capturing and comparing a checksum that Simulink uses to determine whether the code needs to be regenerated. The checksum is an array of four integers computed by using an MD5 checksum algorithm based on attributes of the model and the blocks it contains.

To capture and compare the checksum,

1 Capture two instances of the model's checksum data by using the `Simulink.BlockDiagram.getChecksum` command. For example:

```
[cs1, csdet1] = Simulink.BlockDiagram.getChecksum('myModel');
[cs2, csdet2] = Simulink.BlockDiagram.getChecksum('myModel');
```

2 Compare the two checksum values returned, for example, `cs1` and `cs2`.

3 Examine the data provided by each of the second output arguments. Search for and identify differences and then examine the data.

4 Change the model configuration so the model is not rebuilt.

5 Verify that Simulink Accelerator does not regenerate code for each simulation run.

For an example of this procedure, see the demo model `slAccelDemoWhyRebuild`.

Profiler

The Simulink simulation profiler collects performance data while simulating your model and generates a report, called a *simulation profile*, based on the data. The simulation profile generated by the profiler shows you how much time Simulink spends executing each function required to simulate your model. The profile enables you to determine the parts of your model that require the most time to simulate and hence where to focus your model optimization efforts.

You must have the Simulink Accelerator product installed on your system to use the profiler.

- "How the Profiler Works" on page 15-16
- "Enabling the Profiler" on page 15-18
- "The Simulation Profile" on page 15-19

How the Profiler Works

The following pseudocode summarizes the execution model on which the profiler is based.

```
Sim()
 ModelInitialize().
 ModelExecute()
  for t = tStart to tEnd
  Output()
  Update()
  Integrate()
   Compute states from derivs by repeatedly calling:
    MinorOutput()
    MinorDeriv()
   Locate any zero crossings by repeatedly calling:
    MinorOutput()
    MinorZeroCrossings()
  EndIntegrate
  Set time t = tNew.
 EndModelExecute
 ModelTerminate
EndSim
```

According to this conceptual model, Simulink executes a Simulink model by invoking the following functions zero, one, or more times, depending on the function and the model.

Function	Purpose	Level
`sim`	Simulate the model. This top-level function invokes the other functions required to simulate the model. The time spent in this function is the total time required to simulate the model.	System
`ModelInitialize`	Set up the model for simulation.	System
`ModelExecute`	Execute the model by invoking the output, update, integrate, etc., functions for each block at each time step from the start to the end of simulation.	System
`Output`	Compute the outputs of a block at the current time step.	Block
`Update`	Update a block’s state at the current time step.	Block
`Integrate`	Compute a block’s continuous states by integrating the state derivatives at the current time step.	Block
`MinorOutput`	Compute a block’s output at a minor time step.	Block
`MinorDeriv`	Compute a block’s state derivatives at a minor time step.	Block

Function	Purpose	Level
`MinorZeroCrossings`	Compute a block's zero-crossing values at a minor time step.	Block
`ModelTerminate`	Free memory and perform any other end-of-simulation cleanup.	System
`Nonvirtual Subsystem`	Compute the output of a nonvirtual subsystem (see "Solvers" on page 1-18) at the current time step by invoking the output, update, integrate, etc., functions for each block that it contains. The time spent in this function is the time required to execute the nonvirtual subsystem.	Block

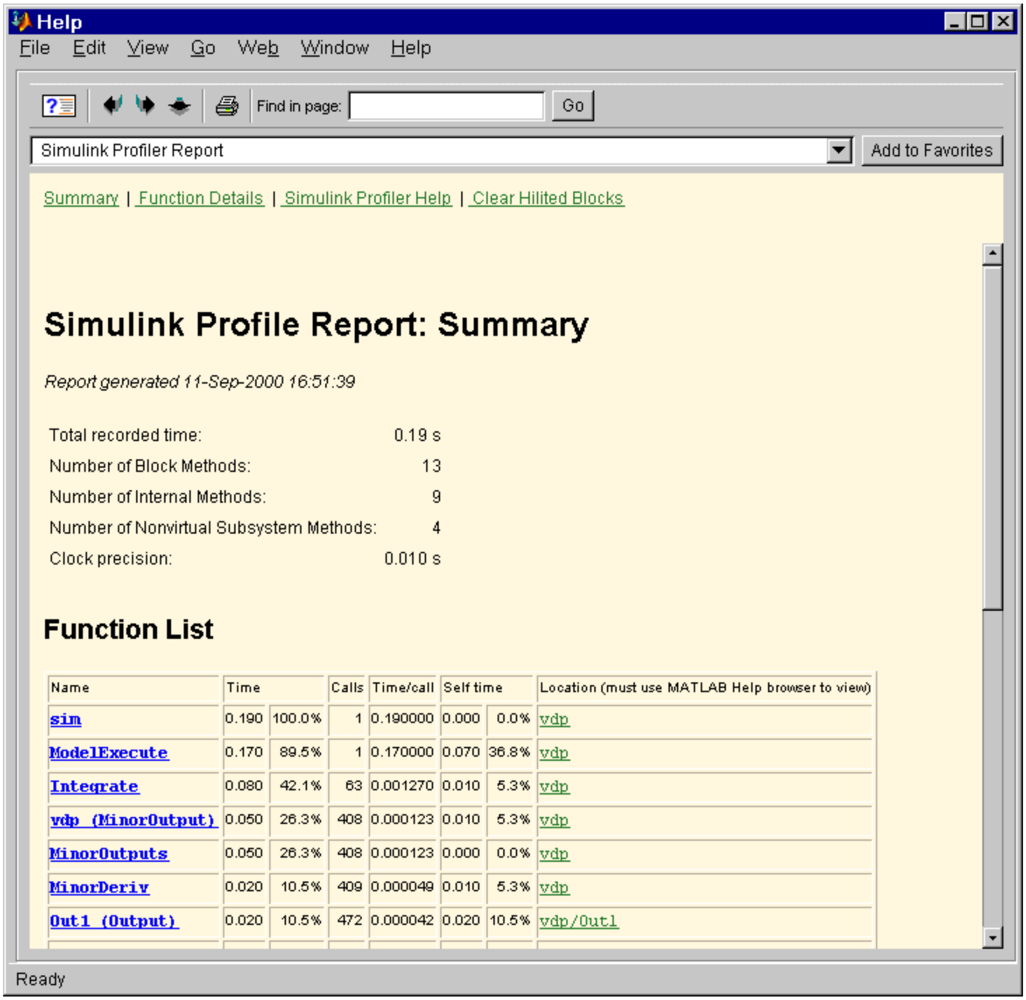
The profiler measures the time required to execute each invocation of these functions and generates a report at the end of the model that describes how much time was spent in each function.

Enabling the Profiler

To profile a model, open the model and select **Profiler** from the Simulink **Tools** menu. Then start the simulation. When the simulation finishes, Simulink generates and displays the simulation profile for the model in the MATLAB Help browser.

The Simulation Profile

Simulink stores the simulation profile in the MATLAB working directory.


Help
File Edit View Go Web Window Help
Find in page:
Go
Simulink Profiler Report
Add to Favorites
Summary | Function Details | Simulink Profiler Help | Clear Hilited Blocks
Ready

Simulink Profile Report: Summary

Report generated 11-Sep-2000 16:51:39

Total recorded time:	0.19 s
Number of Block Methods:	13
Number of Internal Methods:	9
Number of Nonvirtual Subsystem Methods:	4
Clock precision:	0.010 s

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Help browser to view)
sim	0.190	100.0%	1	0.190000	0.000	0.0%	vdp
ModelExecute	0.170	89.5%	1	0.170000	0.070	36.8%	vdp
Integrate	0.080	42.1%	63	0.001270	0.010	5.3%	vdp
vdp (MinorOutput)	0.050	26.3%	408	0.000123	0.010	5.3%	vdp
MinorOutputs	0.050	26.3%	408	0.000123	0.000	0.0%	vdp
MinorDeriv	0.020	10.5%	409	0.000049	0.010	5.3%	vdp
Out1 (Output)	0.020	10.5%	472	0.000042	0.020	10.5%	vdp/Out1

The report has two sections: a summary and a detailed report.

Summary Section

The summary file displays the following performance totals.

Item	Description
Total Recorded Time	Total time required to simulate the model
Number of Block Methods	Total number of invocations of block-level functions (e.g., `Output()`)
Number of Internal Methods	Total number of invocations of system-level functions (e.g., `ModelExecute`)
Number of Nonvirtual Subsystem Methods	Total number of invocations of nonvirtual subsystem functions
Clock Precision	Precision of the profiler's time measurement

The summary section then shows summary profiles for each function invoked to simulate the model. For each function listed, the summary profile specifies the following information.

Item	Description
Name	Name of function. This item is a hyperlink. Clicking it displays a detailed profile of this function.
Time	Total time spent executing all invocations of this function as an absolute value and as a percentage of the total simulation time
Calls	Number of times this function was invoked
Time/Call	Average time required for each invocation of this function, including the time spent in functions invoked by this function
Self Time	Average time required to execute this function, excluding time spent in functions called by this function
Location	Specifies the block or model executed for which this function is invoked. This item is a hyperlink. Clicking it highlights the corresponding icon in the model diagram. That the link works only if you are viewing the profile in the MATLAB Help browser.

Detailed Profile Section

This section contains detailed profiles for each function invoked to simulate the model. Each detailed profile contains all the information shown in the summary profile for the function. In addition, the detailed profile displays the function (parent function) that invoked the profiled function and the functions (child functions) invoked by the profiled function. Clicking the name of the parent or a child function takes you to the detailed profile for that function.

16

Customizing the Simulink User Interface

Adding Items to Model Editor Menus

Simulink allows you to add commands and submenus to the end of Simulink model editor menus. Adding an item to the end of a Simulink Model Editor menu entails performing the following tasks:

- For each item, create a function, called a *schema function*, that defines the item (see "Defining Menu Items" on page 16-4).
- Register the menu customizations with the Simulink customization manager at Simulink startup, e.g., in an `sl_customization.m` file on the MATLAB path (see "Registering Menu Customizations" on page 16-7).
- Create callback functions that implement the commands triggered by the items that you add to the Simulink menus.

Note You can use the procedures described in the following sections to customize the Stateflow® Chart Editor's menu.

For more information on adding items to Model Editor Menus, see:

- "Code Example" on page 16-2
- "Defining Menu Items" on page 16-4
- "Registering Menu Customizations" on page 16-7
- "Callback Info Object" on page 16-8
- "Debugging Custom Menu Callbacks" on page 16-9
- "About Menu Tags" on page 16-9

Code Example

The following `sl_customization.m` file adds three items to the Simulink editor's **Tools** menu.

```
function sl_customization(cm)

  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyMenuItems);
```

```
end

%% Define the custom menu function.
function schemaFcns = getMyMenuItems(callbackInfo)
  schemaFcns = {@getItem1,...
      @getItem2,...
      {@getItem3,3}}; %% Pass 3 as user data to getItem3.
end

%% Define the schema function for first menu item.
function schema = getItem1(callbackInfo)
  schema = sl_action_schema;
  schema.label = 'Item One';
  schema.userdata = 'item one';
  schema.callback = @myCallback1;
end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata ' was called']);
end

function schema = getItem2(callbackInfo)
  % Make a submenu label 'Item Two' with
  % the menu item above three times.
  schema = sl_container_schema;
  schema.label = 'Item Two';
 schema.childrenFcns = {@getItem1, @getItem1, @getItem1};
end

function schema = getItem3(callbackInfo)
  % Create a menu item whose label is
  % 'Item Three: 3', with the 3 being passed
  % from getMyItems above.

  schema = sl_action_schema;
  schema.label = ['Item Three: ' num2str(callbackInfo.userdata)];
end
```

Defining Menu Items

You define a menu item by creating a function that returns an object, called a *schema* object, that specifies the information needed to create the menu item. The menu item that you define may trigger a custom action or display a custom submenu. See the following sections for more information.

- "Defining Menu Items That Trigger Custom Commands" on page 16-4
- "Defining Custom Submenus" on page 16-5

Defining Menu Items That Trigger Custom Commands

To define an item that triggers a custom command, your schema function must accept a callback info object (see "Callback Info Object" on page 16-8) and create and return an action schema object (see "Action Schema Object" on page 16-5) that specifies the item's label and a function, called a *callback*, to be invoked by Simulink when the user selects the item. For example, the following schema function defines a menu item that displays a message when selected by the user.

```
function schema = getItem1(callbackInfo)

  %% Create an instance of an action schema.
  schema = sl_action_schema;

  %% Specify the menu item's label.
  schema.label = 'My Item 1';

  %% Specify the menu item's callback function.
  schema.callback = @myCallback1;

end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata
        ' was called']);
end
```

Action Schema Object. This object specifies information about menu items that trigger commands that you define, including the label that appears on the menu item and the function to be invoked when the user selects the menu item. Use the function `sl_action_schema` to create instances of this object in your schema functions. Its properties include

- `tag`

  Optional string that identifies this action, for example, so that it can be referenced by a filter function.

- `label`

  String specifying the label that appears on a menu item that triggers this action.

- `state`

  String that specifies the state of this action. Valid values are `'Enabled'` (the default), `'Disabled'`, and `'Hidden'`.

- `statustip`

  String specifying text to appear in the editor's status bar when the user selects the menu item that triggers this action.

- `userdata`

  Data that you specify. May be of any type.

- `accelerator`

  String specifying a keyboard shortcut that a user may use to trigger this action. The string must be of the form `'Ctrl+`*`K`*`'`, where *`K`* is the shortcut key. For example, `'Ctrl+T'` specifies that the user may invoke this action by holding down the **Ctrl** key and pressing the **T** key.

- `callback`

  String specifying a MATLAB expression to be evaluated or a handle to a function to be invoked when a user selects the menu item that triggers this action. This function must accept one argument: a callback info object.

Defining Custom Submenus

To define a submenu, create a schema function that accepts a callback info object and returns a container schema object (see "Container Schema

Object" on page 16-6) that specifies the schemas that define the items on the submenu. For example, the following schema function defines a submenu that contains three instances of the menu item defined in the example in "Defining Menu Items That Trigger Custom Commands" on page 16-4.

```
function schema = getItem2( callbackInfo )
    schema = sl_container_schema;
    schema.label = 'Item Two';
    schema.childrenFcns = {@getItem1, @getItem1, @getItem1};
end
```

Container Schema Object. A container schema object specifies a submenu's label and its contents. Use the function `sl_container_schema` to create instances of this object in your schema functions. Properties of the object include

- `tag`

  Optional string that identifies this submenu.

- `label`

  String specifying the submenu's label.

- `state`

  String that specifies the state of this submenu. Valid values are `'Enabled'` (the default), `'Disabled'`, and `'Hidden'`.

- `statustip`

  String specifying text to appear in the editor's status bar when the user selects this submenu.

- `userdata`

  Data that you specify. May be of any type.

- `childrenFcns`

  Cell array that specifies the contents of the submenu. Each entry in the cell array can be

  - a pointer to a schema function that defines an item on the submenu (see "Defining Menu Items" on page 16-4)

  - a two-element cell array whose first element is a pointer to a schema function that defines an item entry and whose second element is data to be inserted as user data in the callback info object (see “Callback Info Object” on page 16-8) passed to the schema function
  - `'separator'`, which causes a separator to appear between the item defined by the preceding entry in the cell array and the item defined in the following entry. Simulink ignores case for this entry, e.g., `'SEPARATOR'` and `'Separator'` are valid entries. Simulink also suppresses a separator if it would appear at the beginning or end of the submenu and combines separators that would appear successively (e.g., as a result of an item being hidden) into a single separator.

  For example, the following cell array specifies two submenu entries:

  ```
  {@getItem1, 'separator', {@getItem2, 1}}
  ```

  In this example, Simulink passes 1 to `getItem2` via a callback info object.

- `generateFcn`

  Pointer to a function that returns a cell array defining the contents of the submenu. The cell array must have the same format as that specified for the container schema objects `childrenFcns` property.

Note The `generateFcn` property takes precedence over the `childrenFcns` property. If you set both, Simulink ignores the `childrenFcns` property and uses the cell array returned by the `generateFcn` to create the submenu.

Registering Menu Customizations

You must register custom items to be included on a Simulink menu with the Simulink customization manager. Use the `sl_customization.m` file for a Simulink installation (see “Registering Customizations” on page 16-19) to perform this task. In particular, for each menu that you want to customize, your system’s `sl_customization` function must invoke the customization manager’s `addCustomMenuFcn` method (see “Customization Manager” on page 16-19). Each invocation should pass the tag of the menu (see “About Menu Tags” on page 16-9) to be customized and a custom menu function that specifies the items to be added to the menu (see “Creating the Custom

Menu Function" on page 16-8) . For example, the following sl_customization function adds custom items to the Simulink Tools menu.

```
function sl_customization(cm)
  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyItems);
```

Creating the Custom Menu Function

The custom menu function returns a list of schema functions that define custom items that you want to appear on Simulink model editor menus (see "Defining Menu Items" on page 16-4).

Your custom menu function should accept a callback info object (see "Callback Info Object" on page 16-8) and return a cell array that lists the schema functions. Each element of the cell array can be either a handle to a schema function or a two-element cell array whose first element is a handle to a schema function and whose second element is user-defined data to be passed to the schema function. For example, the following custom menu function returns a cell array that lists three schema functions.

```
function schemas = getMyItems(callbackInfo)
  schemas = {@getItem1, ...
           @getItem2, ...
             {@getItem3,3} }; % Pass 3 as userdata to getItem3.
end
```

Callback Info Object

Simulink passes instances of these objects to menu customization functions. Properties of these objects include

- uiObject

  Handle to the owner of the menu for which this is the callback. The owner can be the Simulink or Stateflow editor.

- model

  Handle to the model being displayed in the editor window.

- `userdata`

  User data. The value of this field can be any type of data.

Debugging Custom Menu Callbacks

On Microsoft Windows, selecting a custom menu item whose callback contains a breakpoint can cause the mouse to become unresponsive or the menu to remain open and on top of other windows. To fix these problems, use the M-file debugger's keyboard commands to continue execution of the callback.

About Menu Tags

A menu tag is a string that identifies a Simulink Model Editor or Stateflow Chart Editor menu bar or menu. You need to know a menu's tag to add custom items to it (see "Registering Menu Customizations" on page 16-7). You can configure the editor to display all (see "Displaying Menu Tags" on page 16-9) but the following tags:

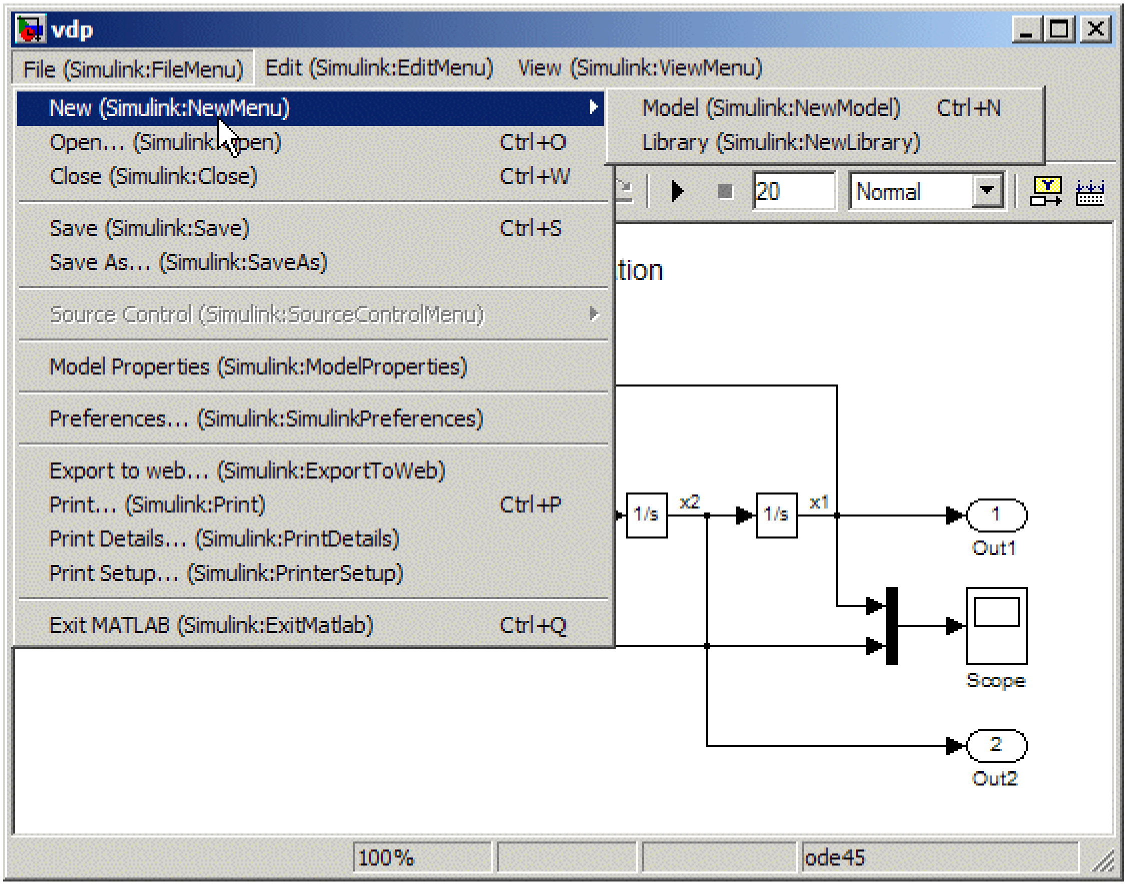
Tag	Usage
Simulink:MenuBar	Add menus to Model Editor's menu bar.
Simulink:ContextMenu	Add items to the end of Model Editor's context menu.
Simulink:PreContextMenu	Add items to the beginning of Model Editor's context menu.
Stateflow:MenuBar	Add menus to Chart Editor's menu bar.
Stateflow:ContextMenu	Add items to the end of Chart Editor's context menu.
Stateflow:PreContextMenu	Add items to the beginning of Chart Editor's context menu.

Displaying Menu Tags

You can configure Simulink (and Stateflow) to display the tag for a menu item next to the item's label, allowing you to determine at a glance the tag for a menu. To configure the editor to display menu tags, set the customization manager's showWidgetIdAsToolTip property to true, e.g., by entering the following commands at the MATLAB command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip=true;
```

The tag of each menu item appears next to the item's label on the menu:


vdp
File (Simulink:FileMenu)
Edit (Simulink:EditMenu)
View (Simulink:ViewMenu)
New (Simulink:NewMenu)
Model (Simulink:NewModel) Ctrl+N
Library (Simulink:NewLibrary)
Open... (Simulink:Open) Ctrl+O
Close (Simulink:Close) Ctrl+W
Save (Simulink:Save) Ctrl+S
Save As... (Simulink:SaveAs)
Source Control (Simulink:SourceControlMenu)
Model Properties (Simulink:ModelProperties)
Preferences... (Simulink:SimulinkPreferences)
Export to web... (Simulink:ExportToWeb)
Print... (Simulink:Print) Ctrl+P
Print Details... (Simulink:PrintDetails)
Print Setup... (Simulink:PrinterSetup)
Exit MATLAB (Simulink:ExitMatlab) Ctrl+Q
20
Normal
tion
1/s
x2
1/s
x1
1
Out1
Scope
2
Out2
100%
ode45

To turn off tag display, enter the following command at the MATLAB command line:

```
cm.showWidgetIdAsToolTip=false;
```

Note Some menu items may not work while menu tag display is enabled. To ensure that all items work, turn off menu tag display before using the menus.

Disabling and Hiding Model Editor Menu Items

Simulink allows you to disable or hide items that appear on Simulink model editor menus. To disable or hide a menu item, you must

- Create a filter function that disables or hides the menu item (see "Creating a Filter Function" on page 16-12).
- Register the filter function with the Simulink customization manager (see "Registering a Filter Function" on page 16-13).

For more information on Model Editor menu items, see:

- "Example: Disabling the New Model Command on the Simulink Editor's File Menu" on page 16-12
- "Creating a Filter Function" on page 16-12
- "Registering a Filter Function" on page 16-13

Example: Disabling the New Model Command on the Simulink Editor's File Menu

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end

function state = myFilter(callbackInfo)
  state = 'Disabled';
end
```

Creating a Filter Function

Your filter function must accept a callback info object and return a string that specifies the state that you want to assign to the menu item. Valid states are

- `'Hidden'`
- `'Disabled'`
- `'Enabled'`

Your filter function may have to compete with other filter functions and with Simulink itself to assign a state to an item. Who succeeds depends on the strength of the state that each assigns to the item. `'Hidden'` is the strongest state. If any filter function or Simulink assigns `'Hidden'` to the item, it is hidden. `'Enabled'` is the weakest state. For an item to be enabled, all filter functions and Simulink or Stateflow must assign `'Enabled'`to the item. The `'Disabled'` state is of middling strength. It overrides `'Enabled'` but is itself overridden by `'Hidden'`. For example, if any filter function or Simulink or Stateflow assigns `'Disabled'` to a menu item and none assigns `'Hidden'` to the item, the item is disabled.

Note Simulink does not allow you to filter some menu items, for example, the **Exit MATLAB** item on the Simulink **File** menu. Simulink displays an error message if you attempt to filter a menu item that you are not allowed to filter.

Registering a Filter Function

Use the customization manager's `addCustomFilterFcn` method to register a filter function. The `addCustomFilterFcn` method takes two arguments: a tag that identifies the menu or menu item to be filtered (see "Displaying Menu Tags" on page 16-9) and a pointer to the filter function itself. For example, the following code registers a filter function for the **New Model** item on the Simulink **File** menu.

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end
```

Disabling and Hiding Dialog Box Controls

Simulink includes a customization API that allows you to disable and hide controls (also referred to as *widgets*), such as text fields and buttons, on most Simulink dialog boxes. The customization API allows you to disable or hide controls on an entire class of dialog boxes, for example, parameter dialog boxes via a single method call.

Before attempting to customize a Simulink dialog box or class of dialog boxes, you must first ensure that the dialog box or class of dialog boxes is customizable. Any dialog box that appears in the dialog pane of Model Explorer is customizable. In addition, any dialog box that has dialog and widget IDs is customizable. To determine whether a standalone dialog box (i.e., one that does not appear in Model Explorer) is customizable, open the dialog box, enable dialog and widget ID display (see "Dialog Box and Widget IDs" on page 16-17), and position the mouse over a widget. If a widget ID appears, the dialog box is customizable.

Once you have determined that a dialog box or class of dialog boxes is customizable, you must write M-code to customize the dialog boxes. This entails writing callback functions that disable or hide controls for a specific dialog box or class of dialog boxes (see "Writing Control Customization Callback Functions" on page 16-16) and registering the callback functions via an object called the customization manager (see "Registering Control Customization Callback Functions" on page 16-18). Simulink then invokes the callback functions to disable or hide the controls whenever a user opens the dialog boxes.

For more information on Dialog Box controls, see:

- "Example: Disabling a Button on a Simulink Dialog Box" on page 16-15
- "Writing Control Customization Callback Functions" on page 16-16
- "Dialog Box Methods" on page 16-16
- "Dialog Box and Widget IDs" on page 16-17
- "Registering Control Customization Callback Functions" on page 16-18

Example: Disabling a Button on a Simulink Dialog Box

The following `sl_customization.m` file disables the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box for any model whose name contains “engine.”

```
function sl_customization(cm)

% Disable for standalone Configuration Parameters dialog box.
cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)
% Disable for Configuration Parameters dialog box that appears in
% the Model Explorer.
cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end

function disableRTWBuildButton(dialogH)
  hSrc   = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName   = get_param(hModel, 'Name');

  if ~isempty(strfind(modelName, 'engine'))
    % Takes a cell array of widget Factory ID.
    dialogH.disableWidgets({'Simulink.RTWCC.Build'})
  end

end
```

To test this customization:

1. Put the preceding `sl_customization.m` file on the MATLAB path.

2. Register the customization by entering `sl_refresh_customizations` at the MATLAB command line or by restarting MATLAB (see “Registering Customizations” on page 16-19).

3. Open the `sldemo_engine` demo model, for example, by entering the command `sldemo_engine` at the MATLAB command line.

Writing Control Customization Callback Functions

A callback function for disabling or hiding controls on a dialog box should accept one argument: a handle to the dialog box object that contains the controls you want to disable or hide. The dialog box object provides methods that the callback function can use to disable or hide the controls that the dialog box contains.

The dialog box object also provides access to objects containing information about the current model. Your callback function can use these objects to determine whether to disable or hide controls. For example, the following callback function uses these objects to disable the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box displayed in Model Explorer for any model whose name contains “engine.”

```
function disableRTWBuildButton(dialogH)

  hSrc   = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName   = get_param(hModel, 'Name');

  if ~isempty(strfind(modelName, 'engine'))
     % Takes a cell array of widget Factory ID.
     dialogH.disableWidgets({'Simulink.RTWCC.Build'})
  end
```

Dialog Box Methods

Dialog box objects provide the following methods for enabling, disabling, and hiding controls:

- `disableWidgets(widgetIDs)`
- `hideWidgets(widgetIDs)`

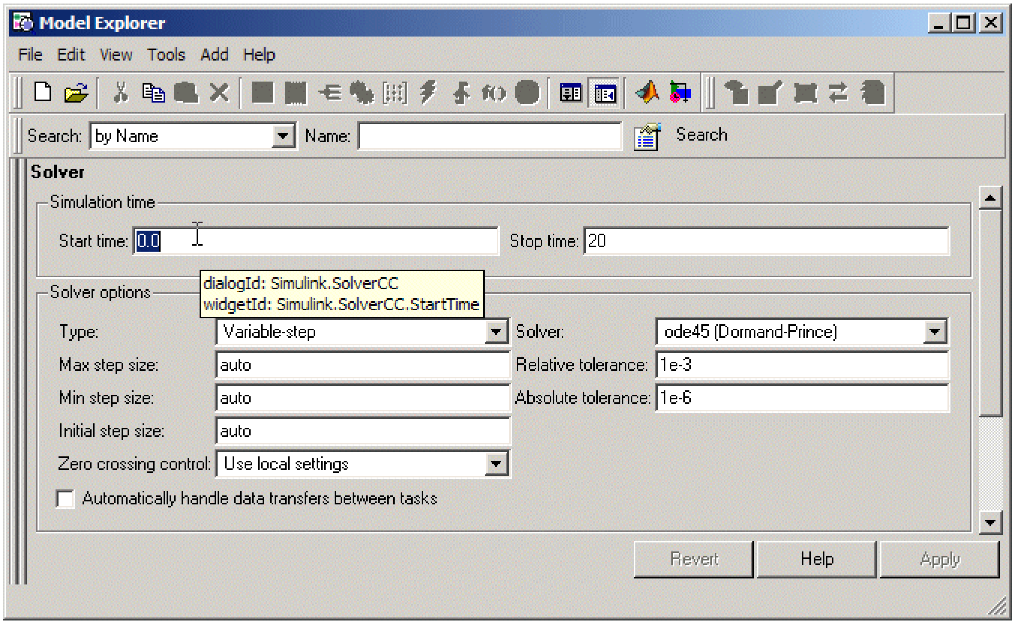
where `widgetIDs` is a cell array of widget identifiers (see “Dialog Box and Widget IDs” on page 16-17) that specify the widgets to be disabled or hidden.

Dialog Box and Widget IDs

Dialog box and widget IDs are strings that identify a control on a Simulink dialog box. To determine the dialog box and widget ID for a particular control, execute the following code at the MATLAB command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip = true
```

Then, open the dialog box that contains the control and move the mouse cursor over the control to display a tooltip listing the dialog box and the widget IDs for the control. For example, moving the cursor over the **Start time** field on the **Solver** pane of the Configuration Parameters dialog box reveals that the dialog box ID for the **Solver** pane is `Simulink.SolverCC` and the widget ID for the **Start time** field is `Simulink.SolverCC.StartTime`.



Solver

Simulation time

Start time: 0.0

Stop time: 20

dialogId: Simulink.SolverCC
widgetId: Simulink.SolverCC.StartTime

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

□ Automatically handle data transfers between tasks

Note The tooltip displays "not customizable" for controls that are not customizable.

Registering Control Customization Callback Functions

To register control customization callback functions for a particular Simulink installation, include code in the installation's `sl_customization.m` file (see "Registering Customizations" on page 16-19) that invokes the customization manager's `addDlgPreOpenFcn` on the callbacks.

The `addDlgPreOpenFcn` takes two arguments. The first argument is a dialog box ID (see "Dialog Box and Widget IDs" on page 16-17) and the second is a pointer to the callback function to be registered. Invoking this method causes Simulink to invoke the registered function for each dialog box of the type specified by the dialog box ID. Simulink invokes the function before it opens the dialog box, allowing the function to perform the customizations before they become visible to the user.

The following example registers a callback that disables the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box (see "Writing Control Customization Callback Functions" on page 16-16).

```
function sl_customization(cm)

  % Disable for standalone Configuration Parameters dialog box.
  cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)

  % Disable for Configuration Parameters dialog box that appears in
  % the Model Explorer
  cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end
```

Note Registering a customization callback causes Simulink to invoke the callback for every instance of the class of dialog boxes specified by the method's dialog box ID argument. This allows you to use a single callback to disable or hide a control for an entire class of dialog boxes. In particular, you can use a single callback to disable or hide the control for a parameter that is common to most built-in Simulink blocks. This is because most built-in block dialog boxes are instances of the same dialog box super class.

Registering Customizations

You must register your user-interface customizations with Simulink, using an M-file function called `sl_customization.m` located on the MATLAB path of the Simulink installation that you want to customize. The `sl_customization` function should accept one argument: a handle to an object called the `Simulink.CustomizationManager`, e.g.,

```
function sl_customization(cm)
```

The customization manager object includes methods for registering menu and control customizations (see "Customization Manager" on page 16-19). Your instance of the `sl_customization` function should use these methods to register customizations specific to your application. For more information, see the following sections on performing customizations.

- "Adding Items to Model Editor Menus" on page 16-2
- "Disabling and Hiding Model Editor Menu Items" on page 16-12
- "Disabling and Hiding Dialog Box Controls" on page 16-14

Simulink reads the `sl_customization.m` file when it starts. If you subsequently change the `sl_customization.m` file, you must restart Simulink or enter the following command at the MATLAB command line to effect the changes:

```
sl_refresh_customizations
```

Customization Manager

The customization manager includes the following methods:

- `addCustomMenuFcn(stdMenuTag, menuSpecsFcn)`

  Adds the menus specified by `menuSpecsFcn` to the end of the standard Simulink menu specified by `stdMenuTag`. The `stdMenuTag` argument is a string that specifies the menu to be customized. For example, the `stdMenuTag` for the Simulink editor's **Tools** menu is `'Simulink:ToolsMenu'` (see "Displaying Menu Tags" on page 16-9 for more information). The `menuSpecsFcn` argument is a handle to a function that returns a list of functions that specify the items to be added to the specified menu. See “Adding Items to Model Editor Menus” on page 16-2 for more information.

- `addCustomFilterFcn(stdMenuItemID, filterFcn)`

  Adds a custom filter function specified by `filterFcn` for the standard Simulink model editor menu item specified by `stdMenuItemID`. The `stdMenuItemID` argument is a string that identifies the menu item. For example, the ID for the **New Model** item on the Simulink editor’s **File** menu is `'Simulink:NewModel'` (see “Displaying Menu Tags” on page 16-9 for more information) . The `filterFcn` argument is a pointer to a function that hides or disables the specified menu item. See “Disabling and Hiding Model Editor Menu Items” on page 16-12 for more information.

17

Using the Embedded MATLAB Function Block

The Embedded MATLAB Function block lets you include MATLAB code in Simulink models for deployment to embedded processors. This topic explains how to create and debug Embedded MATLAB function blocks in Simulink.

Introduction to Embedded MATLAB Function Blocks

This section introduces the Embedded MATLAB Function block in Simulink.

- “What Is an Embedded MATLAB Function Block?” on page 17-3
- “Why Use Embedded MATLAB Function Blocks?” on page 17-5

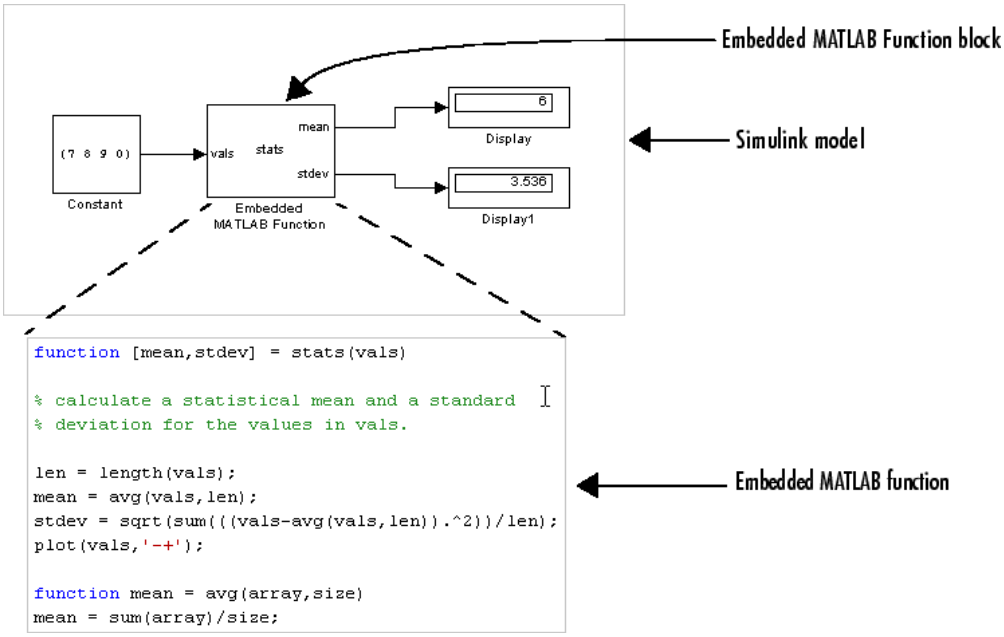
In “Creating an Example Embedded MATLAB Function” on page 17-7, you build a model with an example Embedded MATLAB Function block.

Note For more information on fixed-point support in Embedded MATLAB, refer to “Fixed-Point Embedded MATLAB Features” in the Fixed-Point Toolbox documentation.

What Is an Embedded MATLAB Function Block?

The Embedded MATLAB Function block allows you to add MATLAB functions to a Simulink model. This capability is useful for coding algorithms that are better stated in the textual language of MATLAB than in the graphical language of Simulink. This block works with a subset of the MATLAB language called the Embedded MATLAB subset, which provides optimizations for generating efficient, production-quality C code for embedded applications. For more information, see “Working with Embedded MATLAB” in the Embedded MATLAB documentation.

Here is an example of a Simulink model that contains an Embedded MATLAB Function block:



```
function [mean,stdev] = stats(vals)

% calculate a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

You will build this model in "Creating an Example Embedded MATLAB Function" on page 17-7 .

Note in this Embedded MATLAB function that you can declare local variables implicitly through assignment, just as you would in MATLAB functions. The variable takes its type and size from the context in which it is assigned. For example, the following code line declares x to be a scalar variable of type double.

```
x = 1.54;
```

Once you define a variable, you cannot redefine it to any other type or size in the function body. For example, you cannot declare x and reassign it as follows:

```
x = 2.65; % OK: x is a scalar double
x = [x 2*x]; % Error: x cannot be changed to a vector
```

See “Creating Local Variables Implicitly” in the Embedded MATLAB documentation for detailed descriptions and examples.

In addition to supporting a rich subset of the MATLAB language, Embedded MATLAB Function blocks can call any of the following types of functions:

- **Subfunctions**

  Subfunctions are defined in the body of the Embedded MATLAB block. In the preceding example, `avg` is a subfunction. See “Calling Subfunctions” in the Embedded MATLAB documentation.

- **Embedded MATLAB runtime library functions**

  Embedded MATLAB runtime library functions are a subset of the functions that you call in MATLAB. When you build your model with Real-Time Workshop, these functions generate C code that conforms to the memory and variable type requirements of embedded environments. In the preceding example, `length`, `sqrt`, and `sum` are Embedded MATLAB runtime library functions. See “Calling Embedded MATLAB Run-Time Library Functions” in the Embedded MATLAB documentation.

- **MATLAB functions**

  Function calls that cannot be resolved as subfunctions or Embedded MATLAB runtime library functions are resolved in the MATLAB workspace. These functions do not generate code; they execute only in the MATLAB workspace during simulation of the model. See “Calling MATLAB Functions” in the Embedded MATLAB documentation.

Why Use Embedded MATLAB Function Blocks?

Embedded MATLAB Function blocks provide the following capabilities:

- **Allow you to build MATLAB functions into embeddable applications** — Embedded MATLAB Function blocks support a subset of MATLAB commands that generate efficient C code (see “Embedded MATLAB Run-Time Function Library” in the Embedded MATLAB documentation). With this support, you can use Real-Time Workshop to generate embeddable C code from Embedded MATLAB Function blocks that implements a variety of sophisticated mathematical applications. In this way, you can build executables that harness MATLAB functionality, but run outside the MATLAB environment.

- **Support multiple inputs and outputs** — Unlike MATLAB Fcn blocks, which take a vector input of values and support a single scalar output, Embedded MATLAB Function blocks accept multiple inputs and return multiple outputs.
- **Inherit properties from Simulink input and output signals** — By default, both the size and type of input and output signals to an Embedded MATLAB Function block are inherited from Simulink signals. You can also choose to specify the size and type of inputs and outputs explicitly in the **Model Explorer** or **Ports and Data Manager** (see "Ports and Data Manager" on page 17-42).

Creating an Example Embedded MATLAB Function

Use the following procedure topics to create a model with an Embedded MATLAB Function block. In the process, learn how to use Embedded MATLAB Function blocks in Simulink.

1. "Adding an Embedded MATLAB Function Block to a Model" on page 17-7 -- Start by creating a model with an Embedded MATLAB Function block.

2. "Programming the Embedded MATLAB Function" on page 17-9 -- Describes how to program an Embedded MATLAB function. Provides an overview of program syntax, how to use local variables, and how to write callable functions.

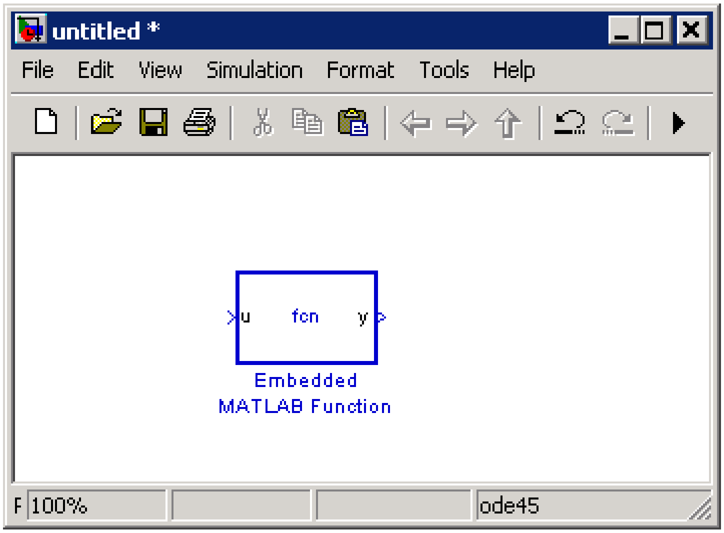
3. "Checking the Function for Errors" on page 17-15 -- Use the built-in diagnostics for Embedded MATLAB Function blocks to test for syntax errors in the Embedded MATLAB function body.

4. "Defining Inputs and Outputs" on page 17-18 -- Define properties for the input and output arguments of the Embedded MATLAB Function block interface with the **Model Explorer** or **Ports and Data Manager** (see "Ports and Data Manager" on page 17-42).

Adding an Embedded MATLAB Function Block to a Model

Start by creating an empty Simulink model and filling it with an Embedded MATLAB Function block, and other blocks necessary to complete the model.

1 Create a new Simulink model and add an Embedded MATLAB Function block to it from the User-Defined Function library of the Simulink library.



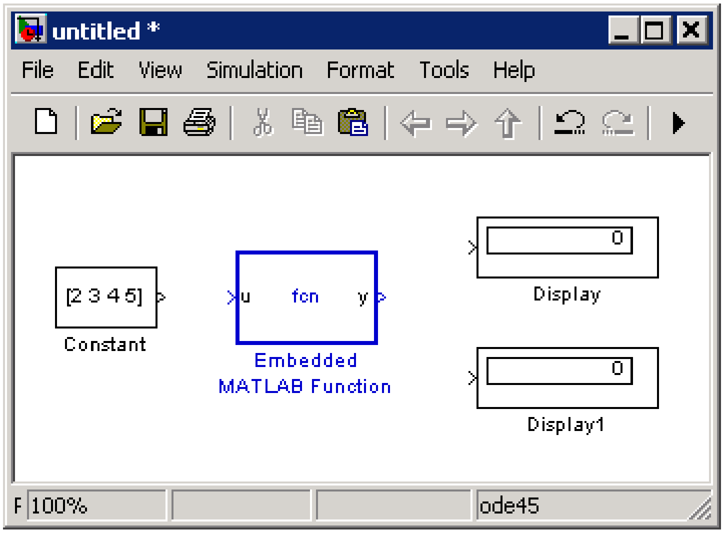
An Embedded MATLAB Function block has two names. The name in the middle of the block is the name of the function you build for the Embedded MATLAB Function block. Its name defaults to `fcn`. The name at the bottom of the block is the name of the block itself. Its name defaults to `Embedded MATLAB Function`.

The default Embedded MATLAB Function block has an input port and an output port. The input port is associated with the input argument `u`, and the output port is associated with the output argument `y`.

2 Add the following Source and Sink blocks to the model:

- From the Simulink Sources library, add a Constant block to the left of the Embedded MATLAB Function block and set its value to the vector `[2 3 4 5]`.
- From the Simulink Sinks library, add two Display blocks to the right of the Embedded MATLAB Function block.

The model should now have the following appearance:



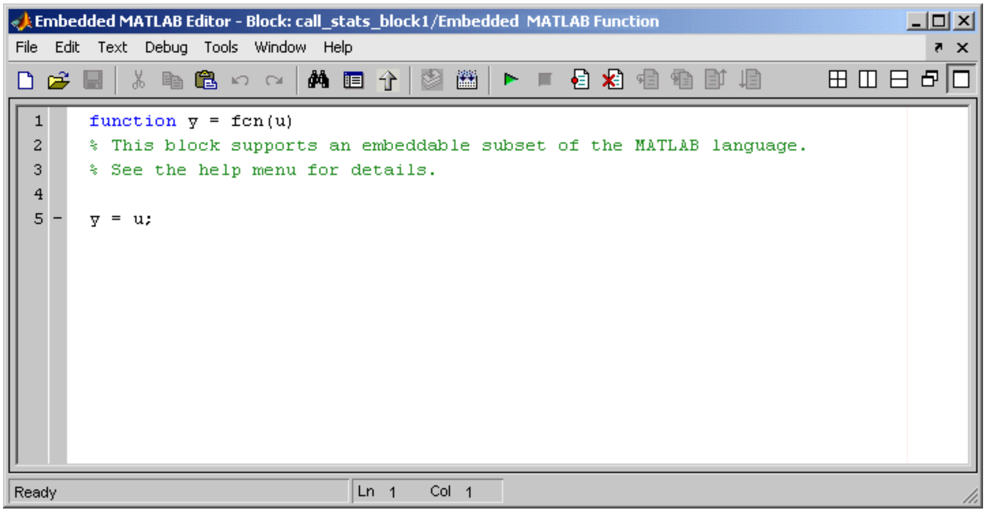
3 In the Simulink window, from the **File** menu, select **Save As** and save the model as `call_stats_block1`.

Programming the Embedded MATLAB Function

You create a model with an Embedded MATLAB Function block in "Creating an Example Embedded MATLAB Function" on page 17-7. Now you want to add code to the block to define it as a function that takes a vector set of values and calculates the mean and standard deviation for those values. Use the following steps to program the function `stats`:

1 If not already open, open the `call_stats_block1` model that you save at the end of "Adding an Embedded MATLAB Function Block to a Model" on page 17-7, and double-click its Embedded MATLAB Function block `fcn` to open it for editing.

The **Embedded MATLAB Editor** appears.


Embedded MATLAB Editor - Block: call_stats_block1/Embedded MATLAB Function
File Edit Text Debug Tools Window Help
Ready
Ln 1 Col 1

```
function y = fcn(u)
% This block supports an embeddable subset of the MATLAB language.
% See the help menu for details.

y = u;
```

The **Embedded MATLAB Editor** window is titled with the syntax <*model name*>/<*Embedded MATLAB Function block name*> in its header. In this example, the model name is `call_stats_block1`, and the block name is `Embedded MATLAB Function`, the name that appears at the bottom of the Embedded MATLAB Function block in Simulink.

Inside the **Embedded MATLAB Editor** is an edit window for editing the function that specifies the Embedded MATLAB Function block. A function header with the function name `fcn` is at the top of the edit window. The header specifies an argument to the function, `u`, and a return value, `y`.

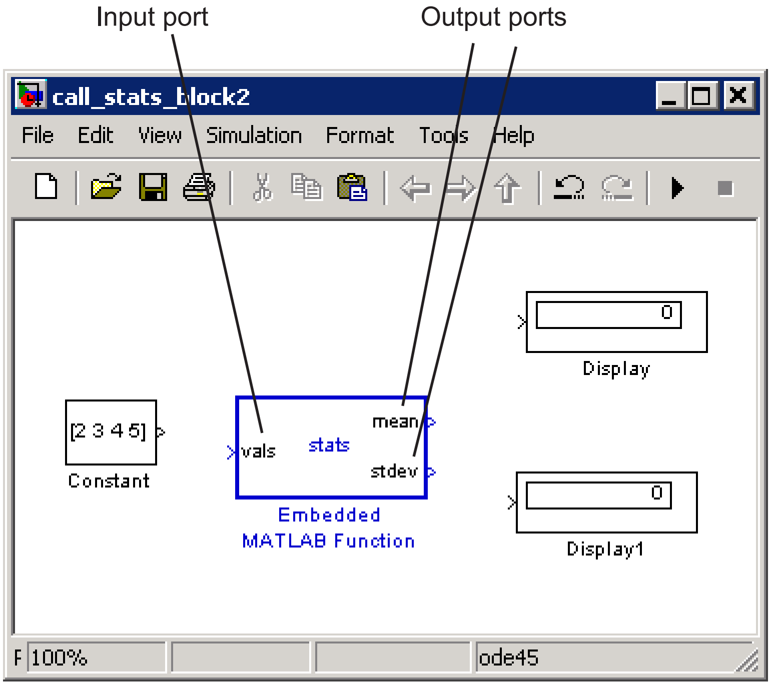
2 Edit the function header line with the return values, function name, and argument as follows:

```
function [mean,stdev] = stats(vals)
```

The Embedded MATLAB function `stats` calculates a statistical mean and standard deviation for the values in the vector `vals`. The function header declares `vals` to be an argument to the `stats` function and `mean` and `stdev` to be return values from the function.

3 In the **Embedded MATLAB Editor**, from the **File** menu, select **Save As Model** and save the model as `call_stats_block2`.

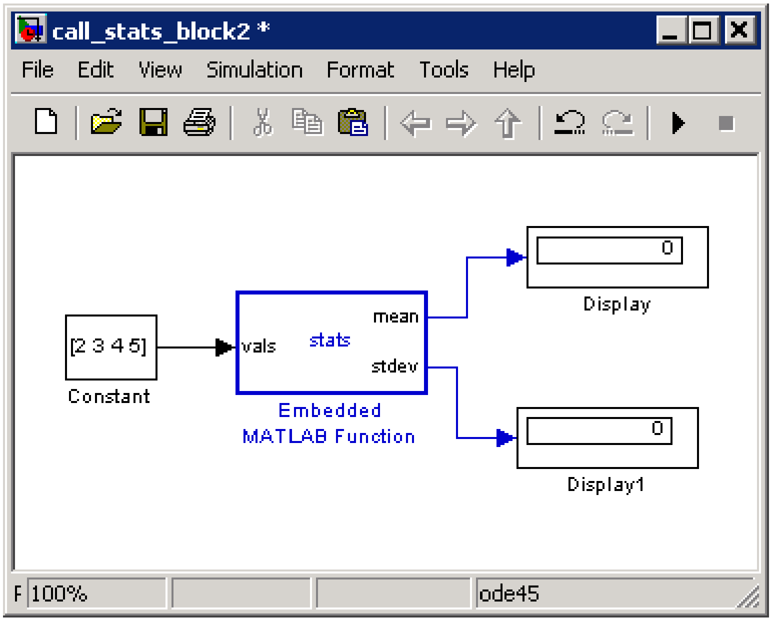
Saving the model updates the Simulink window, which now looks like this:



Changing the function header of the Embedded MATLAB Function block makes the following changes to the Embedded MATLAB Function block in the Simulink model:

- The function name in the middle of the block changes to `stats`.
- The argument `vals` appears as an input port to the block.
- The return values `mean` and `stdev` appear as output ports to the block.

4 In the Simulink window, complete connections to the Embedded MATLAB Function block as shown.



5 In the **Embedded MATLAB Editor**, enter a line space after the function header and replace the default comment line with the following comment lines:

```
% calculates a statistical mean and a standard
% deviation for the values in vals.
```

You specify comments with a leading percent (%) character, just as you do in MATLAB.

6 Enter a line space after the comments and replace the default function line `y = u;` with the following:

```
len = length(vals);
```

The function `length` is an example of a built-in function supported by the runtime function library for Embedded MATLAB Function blocks. This length works just like the MATLAB function length. It returns the vector length of its argument vals. However, when you simulate this model, Simulink generates C code for this function in the simulation application. Callable functions supported for Embedded MATLAB Function blocks are listed in the topic “Embedded MATLAB Run-Time Function Library” in the Embedded MATLAB documentation.

The variable len is a local variable that is automatically typed as a scalar double because the Embedded MATLAB runtime library function, length, returns a scalar of type double. If you want, you can declare len to have a different type and size by changing the way you declare it in the function. See “Creating Local Variables Implicitly” in the Embedded MATLAB documentation for details about implicitly declaring local variables in an Embedded MATLAB Function block.

By default, implicitly declared local variables like len are temporary. They come into existence only when the function is called and cease to exist when the function is exited. To persist implicitly declared variables between function calls, see “Declaring Persistent Variables” in the Embedded MATLAB documentation.

7 Enter the following lines to calculate the value of mean and stdev:

```
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
```

stats stores the mean and standard deviation values for the values in vals in the variable mean and stdev, which are output by port to the Display blocks in the Simulink model. The line that calculates mean calls a subfunction, avg, that has not been defined yet. The line that calculates stdev calls the Embedded MATLAB runtime library functions sqrt and sum.

8 Enter the following line to plot the input values in vals.

```
plot(vals,'-+');
```

This line calls the function plot to plot the input values sent to stats against their vector indices. Because the Embedded MATLAB runtime library has no plot function, the Embedded MATLAB function cannot resolve this call with a subfunction or an Embedded MATLAB runtime function. Instead, it replaces this call with a call to the MATLAB `plot` function in the generated code for the simulation target.

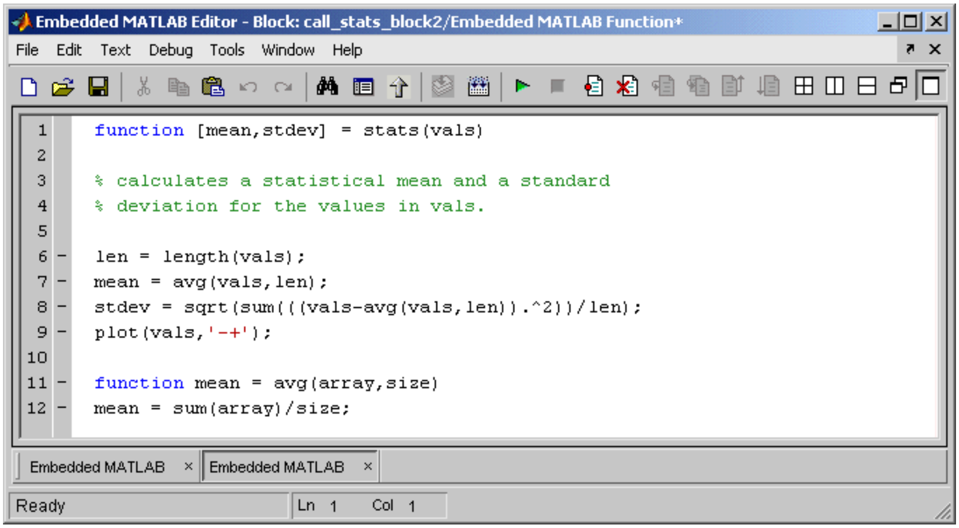
See “Calling MATLAB Functions” in the Embedded MATLAB documentation for more details on using this mechanism to call MATLAB functions from Embedded MATLAB functions.

9 Enter a line space followed by the following lines for the subfunction `avg`, which is called in an earlier line.

```
function mean = avg(array,size)
mean = sum(array)/size;
```

These two lines define the subfunction `avg`. You are free to use subfunctions in Embedded MATLAB function code with single or multiple return values, just as you do in regular MATLAB functions.

The **Embedded MATLAB Editor** should now have the following appearance:



```
function [mean,stdev] = stats(vals)

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

10 Save the model again as `call_stats_block2`.

Checking the Function for Errors

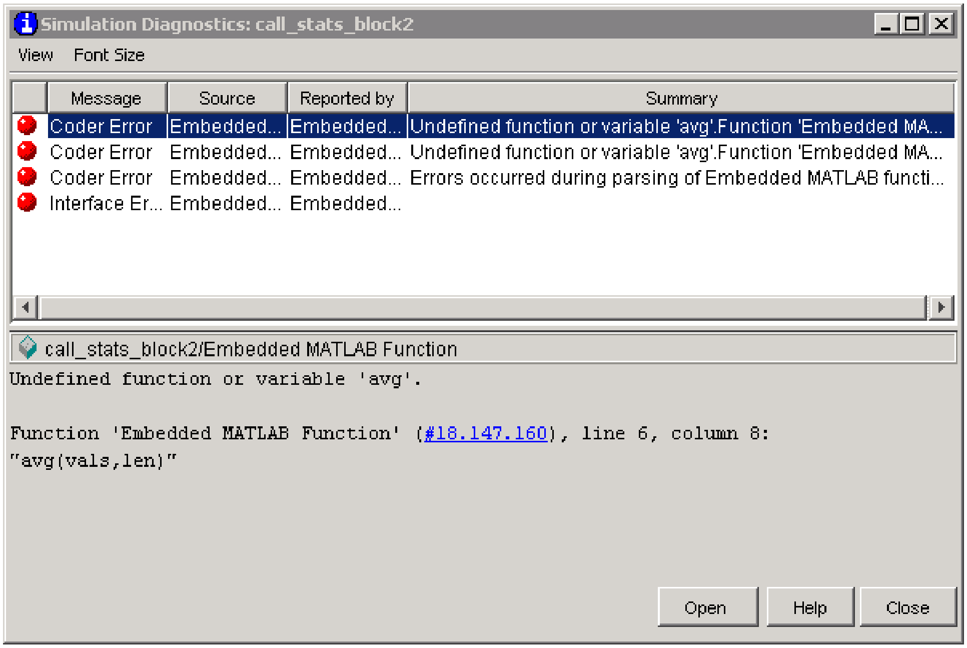
Once you finish specifying an Embedded MATLAB Function block in its Simulink model, use the built-in diagnostics of Embedded MATLAB Function blocks to test for syntax errors with the following procedure:

1. If not already open, open the `call_stats_block2` model that you save at the end of "Programming the Embedded MATLAB Function" on page 17-9, and double-click its Embedded MATLAB Function block `stats` to open it for editing.

2. In the **Embedded MATLAB Editor**, click the **Build** icon to compile and build the example Simulink model:

   If errors are found, the **Diagnostics Manager** window lists them. Otherwise, nothing happens.

3. For example, change the subfunction `avg` to a fictitious subfunction `aug` and then compile to see the following messages in the **Diagnostics Manager** window:



	Message	Source	Reported by	Summary
●	Coder Error	Embedded...	Embedded...	Undefined function or variable 'avg'.Function 'Embedded MA...
●	Coder Error	Embedded...	Embedded...	Undefined function or variable 'avg'.Function 'Embedded MA...
●	Coder Error	Embedded...	Embedded...	Errors occurred during parsing of Embedded MATLAB functi...
●	Interface Er...	Embedded...	Embedded...	

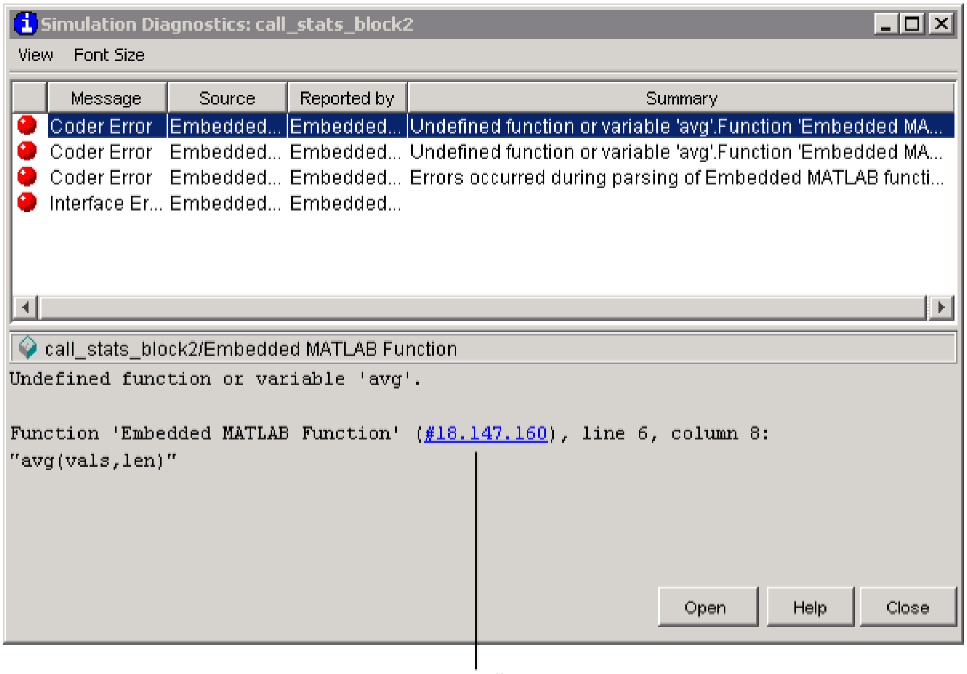
call_stats_block2/Embedded MATLAB Function

```
Undefined function or variable 'avg'.

Function 'Embedded MATLAB Function' (#18.147.160), line 6, column 8:
"avg(vals,len)"
```

Each detected error appears with a red button.

4 Click the first error line to display its message.

5 In the diagnostic message for the selected error, click the blue link to find the offending code:



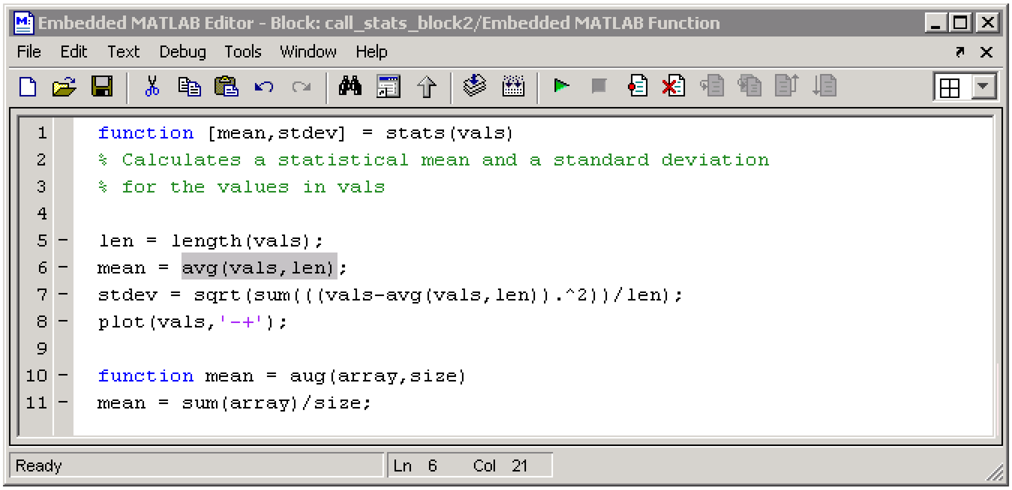
	Message	Source	Reported by	Summary
●	Coder Error	Embedded...	Embedded...	Undefined function or variable 'avg'.Function 'Embedded MA...
●	Coder Error	Embedded...	Embedded...	Undefined function or variable 'avg'.Function 'Embedded MA...
●	Coder Error	Embedded...	Embedded...	Errors occurred during parsing of Embedded MATLAB functi...
●	Interface Er...	Embedded...	Embedded...	

call_stats_block2/Embedded MATLAB Function

```
Undefined function or variable 'avg'.

Function 'Embedded MATLAB Function' (#18.147.160), line 6, column 8:
"avg(vals,len)"
```

Click to display the offending code
in the Embedded MATLAB Editor

The offending line appears highlighted in the **Embedded MATLAB Editor**:


Embedded MATLAB Editor - Block: call_stats_block2/Embedded MATLAB Function
File Edit Text Debug Tools Window Help
Ready
Ln 6 Col 21

```
function [mean,stdev] = stats(vals)
% Calculates a statistical mean and a standard deviation
% for the values in vals

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = aug(array,size)
mean = sum(array)/size;
```

6 Correct the error and recompile.

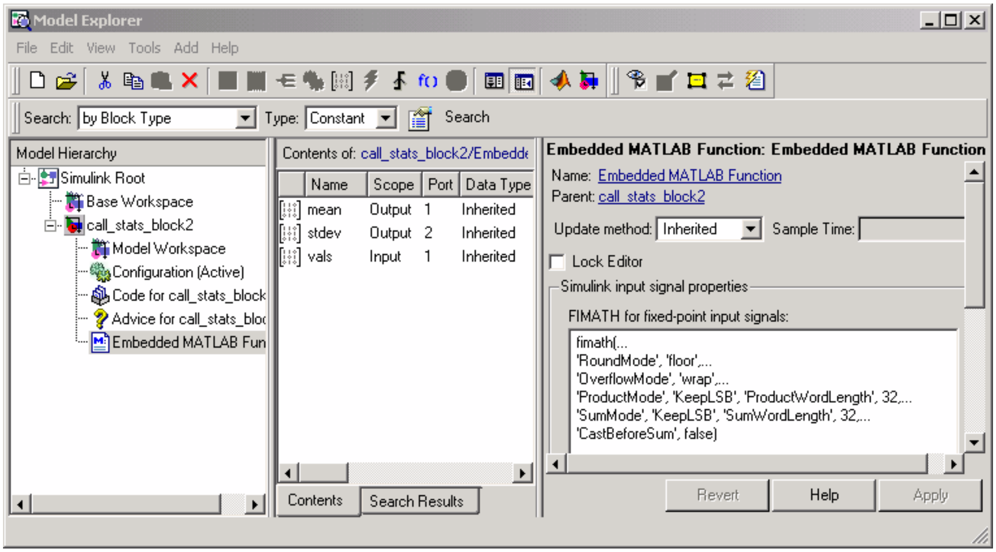
Defining Inputs and Outputs

In the `stats` function header for the Embedded MATLAB Function block you define in "Programming the Embedded MATLAB Function" on page 17-9, the function argument `vals` is an input and `mean` and `stdev` are outputs. By default, function inputs and outputs inherit their data type and size from the Simulink signals attached to their ports. In this topic, you examine input and output data for the Embedded MATLAB Function block to verify that it inherits the correct type and size.

1 If not already open, open the `call_stats_block2` model that you save at the end of "Programming the Embedded MATLAB Function" on page 17-9, and double-click its Embedded MATLAB Function block `stats` to open it for editing.

2 In the **Embedded MATLAB Editor**, select **Tools > Model Explorer** to open the **Model Explorer**.

The **Model Explorer** window opens:



Search: by Block Type Type: Constant

Simulink Root
- Base Workspace
- call_stats_block2
  - Model Workspace
  - Configuration (Active)
  - Code for call_stats_block
  - Advice for call_stats_bloc
  - Embedded MATLAB Fun

Contents of: call_stats_block2/Embedde

Name	Scope	Port	Data Type
mean	Output	1	Inherited
stdev	Output	2	Inherited
vals	Input	1	Inherited

Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function
Parent: call_stats_block2

Update method: Inherited Sample Time:

□ Lock Editor

Simulink input signal properties

FIMATH for fixed-point input signals:

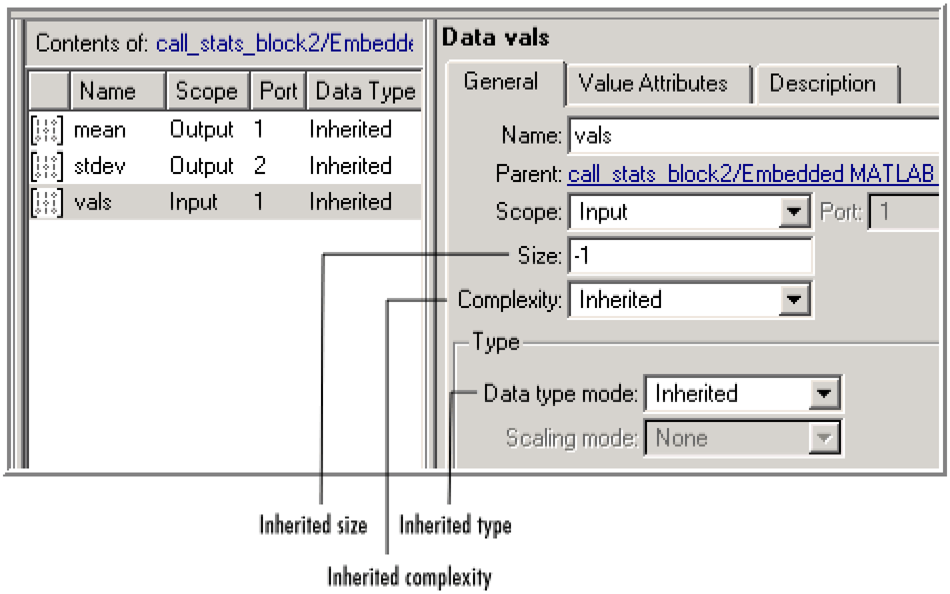
```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'KeepLSB', 'ProductWordLength', 32,...
'SumMode', 'KeepLSB', 'SumWordLength', 32,...
'CastBeforeSum', false)
```

You can use the **Model Explorer** to define arguments for Embedded MATLAB Function blocks. Notice that the Embedded MATLAB Function block `Embedded MATLAB` is highlighted in the left **Model Hierarchy** pane.

The **Contents** pane displays the argument `vals` and the return values `mean` and `stdev` that you have already created for the Embedded MATLAB Function block. Notice that `vals` is assigned a **Scope** of `Input`, which is short for **Input from Simulink**. `mean` and `stdev` are assigned the **Scope** of `Output`, which is short for **Output to Simulink**.

You can also use the **Ports and Data Manager** to define arguments for Embedded MATLAB Function blocks (see “Ports and Data Manager” on page 17-42).

3 In the **Contents** pane of the **Model Explorer** window, click anywhere in the row for `vals` to highlight it:



Contents of: call_stats_block2/Embedde

Name	Scope	Port	Data Type
mean	Output	1	Inherited
stdev	Output	2	Inherited
vals	Input	1	Inherited

Data vals

General | Value Attributes | Description

Name: vals

Parent: call_stats_block2/Embedded MATLAB

Scope: Input Port: 1

Size: -1

Complexity: Inherited

Type

Data type mode: Inherited

Scaling mode: None

The right pane displays the **Data** properties dialog box for `vals`. By default, the type, size, and complexity of input and output arguments are inherited from the Simulink signals attached to each input or output port. Inheritance is specified by setting **Type** and **Complexity** to `Inherited`, and **Size** to `-1`:

The actual inherited values for size and type are set during compilation of the model, and are reported in the **Compiled Type** and **Compiled Size** columns of the **Contents** pane. You compile and build the model by clicking the **Build** icon:

You can specify the type of an input or output argument directly by selecting a type in the **Type** field of the **Data** properties dialog box, for example, `double`. You can also specify the size of an input or output argument directly by entering an expression in the **Size** field of the **Data** properties dialog box for the argument. For example, you can enter `[2 3]` in the **Size** field to size `vals` as a 2-by-3 matrix. See “Typing Function Arguments” on page 17-71 and “Sizing Function Arguments” on page 17-81 for more information on the expressions that you can enter for type and size.

Note The default first index for any arrays that you add to an Embedded MATLAB Function block function is `1`, just as it would be in MATLAB.

Debugging an Embedded MATLAB Function

In “Creating an Example Embedded MATLAB Function” on page 17-7, you create and specify an example Simulink model with an Embedded MATLAB Function block. You use this block to specify an Embedded MATLAB function `stats` that calculates the mean and standard deviation for a set of input values. In this section, you debug `stats` in the example model.

Use the following topics to learn how to debug an Embedded MATLAB function in Simulink:

- “Debugging the Function in Simulation” on page 17-22 — Executes the model in simulation and tests the Embedded MATLAB function `stats`.
- “Watching Function Variables During Simulation” on page 17-30 — Describes tools that you can use to view the values of Embedded MATLAB variables during simulation.
- “How Exiting Debug Mode Affects Simulation” on page 17-33 — Describes how exiting debug mode affects simulation of the Embedded MATLAB function

Debugging the Function in Simulation

You can debug your Embedded MATLAB Function block just like you can debug a function in MATLAB. In simulation, you test your Embedded MATLAB functions for runtime errors with tools similar to the MATLAB debugging tools.

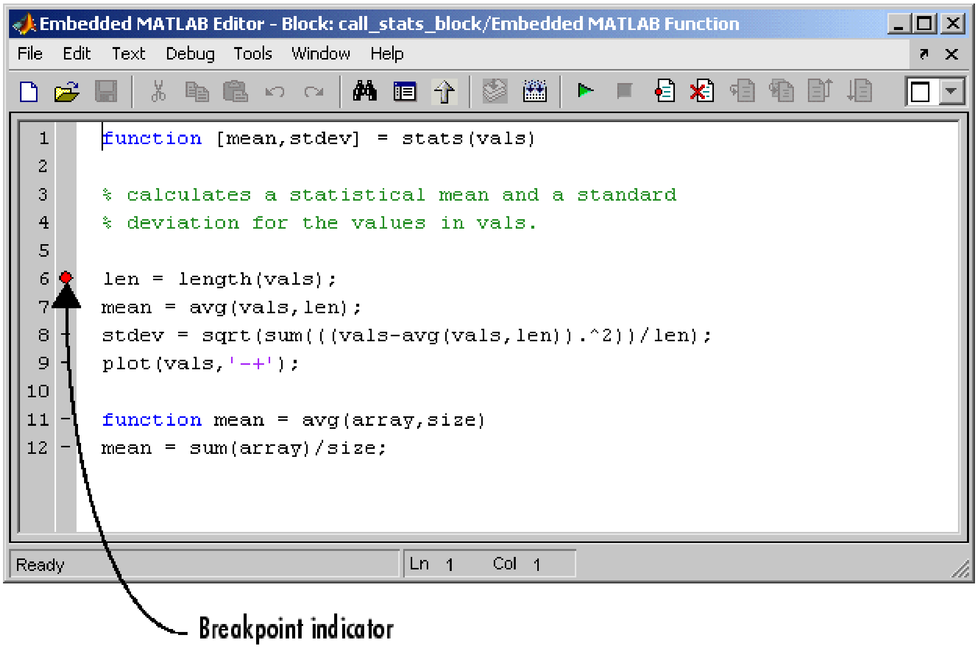
When you start simulation of your model, Simulink checks to see if the Embedded MATLAB Function block has been built since creation, or since a change has been made to the block. If not, it performs the build described in “Checking the Function for Errors” on page 17-15. If no diagnostic errors are found, Simulink begins the simulation of your model.

Use the following procedure to debug the `stats` Embedded MATLAB function during simulation of the model:

1. If not already open, open the `call_stats_block2` model that you save at the end of “Programming the Embedded MATLAB Function” on page 17-9,

and double-click its Embedded MATLAB Function block `stats` to open it for editing in the **Embedded MATLAB Editor**.

2 In the **Embedded MATLAB Editor**, in the left margin of line 6, click the dash (-) character.


Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File
Edit
Text
Debug
Tools
Window
Help
Ready
Ln 1
Col 1
Breakpoint indicator

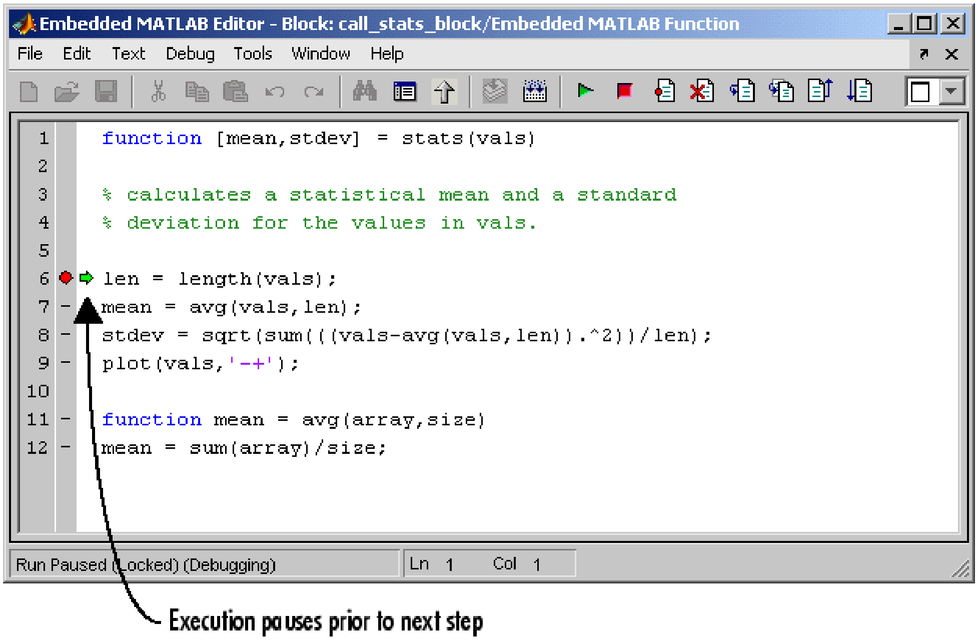
```
 1   function [mean,stdev] = stats(vals)
 2
 3   % calculates a statistical mean and a standard
 4   % deviation for the values in vals.
 5
 6   len = length(vals);
 7   mean = avg(vals,len);
 8   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
 9   plot(vals,'-+');
10
11   function mean = avg(array,size)
12   mean = sum(array)/size;
```

A small red ball appears in the margin of line 6, indicating that you have set a breakpoint. You can also use the **Set/Clear Breakpoint** icon to insert the breakpoint on the line where the cursor is positioned.

3 Click the **Start Simulation** icon to begin simulating the model:

▶

If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. This is indicated by a small green arrow in the left margin, as shown.



Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File Edit Text Debug Tools Window Help

```
1     function [mean,stdev] = stats(vals)
2
3     % calculates a statistical mean and a standard
4     % deviation for the values in vals.
5
6 ● ⇨ len = length(vals);
7 -   mean = avg(vals,len);
8 -   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
9 -   plot(vals,'-+');
10
11 -  function mean = avg(array,size)
12 -  mean = sum(array)/size;
```

Run Paused (Locked) (Debugging)
Ln 1 Col 1

Execution pauses prior to next step

4 In the **Embedded MATLAB Editor** window, click the **Step** icon to advance execution:

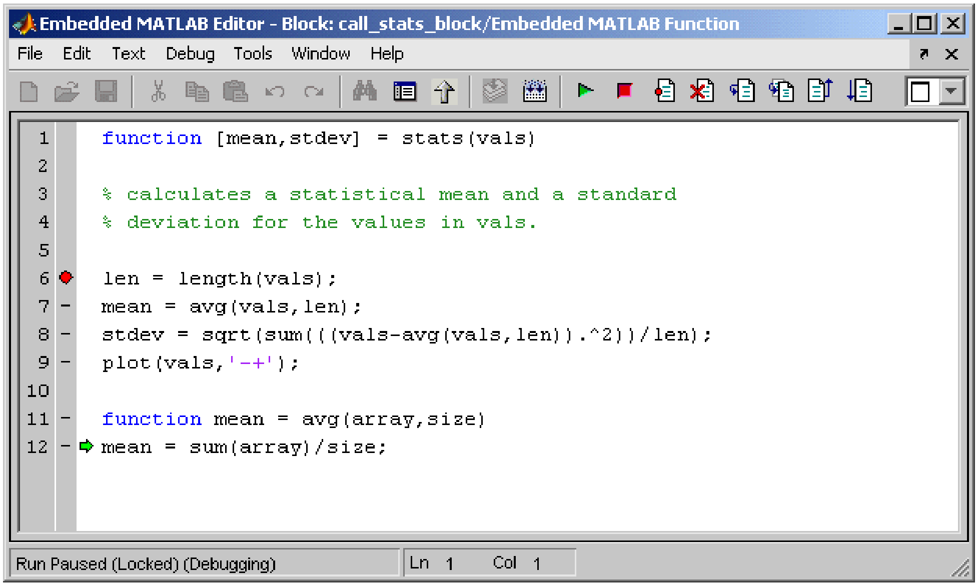
The execution arrow advances one line to line 7 of `stats`.

You can also step execution by entering `dbstep` at the Command Line Debugger. See “Watching with the Command Line Debugger” on page 17-31 for a description of the Command Line Debugger in MATLAB.

Notice that line 7 calls the subfunction avg. If you click **Step** here, execution advances to line 8, past the execution of the subfunction avg. To track execution of the lines in the subfunction avg, you need to click the **Step In** icon.

5 Click the **Step In** icon:

Execution advances to enter the subfunction avg:


Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File Edit Text Debug Tools Window Help

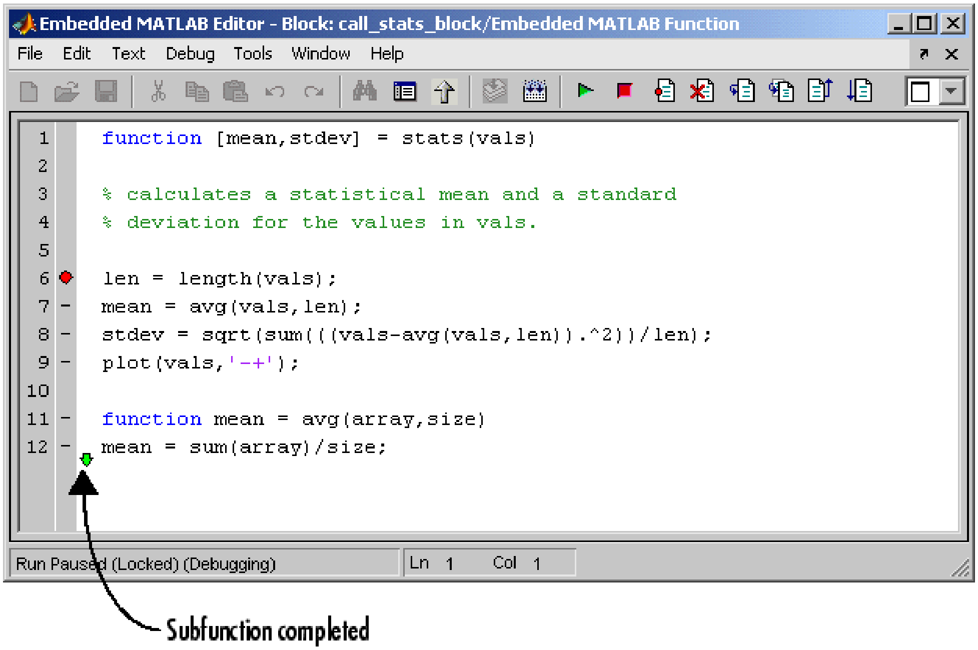
```
 1     function [mean,stdev] = stats(vals)
 2
 3     % calculates a statistical mean and a standard
 4     % deviation for the values in vals.
 5
 6 ●   len = length(vals);
 7 -   mean = avg(vals,len);
 8 -   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
 9 -   plot(vals,'-+');
10
11 -   function mean = avg(array,size)
12 - ⇨ mean = sum(array)/size;
```

Run Paused (Locked) (Debugging) Ln 1 Col 1

Once you are in a subfunction, you can use the **Step** or **Step In** icon to advance execution. If the subfunction calls another subfunction, use the **Step In** icon to enter it. If you want to execute the remaining lines of the subfunction, click the **Step Out** icon:

6 Click the **Step** icon to execute the only line in the subfunction avg.

The subfunction avg finishes its execution, and you see a green arrow pointing down under its last line as shown.

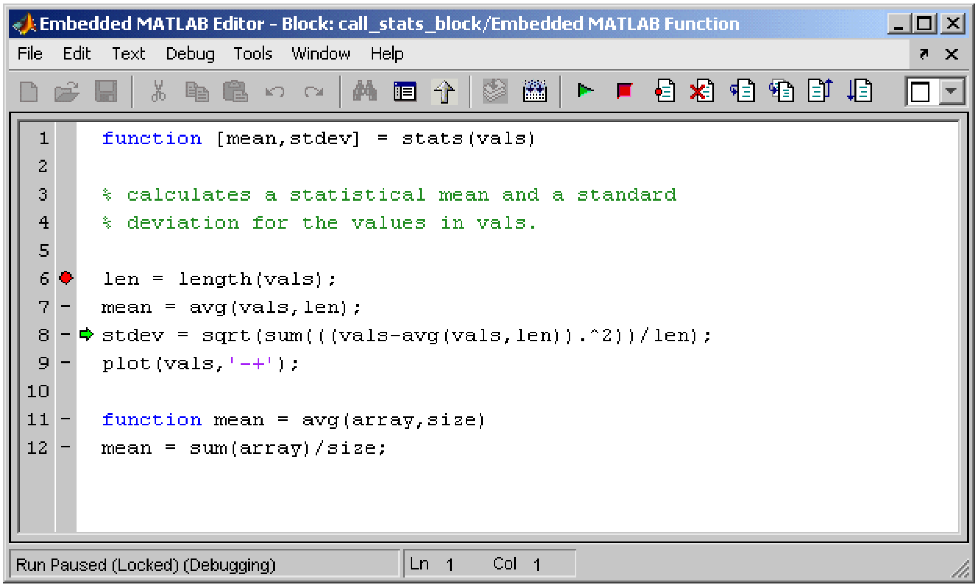

Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File Edit Text Debug Tools Window Help

```
function [mean,stdev] = stats(vals)

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

Run Paused (Locked) (Debugging)
Ln 1 Col 1

Subfunction completed

7 Click the **Step** icon to return to the function `stats`.


Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File
Edit
Text
Debug
Tools
Window
Help
Ln 1
Col 1

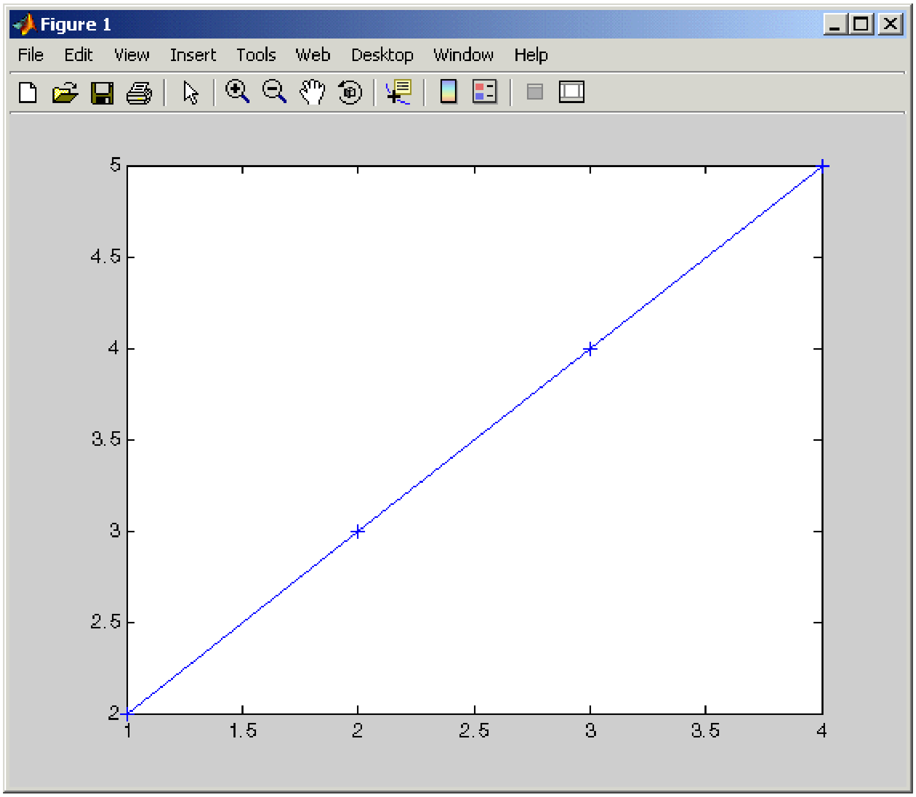
```
 1   function [mean,stdev] = stats(vals)
 2
 3   % calculates a statistical mean and a standard
 4   % deviation for the values in vals.
 5
 6   len = length(vals);
 7   mean = avg(vals,len);
 8   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
 9   plot(vals,'-+');
10
11   function mean = avg(array,size)
12   mean = sum(array)/size;
```

Run Paused (Locked) (Debugging)

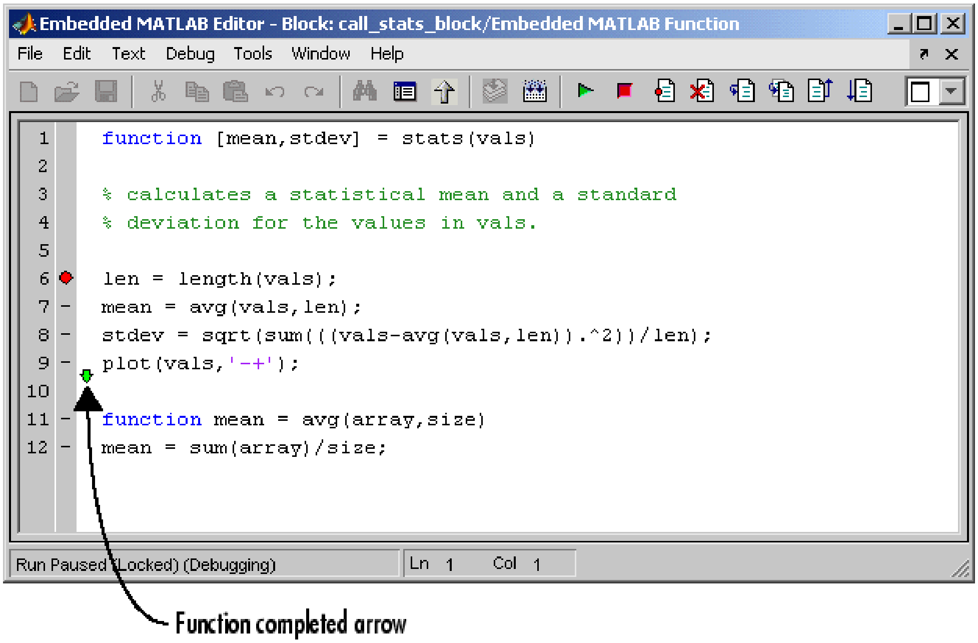
Execution advances to the line after to the call to the subfunction `avg`, line 8.

8 Click **Step** twice to execute line 8 and the `plot` function in line 9.

The plot function executes in MATLAB, and you see the following plot.



In the **Embedded MATLAB Editor**, a green arrow points down under line 9, indicating the completion of the function `stats`.


Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File Edit Text Debug Tools Window Help
Run Paused (Locked) (Debugging)
Ln 1 Col 1

```
1   function [mean,stdev] = stats(vals)
2
3   % calculates a statistical mean and a standard
4   % deviation for the values in vals.
5
6   len = length(vals);
7   mean = avg(vals,len);
8   stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
9   plot(vals,'-+');
10
11  function mean = avg(array,size)
12  mean = sum(array)/size;
```

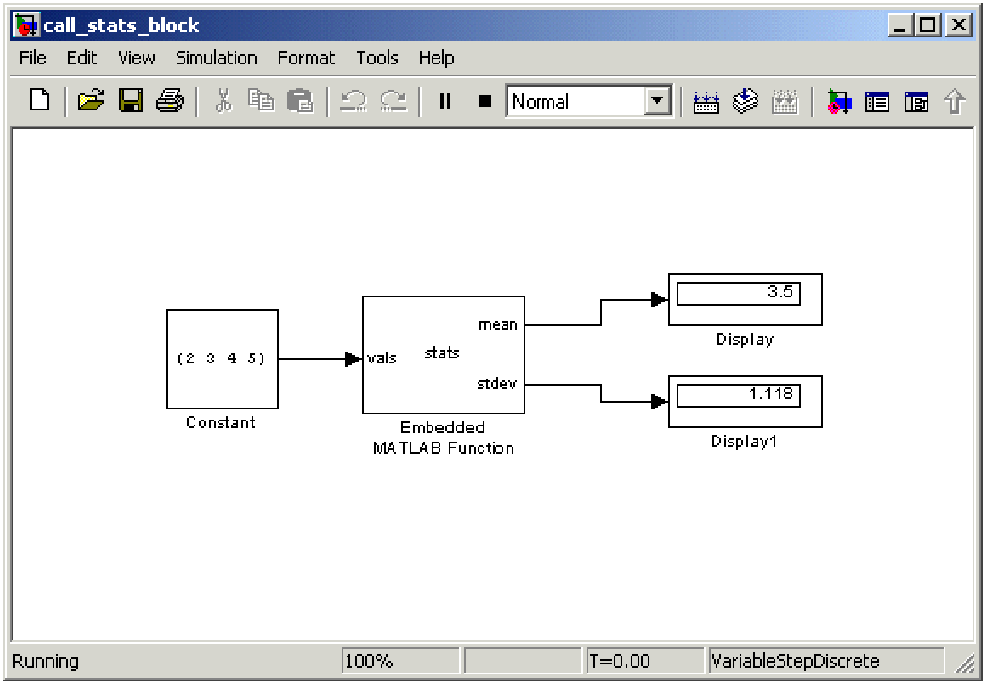
Function completed arrow

9 Click the **Continue Debugging** icon to continue execution of the model.

At any point in a function, you can advance through the execution of the remaining lines of the function with the **Continue Debugging** icon. If you are at the end of the function, clicking the **Step** icon accomplishes the same thing.

You can also continue execution by entering `dbcont` at the Command Line Debugger. See “Watching with the Command Line Debugger” on page 17-31 for a description of the Command Line Debugger in MATLAB.

In the Simulink window, the computed values of `mean` and `stdev` now appear in the Display blocks.



10 In the **Embedded MATLAB Editor**, click the **Exit Debug Mode** icon to stop simulation:

For more information, see "How Exiting Debug Mode Affects Simulation" on page 17-33.

Watching Function Variables During Simulation

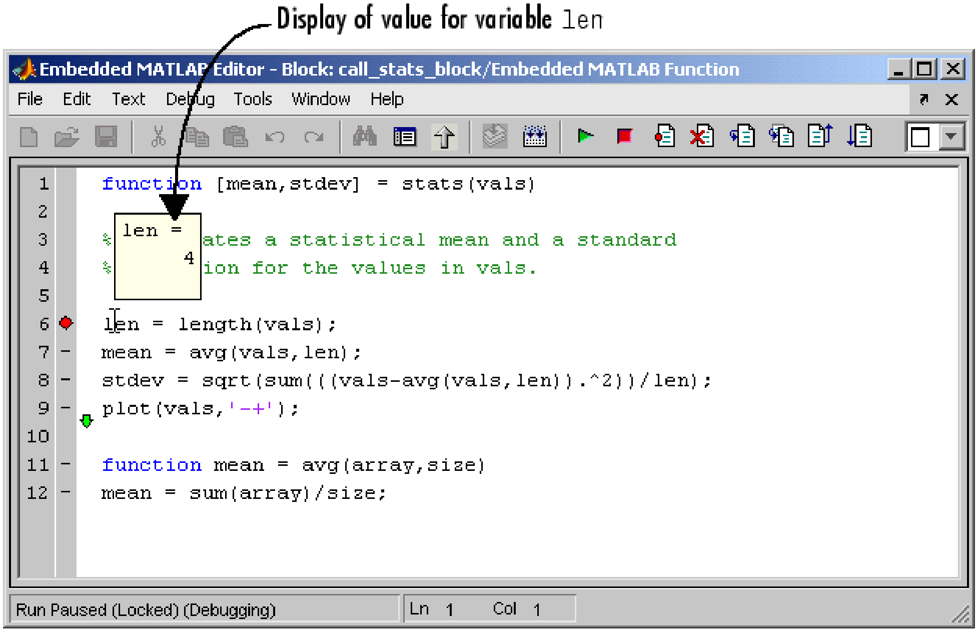
While you are simulating the function of an Embedded MATLAB Function block, you can use several tools to keep track of variable values in the function. These tools are described in the topics that follow.

Watching with the Interactive Display

To display the value of a variable in the function of an Embedded MATLAB Function block during simulation, in the **Embedded MATLAB Editor**, place the mouse cursor over the variable text and observe the pop-up display.

For example, to watch the variable `len` during simulation, place the mouse cursor over the text `len` in line 6 for at least a second. The value of `len` appears adjacent to the cursor, as shown:

Display of value for variable `len`



Embedded MATLAB Editor - Block: call_stats_block/Embedded MATLAB Function
File
Edit
Text
Debug
Tools
Window
Help

```
 1     function [mean,stdev] = stats(vals)
 2
 3     %           ates a statistical mean and a standard
 4     %           ion for the values in vals.
 5
 6     len = length(vals);
 7  -  mean = avg(vals,len);
 8  -  stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
 9  -  plot(vals,'-+');
10
11  -  function mean = avg(array,size)
12  -  mean = sum(array)/size;
```

```
len =
     4
```

Run Paused (Locked) (Debugging)
Ln 1
Col 1

You can display the value for any variable in the Embedded MATLAB function in this way, no matter where it appears in the function.

Watching with the Command Line Debugger

You can report the values for an Embedded MATLAB function variable with the Command Line Debugger utility in the MATLAB window during simulation. When you reach a breakpoint, press **Enter** in the MATLAB window and the Command Line Debugger prompt, `debug>>`, appears. At this prompt, you can see the value of a variable defined for the Embedded MATLAB Function block by entering its name:

```
debug>> stdev

  1.1180

debug>>
```

The Command Line Debugger also provides the following commands during simulation:

Command	Description
dbstep	Advance to next program step after a breakpoint is encountered.
dbcont	Continue execution to next breakpoint.
dbquit	Stop simulation of the model. Press **Enter** after this command to return the MATLAB prompt.
help	Display help for command line debugging.
print x	Display the value of the variable x. If x is a vector or matrix, you can also index into x. For example, x(1,2).
save	Saves all variables to the specified file. Follows the syntax of the MATLAB save command. To retrieve variables to the MATLAB base workspace, use load command after simulation has been ended.
whos	Display the size and class (type) of all variables in the scope of the halted Embedded MATLAB Function block.

You can issue any other MATLAB command at the debug>> prompt, but the results are executed in the workspace of the Embedded MATLAB Function block. To issue a command in the MATLAB base workspace at the debug>> prompt, use the evalin command with the first argument 'base' followed by the second argument command string, for example, evalin('base','whos'). To return to the MATLAB base workspace, use the dbquit command.

Watching with MATLAB

You can display the execution result of an Embedded MATLAB function line by omitting the terminating semicolon. If you do, execution results for the line are echoed to the MATLAB window during simulation.

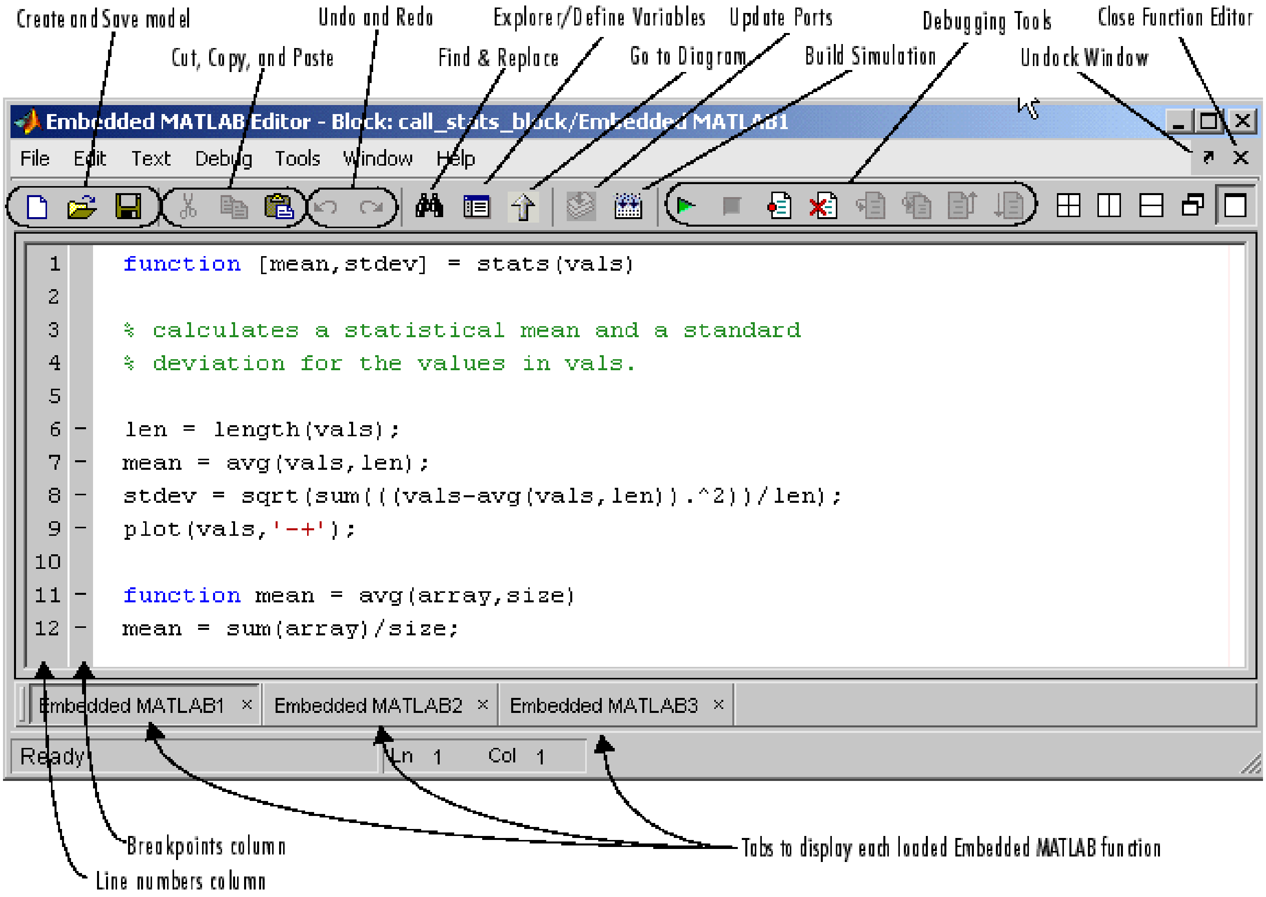
How Exiting Debug Mode Affects Simulation

The behavior of the **Exit Debug Mode** option depends on the run-time context, as follows:

If You Exit Debug Mode	What Happens
When the Embedded MATLAB function is stopped at a breakpoint	Simulation stops immediately
From the **Embedded MATLAB Editor**	Simulation stops when the Embedded MATLAB Function block finishes executing
From the Simulink Editor	Simulation stops immediately
By typing **Ctrl-C** when the Simulink Editor window has focus	Simulation stops immediately

The Embedded MATLAB Function Editor

You edit an Embedded MATLAB function to specify its function header and body. When you open an unspecified Embedded MATLAB function for editing, it has the following default appearance in the **Embedded MATLAB Editor**:



```
function [mean,stdev] = stats(vals)

% calculates a statistical mean and a standard
% deviation for the values in vals.

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

This section provides the following topics to describe tools for editing Embedded MATLAB functions in the **Embedded MATLAB Editor**:

Changing the Embedded MATLAB Editor

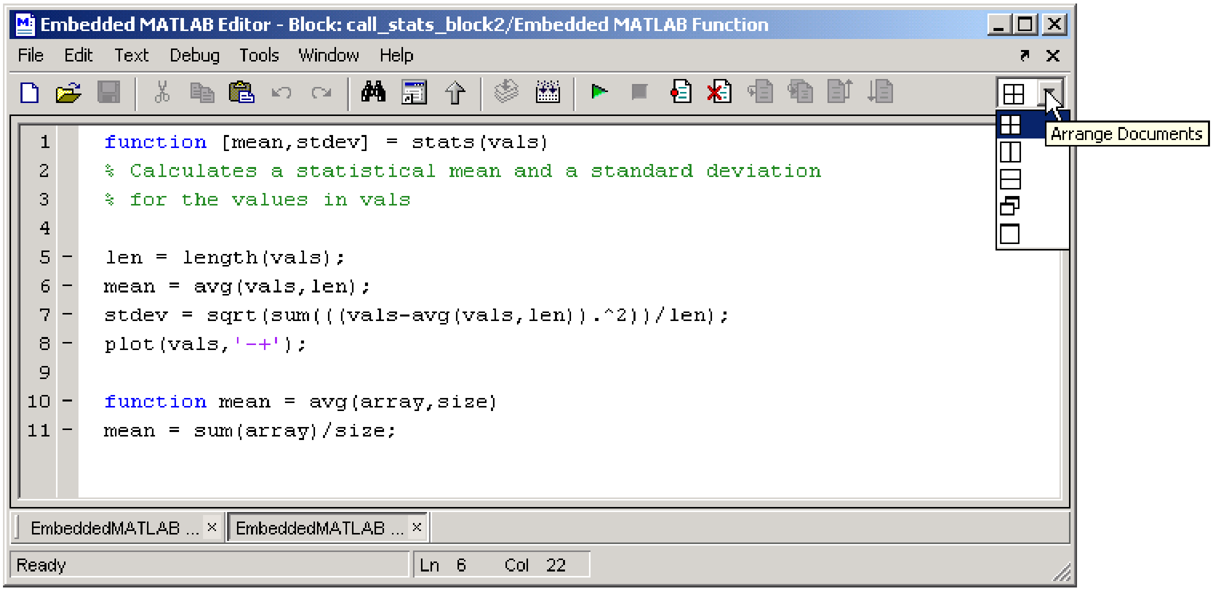
Use the toolbar icons described in the following topics to customize the appearance of the **Embedded MATLAB Editor**.

Displaying Embedded MATLAB Function Windows

By default, if you have more than one Embedded MATLAB function loaded in the **Embedded MATLAB Editor**, only the most recently loaded is displayed. Editing windows for previously loaded functions are accessed individually with tabs in the document bar. You can display the editing windows for all loaded functions simultaneously by selecting one of the following options from the **Window** menu:

Menu Option	Description
⊞ **Tile**	Tiles all loaded Embedded MATLAB windows into an adjustable matrix of windows. When you select the option, an array of squares representing the tiled windows is available as a submenu. Select an appropriate array of windows.
◫ **Left/Right Tile**	Displays the selected window and the next most recently loaded Embedded MATLAB function at full height, side by side.
⊟ **Top/Bottom Tile**	Displays the selected window and the next most recently loaded Embedded MATLAB function at full width, top to bottom.
❐ **Float**	Displays the loaded Embedded MATLAB functions in separate cascading and overlapping windows of the same size.
□ **Maximize**	Displays the Embedded MATLAB function in current focus at the full width and height of the editor. This is the default setting.

These options are also available from the **Arrange Documents** drop-down menu on the right side of the toolbar:


Embedded MATLAB Editor - Block: call_stats_block2/Embedded MATLAB Function
File Edit Text Debug Tools Window Help
Arrange Documents
EmbeddedMATLAB ...
EmbeddedMATLAB ...
Ready
Ln 6 Col 22

```
function [mean,stdev] = stats(vals)
% Calculates a statistical mean and a standard deviation
% for the values in vals

len = length(vals);
mean = avg(vals,len);
stdev = sqrt(sum(((vals-avg(vals,len)).^2))/len);
plot(vals,'-+');

function mean = avg(array,size)
mean = sum(array)/size;
```

Creating Editors for Embedded MATLAB Functions

When you open more than one Embedded MATLAB function into the **Embedded MATLAB Editor** you can create separate editors for each function by undocking the function from the main editor.

To create a new editor window for the Embedded MATLAB function that is in focus in the **Embedded MATLAB Editor**, select the **Undock Embedded MATLAB Function** icon on the right side of the main editor's menu bar:

↗

To reverse the process, select the **Dock Embedded MATLAB Function** icon on the right side of the editor's menu bar:

↘

Moving the Document Bar

When you edit an Embedded MATLAB function, it is displayed as a window in the **Embedded MATLAB Editor**. You can open more than one Embedded MATLAB function in the editor. For each function that you open, a tab is added to a document bar at the bottom of the **Embedded MATLAB Editor**. If you want to edit an Embedded MATLAB function not in focus, click its tab.

To change the location of the document bar or hide it altogether, follow these steps:

1 Right-click in the document bar and select **Bar Position** from the drop-down menu.

2 From the submenu, select **Top**, **Bottom**, **Right**, **Left**, or **Hide**.

Sorting Functions in the Editor

You can sort functions in the **Embedded MATLAB Editor** by right-clicking in the document bar and toggling the **Alphabetize** option on or off.

Toggling the Toolbar Display

By default, the **Embedded MATLAB Editor** has a toolbar with shortcuts to tools that you can access from the menus in the menu bar. You can toggle the toolbar display by following these steps:

1 Right-click in the toolbar and select **Embedded MATLAB Editor Toolbar**.

   The toolbar disappears.

2 Right-click in the menu bar and select **Embedded MATLAB Editor Toolbar**.

   The **Toolbar** reappears.

Setting Preferences

You can choose preferences for the **Embedded MATLAB Editor**, such as font size, tab size, and so on, as follows:

1 From the **File** menu, select **Preferences**.

   The MATLAB **Preferences** dialog box appears.

2 Change preferences in pages accessed only through the following nodes:

   - **Fonts**
   - **Colors**
   - **Display** (under **Editor/Debugger**)
   - **Keyboard & Indenting** (under **Editor/Debugger**)

Note The **Embedded MATLAB Editor** is a derivation of the MATLAB editor you use to edit M-files in MATLAB. The preference changes that you specify are made to the MATLAB editor.

Editing the Embedded MATLAB Function

Use the tools in the following topics to edit an Embedded MATLAB function in the **Embedded MATLAB Editor**:

Undoing and Redoing Operations

Tool Button	Description
↶ **Undo**	Undo the effects of the preceding operation. Alternatively, from the **Edit** menu, select **Undo**.
↷ **Redo**	Redo the effects of the most recently undone operation. Alternatively, from the **Edit** menu, select **Redo**.

Comment and Uncomment Embedded MATLAB Function Lines

You can comment text or uncomment commented text as follows:

- To turn selected function text lines into commented text lines, from the **Text** menu, select **Comment**.
- To turn selected comment text lines into function text lines, from the **Text** menu, select **Uncomment**.

Any text selected on a line, or the presence of the text cursor, selects the line.

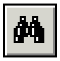
Going to a Specified Line of the Embedded MATLAB Function

To place the text cursor at the beginning of a specified line, from the **Edit** menu, select **Go To Line**. In the resulting dialog box, enter the line number and click **OK**.

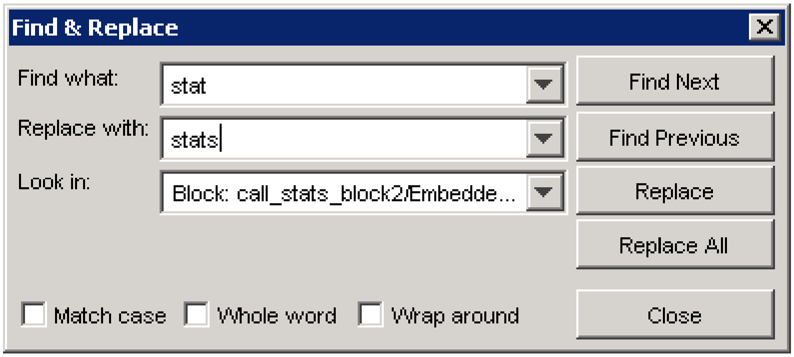
Searching for and Replacing Text in Embedded MATLAB Functions

You can use the Find & Replace icon in the toolbar to search and replace text in the **Embedded MATLAB Editor** as follows.

1 Click the **Find & Replace** icon:



The **Find & Replace** dialog box appears.



Find & Replace

Find what: stat

Replace with: stats

Look in: Block: call_stats_block2/Embedde...

□ Match case □ Whole word □ Wrap around

By default, the **Look in** field is set to search the current Embedded MATLAB function, but you can select from any Embedded MATLAB functions that you have open for editing.

2. Enter the search string in the **Find what** field.

3. Modify the text you want to search for by checking any or all of the following:

   - **Match case** — The text must match the case of the search string.
   - **Whole word** — The text must be a whole word and not part of a larger word.
   - **Wrap around** — Continue searching after reaching the bottom of the editor. Otherwise, stop searching.

4. Enter the replacement string in the **Replace with** field.

5. Click the **Find Next** or **Find Previous** button to find a single occurrence of the search string.

   If the text is present in the **Embedded MATLAB Editor**, it is highlighted with a gray background.

6. Click the **Replace** button to replace the highlighted text in the editor with the replacement string.

7. Click the **Replace All** button to replace every occurrence of the search string with the replacement string.

Defining Embedded MATLAB Function Arguments

Once you edit the Embedded MATLAB function, you can set the size, type, or source of an input or output argument using any of the following tools:

Tool Button	Description
Edit Data/Ports	Opens the **Ports and Data Manager** dialog to add or modify arguments for the current Embedded MATLAB function block (see "Ports and Data Manager" on page 17-42). You can also open this dialog by selecting **Edit Data/Ports** from the **Tools** menu. To define and modify input and output arguments for any Embedded MATLAB Function block in the model hierarchy, use the **Model Explorer**, which you can open from the **Tools** menu.
⇧ **Goto Diagram**	Displays the Embedded MATLAB function in its native diagram without closing the **Embedded MATLAB Editor**.
Update Ports	Updates the ports of the Embedded MATLAB Function block with the latest changes made to the function argument and return values without closing the **Embedded MATLAB Editor**.

See "Defining Inputs and Outputs" on page 17-18 for an example of defining an input argument for an Embedded MATLAB Function block.

Ports and Data Manager

The **Ports and Data Manager** provides a convenient method for defining objects and modifying their properties for an Embedded MATLAB Function block that is open and has focus. You can open the **Ports and Data Manager** directly from the Embedded MATLAB Editor for the block of interest.

The Ports and Data Manager provides the same data definition capabilities as the Model Explorer, but restricted to individual Embedded MATLAB Function blocks. To modify objects and properties for blocks across the Simulink model hierarchy, use the Model Explorer, as described in "The Model Explorer" on page 10-2.

The **Ports and Data Manager** lets you add the following objects to an Embedded MATLAB Function block:

Element	Tool	Description
Data argument	[⁞⁞]	Data arguments are used to communicate between the Embedded MATLAB function and the Simulink environment. To learn how to add data arguments and modify their properties, see “Adding Data to an Embedded MATLAB Function Block” on page 17-51.
Input trigger	↑	An *input trigger* causes an Embedded MATLAB Function block to execute when a Simulink control signal changes, through a Simulink block that outputs function-call events, or through an S-function. To learn how to add input triggers and modify their properties, see “Adding Input Triggers to an Embedded MATLAB Function Block” on page 17-62.
Function call output	f()	A *function call output* creates an output port on an Embedded MATLAB Function block, and a function that can be called from the block’s Embedded MATLAB script. When the function is called, it triggers the subsystem attached to the output port. For more information, see “Function-Call Subsystems” in the online Simulink reference. To learn how to add function call outputs and modify their properties, see “Adding Function Call Outputs to an Embedded MATLAB Function Block” on page 17-65.

The Ports and Data Manager Dialog

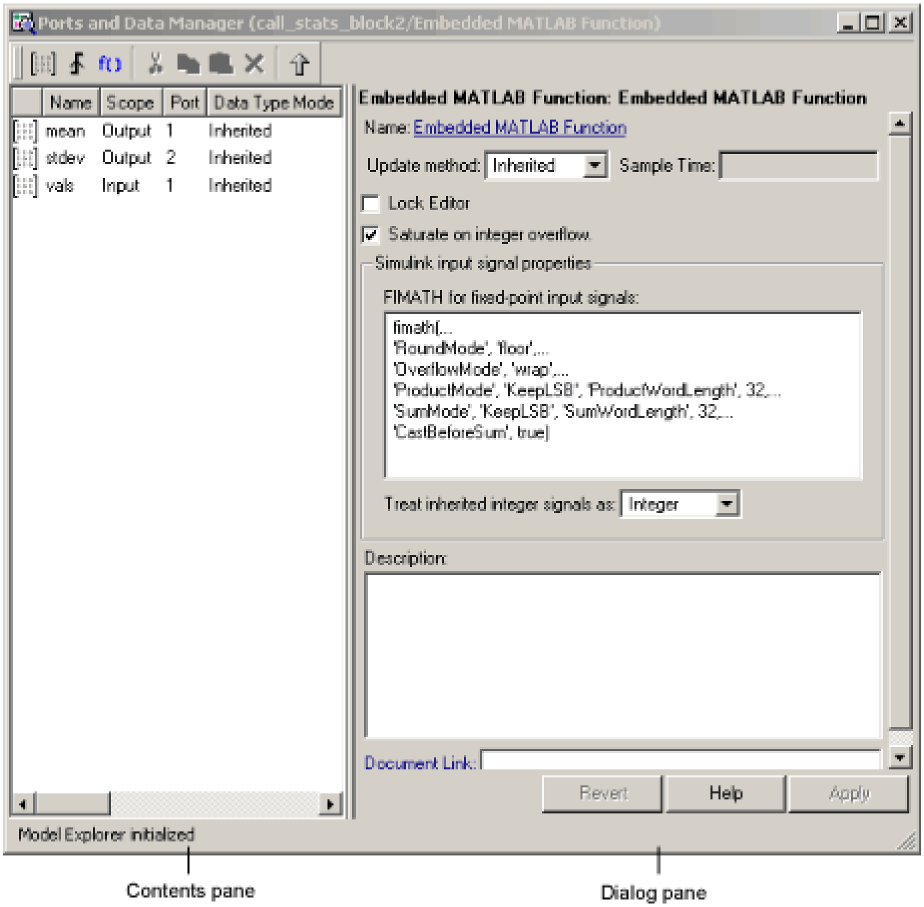
The Ports and Data Manager dialog allows you to define data arguments, input triggers, and function call outputs for Embedded MATLAB Function blocks. Using this dialog, you can also modify properties for the Embedded MATLAB Function and the objects it contains.

The dialog consists of two panes:

- Contents pane lists the objects that have been defined for the Embedded MATLAB Function block
- Dialog pane displays fields for modifying the properties of the selected object

Properties vary according to the scope and type of the object. Therefore, the **Ports and Data Manager** properties dialogs are dynamic, displaying only the property fields that are relevant for the object you add or modify.

When you first open the dialog, it displays the properties of the Embedded MATLAB Function block, as follows:



Ports and Data Manager (call_stats_block2/Embedded MATLAB Function)

Name	Scope	Port	Data Type Mode
mean	Output	1	Inherited
stdev	Output	2	Inherited
vals	Input	1	Inherited

Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function

Update method: Inherited Sample Time:

□ Lock Editor

☑ Saturate on integer overflow

Simulink input signal properties

FIMATH for fixed-point input signals:

```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'KeepLSB', 'ProductWordLength', 32,...
'SumMode', 'KeepLSB', 'SumWordLength', 32,...
'CastBeforeSum', true)
```

Treat inherited integer signals as: Integer

Description:

Document Link:

Model Explorer initialized

Contents pane

Dialog pane

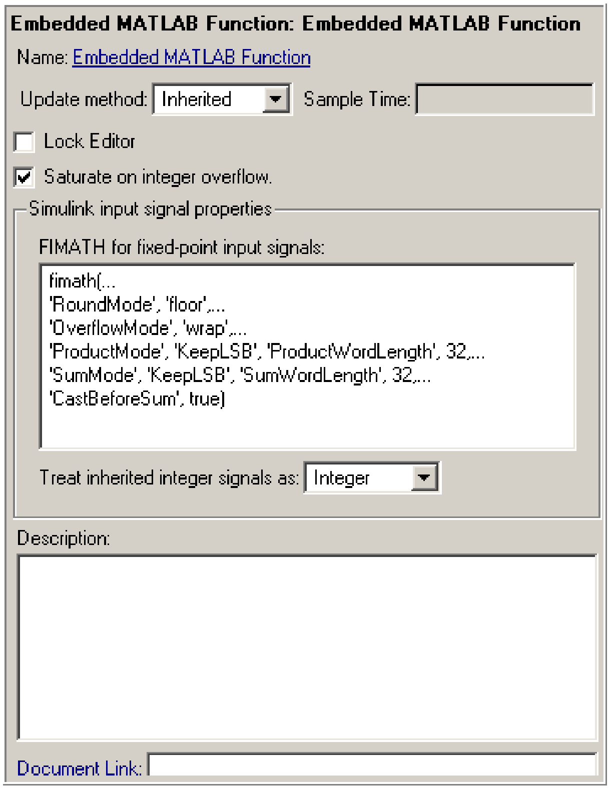
Opening the Ports and Data Manager

To open the **Ports and Data Manager** from the Embedded MATLAB Editor, select **Tools > Edit Data/Ports** or click the **Edit Data/Ports** icon:

The **Ports and Data Manager** appears for the Embedded MATLAB Function block that is open and has focus.

Setting Embedded MATLAB Function Block Properties

The Dialog pane for an Embedded MATLAB Function block looks like this:



Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function

Update method: Inherited   Sample Time:

□ Lock Editor

☑ Saturate on integer overflow.

Simulink input signal properties

FIMATH for fixed-point input signals:

```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'KeepLSB', 'ProductWordLength', 32,...
'SumMode', 'KeepLSB', 'SumWordLength', 32,...
'CastBeforeSum', true)
```

Treat inherited integer signals as: Integer

Description:

Document Link:

This section describes each property of an Embedded MATLAB Function block.

Name. Name of the Embedded MATLAB Function block, following the same naming conventions as for Simulink blocks (see “Manipulating Block Names” on page 4-25.

Update method. Method for activating the Embedded MATLAB Function block. You can choose from the following update methods:

Update method	Description
Inherited (default)	Input from the Simulink model activates the Embedded MATLAB Function block. If you define an input trigger, the Embedded MATLAB Function block executes in response to a Simulink signal or function-call event on the trigger port. If you do not define an input trigger, the Embedded MATLAB Function block implicitly inherits triggers from the Simulink model. These implicit events are the sample times (discrete or continuous) of the Simulink signals that provide inputs to the chart. If you define data inputs, the Embedded MATLAB Function block awakens at the rate of the fastest data input. If you do not define data inputs, the Embedded MATLAB Function block awakens as defined by its parent subsystem's execution behavior.
Discrete	Simulink awakens (samples) the Embedded MATLAB Function block at the rate you specify as the block's **Sample Time** property. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model can have different sample times.
Continuous	Simulink wakes up (samples) the Embedded MATLAB Function block at each step in the simulation, as well as at intermediate time points that can be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

Lock Editor. Option for locking the Embedded MATLAB Editor. When enabled, this option prevents users from making changes to the Embedded MATLAB Function block.

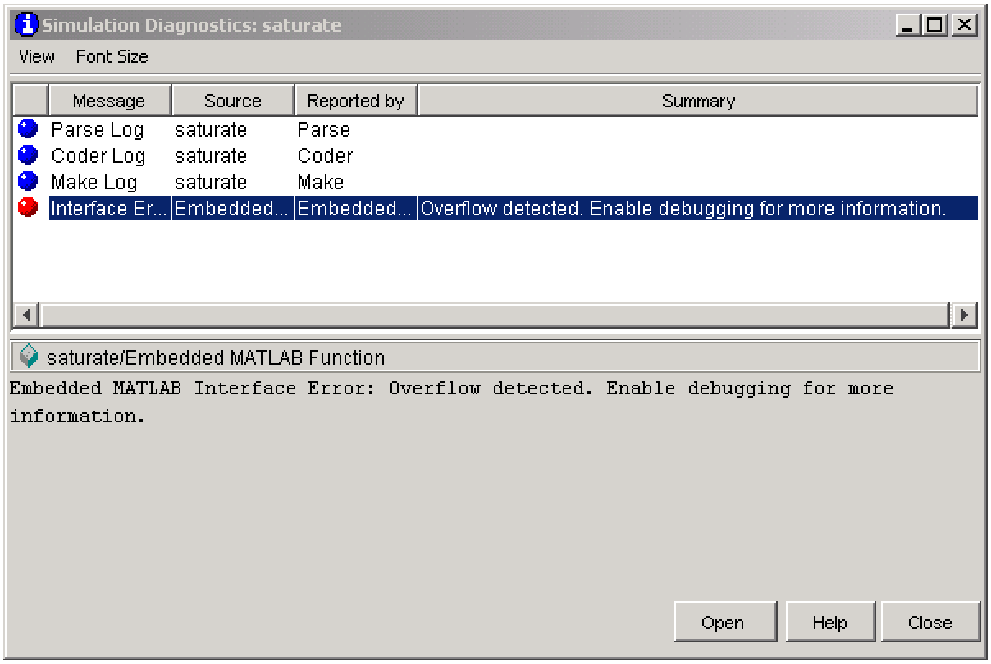
Saturate on integer overflow. Option that determines how the Embedded MATLAB Function block handles overflow conditions during integer operations, as follows:

Setting	Action When Overflow Occurs
Enabled (default)	Saturates integer by setting it to the maximum positive or negative value allowed by the word size. Matches MATLAB behavior.
Disabled	In simulation mode, generates a runtime error. For RTW code generation, the behavior depends on your C-language compiler.

When you enable **Saturate on Integer Overflow**, the Embedded MATLAB Function block adds additional checks in the generated code to detect integer overflow or underflow. Therefore, it is more efficient to disable this option if you are sure that integer overflow and underflow will not occur in your Embedded MATLAB function code.

Even when you disable this option, the code for a simulation target checks for integer overflow and underflow. If either condition occurs, simulation stops and an error is generated. If you enabled debugging for the Embedded MATLAB Function block, the debugger displays the error and lets you examine the data.

If you have not enabled debugging for the Embedded MATLAB Function block, the block generates a runtime error, as in this example:



	Message	Source	Reported by	Summary
●	Parse Log	saturate	Parse	
●	Coder Log	saturate	Coder	
●	Make Log	saturate	Make	
●	Interface Er...	Embedded...	Embedded...	Overflow detected. Enable debugging for more information.

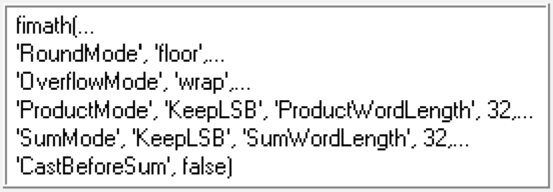
saturate/Embedded MATLAB Function

```
Embedded MATLAB Interface Error: Overflow detected. Enable debugging for more
information.
```

It is important to note that the code for an RTW target does *not* check for integer overflow or underflow and, therefore, may produce unpredictable results when **Saturate on Integer Overflow** is disabled. In this situation, it is recommended that you simulate first to test for overflow and underflow before generating the RTW target.

The **Saturate on Integer Overflow** option is relevant only for integer arithmetic. It has no effect on fixed point or double-precision arithmetic.

Simulink input signal properties. Parameters that apply to Embedded MATLAB Function blocks in models that use fixed-point or integer data types. You can specify the following properties for Simulink input signals:

Property	Description
FIMATH for fixed-point input signals	Defines the `fimath` object to be associated with Simulink fixed-point or integer signals that enter the Embedded MATLAB Function block as inputs. Enter an expression that evaluates to a `fimath` object, as in these examples: - Fully define the `fimath` object using the Fixed-Point Toolbox `fimath` function. - Enter the variable name of a `fimath` object that is defined in the MATLAB workspace. The default `fimath` object defined for this parameter emulates C-style math for a standard 32-bit processor:  fimath(... 'RoundMode', 'floor',... 'OverflowMode', 'wrap',... 'ProductMode', 'KeepLSB', 'ProductWordLength', 32,... 'SumMode', 'KeepLSB', 'SumWordLength', 32,... 'CastBeforeSum', false) This property applies to all input signals in the Embedded MATLAB Function block, not to each input individually, because signals with different `fimath` properties cannot interact. You can change the `fimath` properties of an input by casting it to a variable with the desired `fimath` properties, as follows: `x = fi(u, F);` In this example, `u` is the input, `x` is the variable, and `F` is the desired `fimath` object. For more information, see "Working with fimath Objects".
Treat inherited integer signals as	Specifies whether to treat inherited integer signals as MATLAB integers or Fixed-Point Toolbox `fi` objects.

Description. Description of the Embedded MATLAB Function block.

Document Link. Link to online documentation for the Embedded MATLAB Function block. To document an Embedded MATLAB Function block, set the **Document Link** property to a Web URL address or MATLAB expression that displays documentation in a suitable online format (for example, an HTML file or text in the MATLAB command window). The Embedded MATLAB Function block evaluates the expression when you click the blue **Document Link** text.

Adding Data to an Embedded MATLAB Function Block

You can define input and output data arguments for an Embedded MATLAB Function block directly in the script, or by using the Ports and Data Manager or Model Explorer. You can use the Ports and Data Manager to add data arguments to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of data arguments in the block.

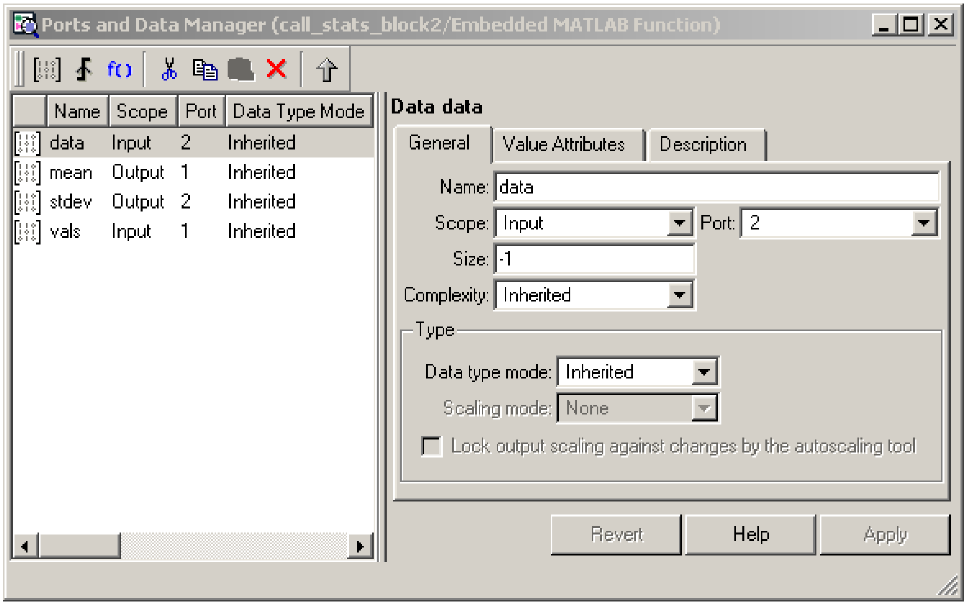
You can define the following data arguments for Embedded MATLAB Function blocks:

Method	For Defining	Reference
Define data directly in the Embedded MATLAB Function script	Input and output data	See “Defining Inputs and Outputs” on page 17-18.
Use the Ports and Data Manager	Input, output, and parameter data in the Embedded MATLAB Function that is open and has focus	See “Defining Data in the Ports and Data Manager” on page 17-52.
Use the Model Explorer	Input, output, and parameter data in Embedded MATLAB Function blocks at all levels of the Simulink model hierarchy	See “The Model Explorer” on page 10-2.

Defining Data in the Ports and Data Manager. To add a data argument and modify its properties, follow these steps:

1 In the Ports and Data Manager, click the **Add Data** icon:

The Ports and Data Manager adds a default definition of the data argument to the Embedded MATLAB Function block and displays the Data properties dialog in its Dialog pane, as in this example.



Ports and Data Manager (call_stats_block2/Embedded MATLAB Function)

Name	Scope	Port	Data Type Mode
data	Input	2	Inherited
mean	Output	1	Inherited
stdev	Output	2	Inherited
vals	Input	1	Inherited

Data data

General | Value Attributes | Description

Name: data

Scope: Input Port: 2

Size: -1

Complexity: Inherited

Type

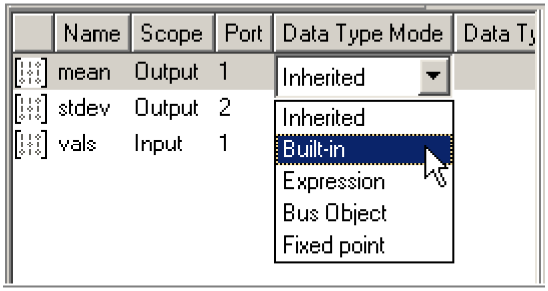
Data type mode: Inherited

Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

2 Modify properties for the new data argument, using one of the following methods:

- In the Contents pane, select the row that contains the data argument you want to modify and then click the value of the property of interest, as in this example:



	Name	Scope	Port	Data Type Mode	Data Ty
	mean	Output	1	Inherited	
	stdev	Output	2		
	vals	Input	1		

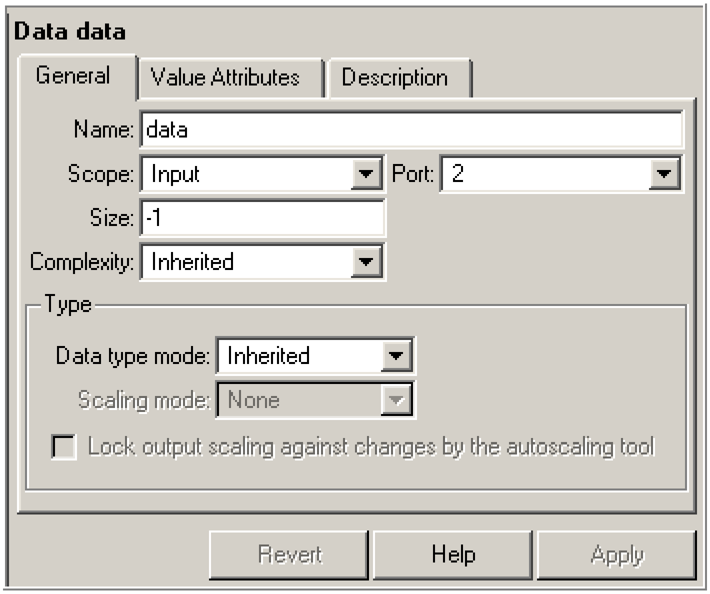
- Modify fields in the Data properties dialog, as described in "The Data Properties Dialog" on page 17-53.

The Data Properties Dialog. The Data properties dialog in the Ports and Data Manager allows you to set and modify the properties of data arguments in Embedded MATLAB Function blocks. Properties vary according to the scope and type of the data object. Therefore, the Data properties dialog is dynamic, displaying only the property fields that are relevant for the data argument you are defining.

You can open the Data properties dialog using one of these methods:

- Select a data argument in the Contents pane of the Ports and Data Manager to open the Data properties dialog in the Dialog pane.

The Data properties dialog provides a set of tabbed panes, as in this example:



Data data

Name: data

Scope: Input

Port: 2

Size: -1

Complexity: Inherited

Type

Data type mode: Inherited

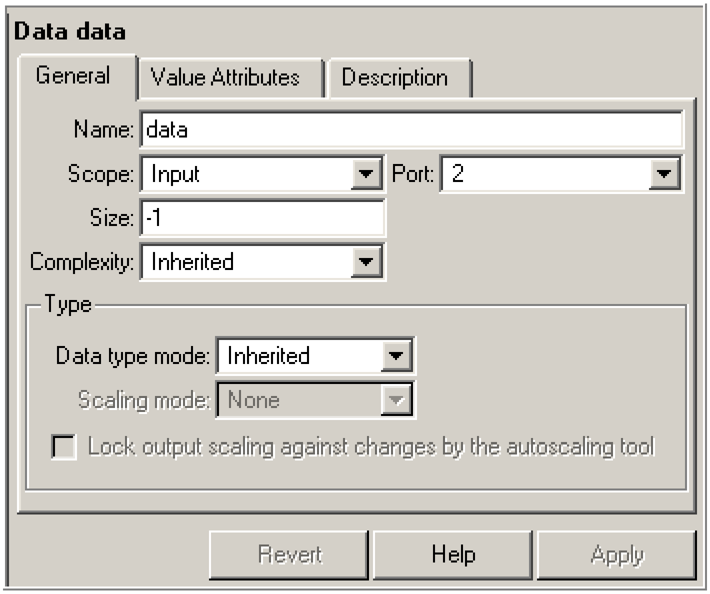
Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

Each pane lets you define different features of your data argument:

- The **General** pane lets you define the scope, size, complexity, and type of the data argument. See “Setting General Properties” on page 17-54.
- The **Value Attributes** pane lets you set a limit range, and save data argument values to the model’s base workspace. See “Setting Value Attributes Properties” on page 17-59.
- The **Description** pane lets you enter a description and link to documentation about the data argument. See “Setting Description Properties” on page 17-61.

Setting General Properties. The General tab of the Data properties dialog looks like this:



Data data

Name: data

Scope: Input

Port: 2

Size: -1

Complexity: Inherited

Type

Data type mode: Inherited

Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

You can set the following properties in the General tab:

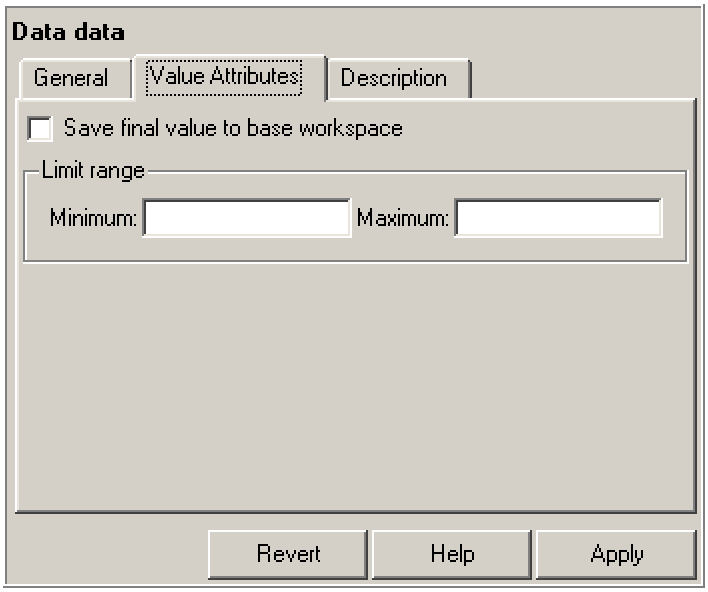
Property	Description
Name	Name of the data argument, following the same naming conventions used in MATLAB (see “Naming Variables” in the online MATLAB documentation.

Property	Description
Scope	Where data resides in memory, relative to its parent. Scope determines the range of functionality of the data argument. You can set scope to one of the following values: • **Parameter**: Specifies that the source for this data is a variable of the same name in the MATLAB or model workspace or in the workspace of a masked subsystem containing this block. If a variable of the same name exists in more than one of the workspaces visible to the block, Simulink uses the variable closest to the block in the in the workspace hierarchy (see "Working with Model Workspaces" on page 3-107). • **Input**: Data provided by the Simulink model via an input port to the Embedded MATLAB Function block. • **Output**: Data provided by the Embedded MATLAB Function block via an output port to the Simulink model. For more information, see "Defining Inputs and Outputs" on page 17-18 and "Parameter Arguments in Embedded MATLAB Functions" on page 17-85.
Port	Index of the port associated with the data argument. This property applies only to input and output data.
Tunable	Indicates whether the parameter used as the source of this data item is tunable (see "Tunable Parameters" on page 1-9). You must uncheck this option if you want to use the parameter where Embedded MATLAB requires a constant expression, such as `zeros` (see entry for `zeros` in "Embedded MATLAB Run-Time Function Library — Alphabetical List" in the Embedded MATLAB documentation). This property applies only to data with scope equal to **Parameter**.
Size	Size of the data argument. Size can be a scalar value or a MATLAB vector of values, as described in "Specifying Argument Sizes with Expressions" on page 17-83. Size defaults to –1, which means that it is inherited, as described in "Inheriting Argument Sizes from Simulink" on page 17-81. For more details, see "Sizing Function Arguments" on page 17-81.

Property	Description
Complexity	Indicates whether the value of the data argument is a real or complex number. You can set complexity to one of the following values: • **Off**: Data argument is a real number • **On**: Data argument is a complex number • **Inherited**: Data argument inherits complexity based on its scope. Input and output data inherit complexity from the Simulink signals connected to them; parameter data inherits complexity from the parameter to which it is bound.

Property	Description
Sampling mode	Specifies how an output signal propagates through a model. You can set sampling mode to one of the following values: • **Sample based**: Propagate the signal sample by sample (default) • **Frame based**: Propagate the signal in batches of samples This property applies only to data with scope equal to **Output**.
Data type mode and Data type	Data type mode lets you choose a method for specifying the type of a data argument. You can set data type mode to one of the following values: • **Inherited**: Data object inherits its type based on its scope. Input and output data inherit type from the Simulink signals connected to them; parameter data inherits type from the parameter to which it is bound. See "Inheriting Argument Data Types" on page 17-75. • **Built-in**: Select from a list of supported data types in the **Data type** field, as described in "Built-In Data Types for Arguments" on page 17-77. • **Expression**: In the **Data type** field, enter an expression that evaluates to a data type. See "Specifying Argument Types with Expressions" on page 17-77. • **Bus Object**: Data object is a structure (see "Working with Structures and Bus Signals" on page 17-87). In the **Data type** field, enter the name of a `Simulink.Bus` object that you have created in the base workspace to define the properties of the structure. Click the **Edit** button to create or modify `Simulink.Bus` objects using the Simulink Bus Types Editor (see "Using Bus Objects" on page 6-8 in the Using Simulink documentation). • **Fixed-point**: Specify word length, scaling mode, whether the data is signed or unsigned, and whether to lock output scaling, as described in "Specifying Fixed-Point Data Properties" on page 17-78.

Setting Value Attributes Properties. The **Value Attributes** tab of the **Data** properties dialog looks like this:



Data data

☐ Save final value to base workspace

Limit range

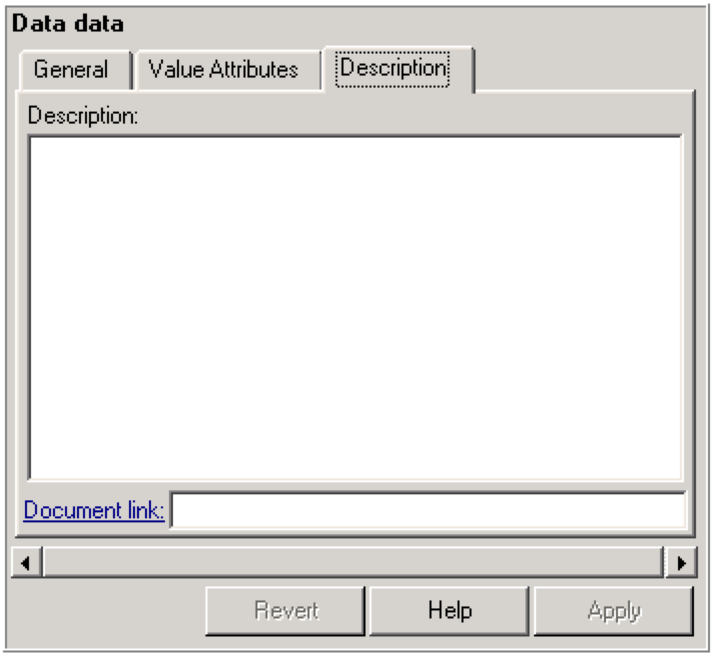
Minimum:

Maximum:

You can set the following properties on the Value Attributes tab:

Property	Description
Save final value to base workspace	If you select this option, the Embedded MATLAB Function block assigns the value of the data argument to a variable of the same name in the model's base workspace at the end of simulation (see "Working with Model Workspaces" on page 3-107).
Limit range properties	The range of acceptable values for this data object. The Embedded MATLAB Function block uses this range to validate the data object during simulation. To establish the range, specify these properties: • **Maximum** — The largest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value. • **Minimum** — The smallest value allowed for the data item during simulation. You can enter an expression or parameter that evaluates to a numeric scalar value. If you do not specify a value, the default for Maximum is `inf` and the default for Minimum is `-inf`.

Setting Description Properties. The Description tab of the Data properties dialog looks like this:



You can set the following properties on the Description tab:

Property	Description
Description	Description of the data argument.
Document link	Link to online documentation for the data argument. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB command window. When you click the blue text that reads **Document link** displayed at the bottom of the **Data** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

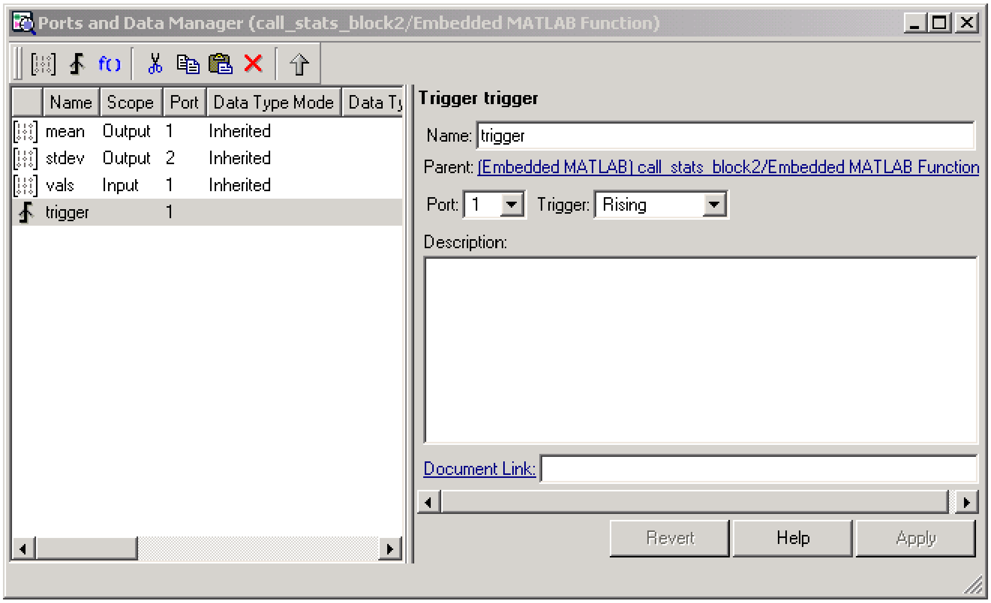
Adding Input Triggers to an Embedded MATLAB Function Block

You can use the Ports and Data Manager to add input triggers to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of input triggers in the block.

To add an input trigger and modify its properties, follow these steps:

1. In the Ports and Data Manager, click the **Add Input Trigger** icon:

   The Ports and Data Manager adds a default definition of the new input trigger to the Embedded MATLAB Function block and displays the Trigger properties dialog in its Dialog pane, as in this example.



Ports and Data Manager (call_stats_block2/Embedded MATLAB Function)

Name	Scope	Port	Data Type Mode	Data Ty
mean	Output	1	Inherited	
stdev	Output	2	Inherited	
vals	Input	1	Inherited	
trigger		1		

Trigger trigger

Name: trigger

Parent: [Embedded MATLAB] call_stats_block2/Embedded MATLAB Function

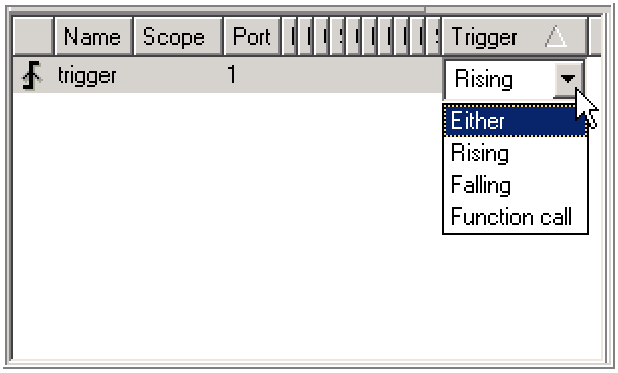
Port: 1 Trigger: Rising

Description:

Document Link:

2. Modify properties for the new input trigger, using one of the following methods:

- In the Contents pane, select the row that contains the input trigger you want to modify and then click the value of the property of interest, as in this example:



Name	Scope	Port	Trigger
trigger		1	Rising

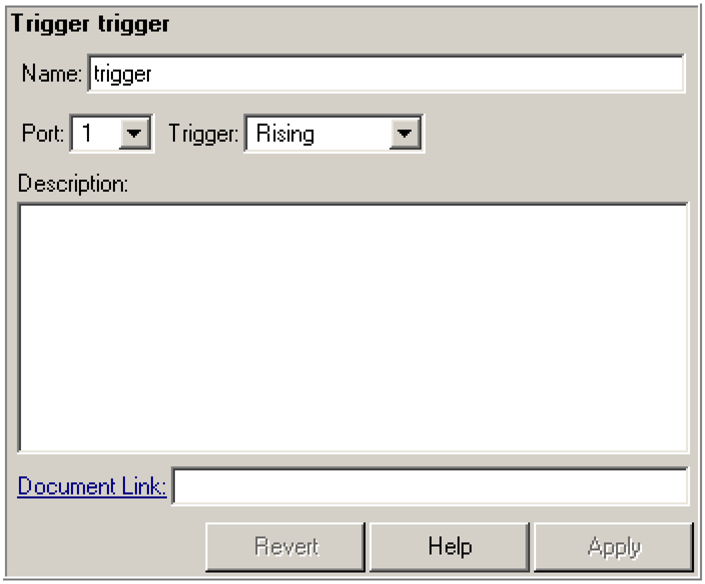
- Modify fields in the Trigger properties dialog, as described in "The Trigger Properties Dialog" on page 17-63.

The Trigger Properties Dialog. The Trigger properties dialog in the Ports and Data Manager allows you to set and modify the properties of input triggers in Embedded MATLAB Function blocks.

You can open the Trigger properties dialog using one of these methods:

- Select an input trigger in the Contents pane of the Ports and Data Manager to open the Trigger properties dialog in the Dialog pane.
- Right-click an input trigger in the Contents pane and select **Properties** from the submenu to open the Trigger properties dialog outside the Ports and Data Manager.

The Trigger properties dialog looks like this:



Trigger trigger

Name: trigger

Port: 1 Trigger: Rising

Description:

Document Link:

Setting Input Trigger Properties. You can set the following properties in the Trigger properties dialog:

Property	Description
Name	Name of the input trigger, following the same naming conventions used in MATLAB (see "Naming Variables" in the online MATLAB documentation.
Port	Index of the port associated with the input trigger.

Property	Description
Trigger	Type of event that triggers execution of the Embedded MATLAB Function block. You can select one of the following types of triggers: • **Rising**: Triggers execution of the Embedded MATLAB Function block when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative). • **Falling**: Triggers execution of the Embedded MATLAB Function block when the control signal falls from a positive or zero value to a negative value (or zero if the initial value is positive). • **Either**: Triggers execution of the Embedded MATLAB Function block when the control signal is either rising or falling. • **Function call**: Triggers execution of the Embedded MATLAB Function block from a Simulink block that outputs function-call events, or from an S-function
Description	Description of the input trigger.
Document link	Link to online documentation for the input trigger. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB command window. When you click the blue text that reads **Document link** displayed at the bottom of the **Trigger** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

Adding Function Call Outputs to an Embedded MATLAB Function Block

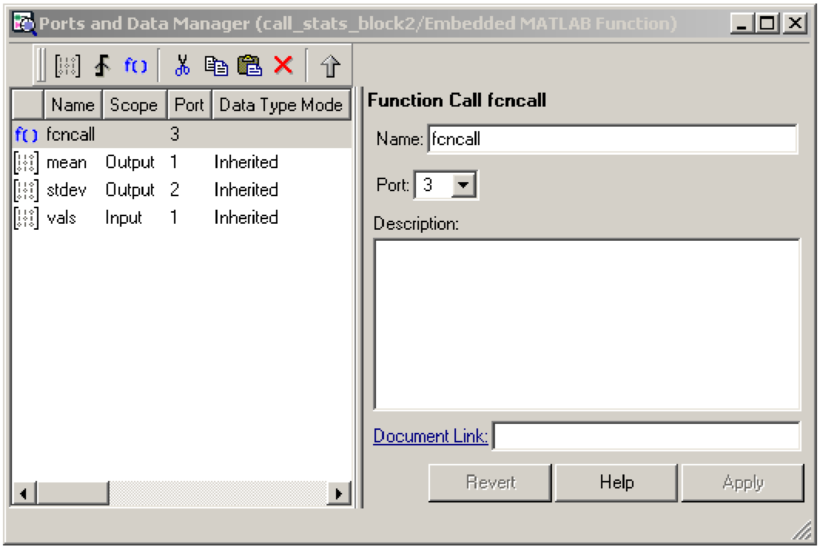
You can use the Ports and Data Manager to add function call outputs to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of function call outputs in the block.

To add a function call output and modify its properties, follow these steps:

1 In the Ports and Data Manager, click the **Add Function Call Output** icon:

f()

The Ports and Data Manager adds a default definition of the new function call output to the Embedded MATLAB Function block and displays the Function Call properties dialog in its Dialog pane, as in this example:



Ports and Data Manager (call_stats_block2/Embedded MATLAB Function)

	Name	Scope	Port	Data Type Mode
f()	fcncall		3	
	mean	Output	1	Inherited
	stdev	Output	2	Inherited
	vals	Input	1	Inherited

Function Call fcncall

Name: fcncall

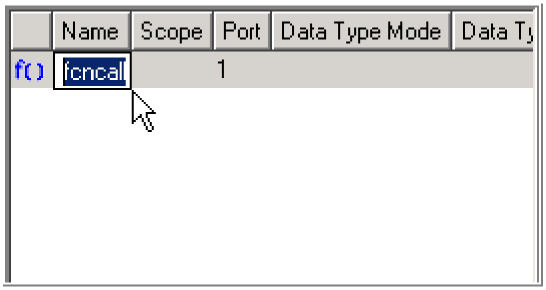
Port: 3

Description:

Document Link:

2 Modify properties for the new function call output, using one of the following methods:

- In the Contents pane, select the row that contains the function call output you want to modify and then click the value of the property of interest, as in this example:



	Name	Scope	Port	Data Type Mode	Data Ty
f()	fcncall		1		

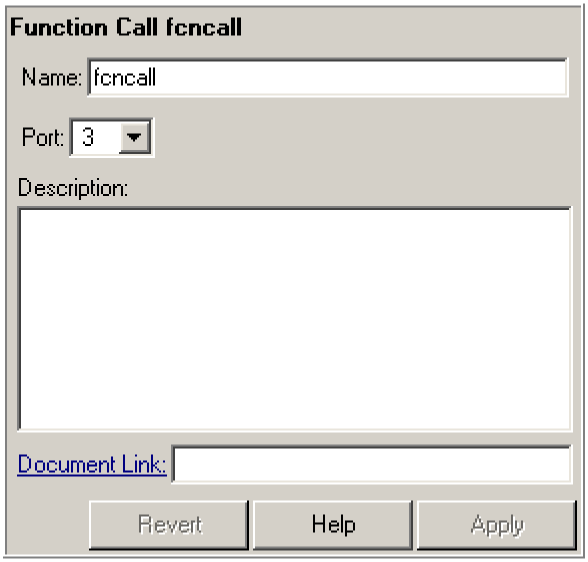
- Modify fields in the Function Call properties dialog, as described in “The Function Call Properties Dialog” on page 17-67.

The Function Call Properties Dialog. The Function Call properties dialog in the Ports and Data Manager allows you to set and modify the properties of function call outputs in Embedded MATLAB Function blocks.

You can open the Function Call properties dialog using one of these methods:

- Select a function call output in the Contents pane of the Ports and Data Manager to open the Function Call properties dialog in the Dialog pane.
- Right-click a function call output in the Contents pane and select **Properties** from the submenu to open the Function Call properties dialog outside the Ports and Data Manager.

The Function Call properties dialog looks like this:



Function Call fcncall

Name: fcncall

Port: 3

Description:

Document Link:

Setting Function Call Output Properties. You can set the following properties in the Function Call properties dialog:

Property	Description
Name	Name of the function call output, following the same naming conventions used in MATLAB (see "Naming Variables" in the online MATLAB documentation.
Port	Index of the port associated with the function call output.
Description	Description of the function call output.
Document link	Link to online documentation for the function call output. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable online format, such as an HTML file or text in the MATLAB command window. When you click the blue text that reads **Document link** displayed at the bottom of the **Function Call** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

Debugging Embedded MATLAB Functions

Use the following tools during an Embedded MATLAB function debugging session:

Tool Button	Description
●	A breakpoint indicator. To set a breakpoint for a line of function code, click the hyphen character (-) in the breakpoints column for the line. A breakpoint indicator appears in place of the hyphen. Click the breakpoint indicator to clear the breakpoint.
Build	Check for errors and build a simulation application (if no errors are found) for the model containing this Embedded MATLAB function.
▶ **Start Simulation**	Start simulation of the model containing the Embedded MATLAB function. Alternatively, press **F5**, or, from the **Debug** menu, select **Start**.
■ **Stop Simulation**	Stop simulation of the model containing the Embedded MATLAB function. You can also select **Exit debug mode** from the **Debug** menu if execution is paused at a breakpoint.
Set/Clear Breakpoint	Set a new breakpoint or clear an existing breakpoint for the selected Embedded MATLAB code line. The presence of the text cursor or highlighted text selects the line.
Clear All Breakpoints	Clear all set breakpoints in the Embedded MATLAB function.

Tool Button	Description
Step	Step through the execution of the next Embedded MATLAB code line. This tool steps past function calls and does not enter called functions for line-by-line execution. You can use this tool only after execution has stopped at a breakpoint. Alternatively, press **F11**, or, from the **Debug** menu, select **Step**.
Step In	Step through the execution of the next Embedded MATLAB code line. If the line calls a subfunction, step into line-by-line execution of the subfunction. You can use this tool only after execution has stopped at a breakpoint.
Step Out	Step out of line-by-line execution of the current subfunction to the line after the line that calls this subfunction. You can use this tool only after execution has stopped at a breakpoint.
Continue Debugging	Continue debugging after a pause, such as stopping at a breakpoint.

See “Debugging the Function in Simulation” on page 17-22 for an example using some of these debugging tools.

Typing Function Arguments

In "Programming the Embedded MATLAB Function" on page 17-9, you create two output arguments and an input argument for an Embedded MATLAB Function block by entering them in its function header. When you define arguments, Simulink creates corresponding ports on the Embedded MATLAB Function block that you can attach to Simulink signals. You can select a data type mode for each argument that you define for an Embedded MATLAB Function block. Each *data type mode* presents its own set of options for selecting a *data type*.

By default, the data type mode for Embedded MATLAB function arguments is **Inherited**. This means that the function argument inherits its data type from the incoming or outgoing Simulink signal. To override the default type, you first choose a data type mode and then select a data type based on the mode. The following procedure describes how to use the **Model Explorer** to set data types for function arguments. You can also use the **Ports and Data Manager** tool (see "Ports and Data Manager" on page 17-42).

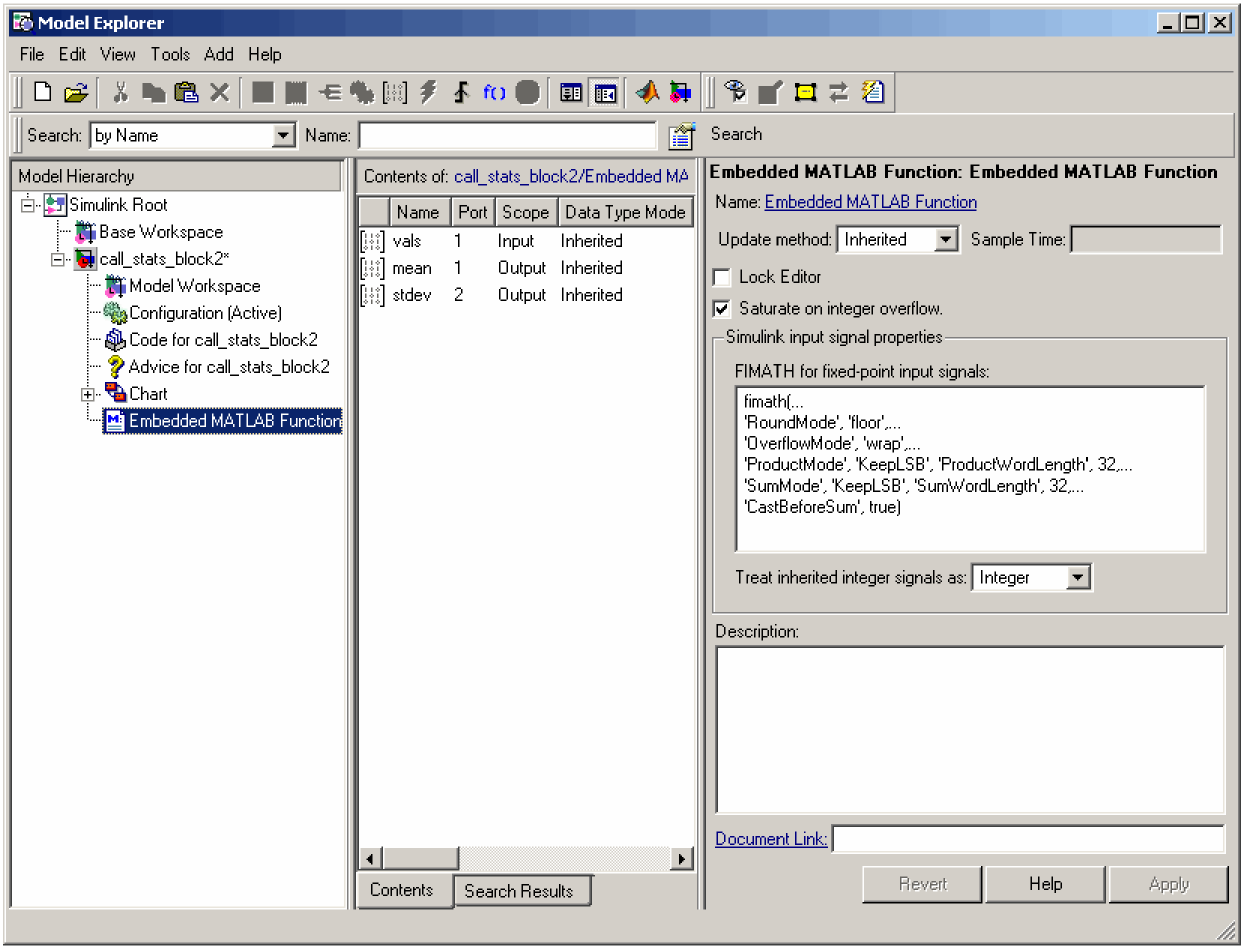
- "Specifying Argument Types" on page 17-71
- "Inheriting Argument Data Types" on page 17-75
- "Built-In Data Types for Arguments" on page 17-77
- "Specifying Argument Types with Expressions" on page 17-77
- "Specifying Fixed-Point Data Properties" on page 17-78

Specifying Argument Types

To specify the type of an Embedded MATLAB function argument:

1 From the **Embedded MATLAB Editor**, select **Model Explorer** from the **Tools** menu.

The **Model Explorer** appears with the Embedded MATLAB Function block highlighted in the **Model Hierarchy** pane on the left.



Contents of: call_stats_block2/Embedded MA

Name	Port	Scope	Data Type Mode
vals	1	Input	Inherited
mean	1	Output	Inherited
stdev	2	Output	Inherited

Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function

Update method: Inherited Sample Time:

☐ Lock Editor

☑ Saturate on integer overflow.

Simulink input signal properties

FIMATH for fixed-point input signals:

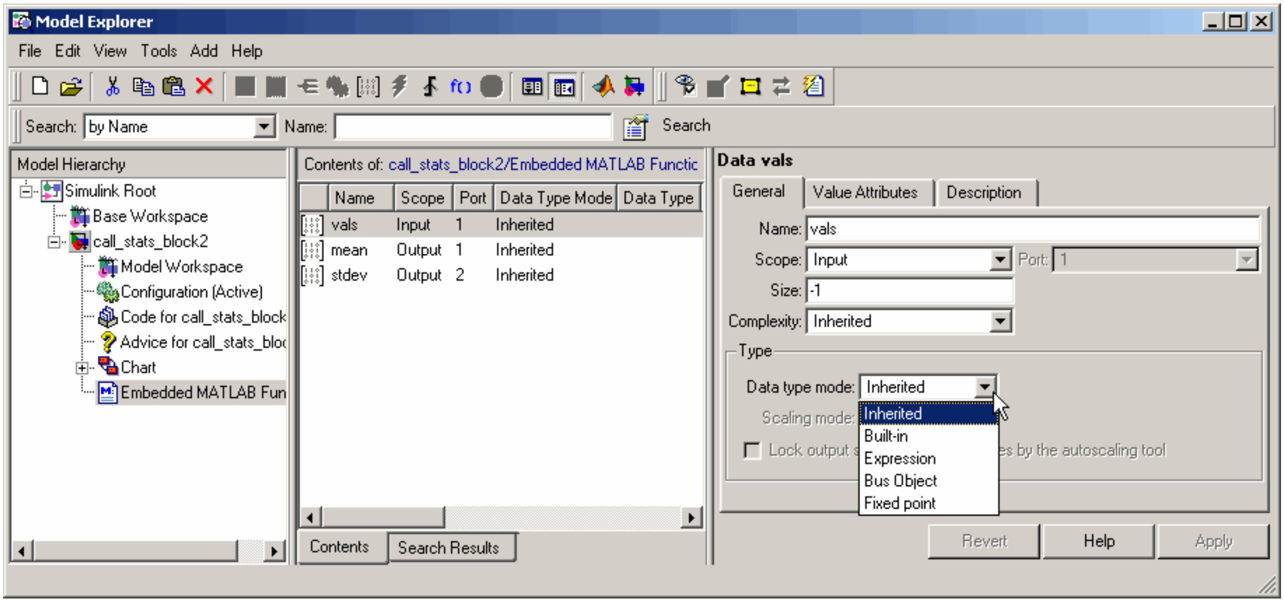
```
fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'KeepLSB', 'ProductWordLength', 32,...
'SumMode', 'KeepLSB', 'SumWordLength', 32,...
'CastBeforeSum', true)
```

Treat inherited integer signals as: Integer

Description:

Document Link:

2 In the **Contents** pane, click the row containing the argument of interest and choose an option from the drop-down menu in the **Data Type Mode** column, or from the **Data type mode** field in the **Data** properties dialog on the right, as shown:



Name	Scope	Port	Data Type Mode	Data Type
vals	Input	1	Inherited	
mean	Output	1	Inherited	
stdev	Output	2	Inherited	

Name: vals
Scope: Input Port: 1
Size: -1
Complexity: Inherited
Data type mode: Inherited

The **Data** properties dialog changes dynamically to display additional fields for specifying the data type associated with the mode.

3 Based on the mode you select, specify a data type as follows:

Mode	What To Specify
Inherited	You cannot specify a value. The data type is inherited from previously-defined data, based on the scope you selected for the Embedded MATLAB function argument, as follows: • If scope is **Input**, data type is inherited from the Simulink input signal on the designated port. • If scope is **Output**, data type is inherited from the Simulink output signal on the designated port. • If scope is **Parameter**, data type is inherited from the associated parameter, which can be defined in the Simulink masked subsystem or the MATLAB workspace. See "Inheriting Argument Data Types" on page 17-75.
Built-in	In the **Data type** field of the **Data** properties dialog or **Contents** pane, select from the drop-down list of supported data types, as described in "Built-In Data Types for Arguments" on page 17-77.
Expression	In the **Data type** field of the **Data** properties dialog or **Contents** pane, enter an expression that evaluates to a data type . See "Specifying Argument Types with Expressions" on page 17-77.

Mode	What To Specify
Bus Object	In the **Data type** field of the **Data** properties dialog or **Contents** pane, enter the name of a `Simulink.Bus` object that you have created in the base workspace to define the properties of an Embedded MATLAB structure. See "Working with Structures and Bus Signals" on page 17-87. **Note** You can click the **Edit** button to create or modify `Simulink.Bus` objects using the Simulink Bus Types Editor (see "Using Bus Objects" on page 6-8 in the Simulink User's Guide).
Fixed point	Specify the following information about the fixed point data in the **Data** properties dialog: • Whether the data is signed or unsigned • Word length • Scaling mode For information on how to specify these fixed point data properties, see "Specifying Fixed-Point Data Properties" on page 17-78.

Inheriting Argument Data Types

Embedded MATLAB Function arguments can inherit their data types, including fixed point types, from the Simulink signals to which they are connected. Select the argument of interest in the Contents pane of the Model Explorer or Ports and Data Manager, and set data type mode using one of these methods:

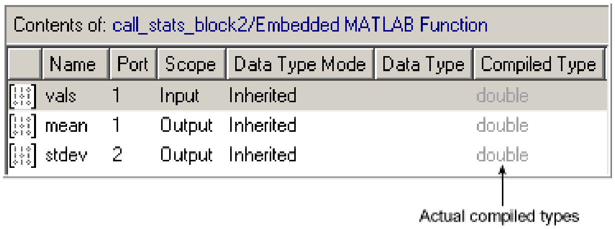
- In the Dialog pane, set the **Data type mode** field in the **Data** properties dialog to **Inherited**.
- In the Contents pane, set the **Data Type Mode** column to **Inherited**.

See “Built-In Data Types for Arguments” on page 17-77 for a list of supported data types.

Note An argument can also inherit its complexity (whether its value is a real or complex number) from the Simulink signal that is connected to it. To inherit complexity, set the **Complexity** field on the **Data** properties dialog to **Inherited**.

Once you build the model, the **Compiled Type** column of the Model Explorer or Ports and Data Manager gives the actual type used in the compiled simulation application. To conveniently compile and build the model from the **Embedded MATLAB Editor**, click the **Build** icon:

In the following example, an Embedded MATLAB Function block argument inherits its data type from an input signal of type `double`:



Contents of: call_stats_block2/Embedded MATLAB Function

Name	Port	Scope	Data Type Mode	Data Type	Compiled Type
vals	1	Input	Inherited		double
mean	1	Output	Inherited		double
stdev	2	Output	Inherited		double

Actual compiled types

The inherited type of output data is inferred from diagram actions that store values in the specified output. In the preceding example, the variables `mean` and `stdev` are computed from operations with double operands, which yield results of type `double`. If the expected type in Simulink matches the inferred type, inheritance is successful. In all other cases, a mismatch occurs during build time.

Note Library Embedded MATLAB function blocks can have inherited data types, sizes, and complexities like ordinary Embedded MATLAB function blocks. However, all instances of the library block in a given model must have inputs with the same properties.

Built-In Data Types for Arguments

When you select **Built-in** for **Data type mode**, the **Data** properties dialog displays a **Data type** field that provides a drop-down list of supported data types. You can also choose a data type from the **Data Type** column in the **Contents** pane of the Model Explorer or Ports and Data Manager. Here is a list of the supported data types:

Data Type	Description
`double`	64-bit double-precision floating point
`single`	32-bit single-precision floating point
`int32`	32-bit signed integer
`int16`	16-bit signed integer
`int8`	8-bit signed integer
`uint32`	32-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint8`	8-bit unsigned integer
`boolean`	Boolean (1 = true; 0 = false)

Specifying Argument Types with Expressions

You can specify the types of Embedded MATLAB Function arguments as expressions in the Model Explorer or Ports and Data Manager. Follow these steps:

1 Select **Expression** as the data type mode for your function argument, either in the Data properties dialog in the Dialog pane or Data Type Mode column in the Contents pane.

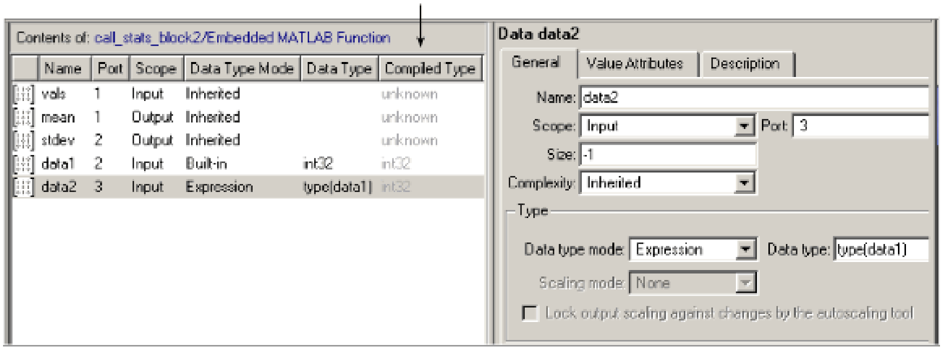
A Data type field appears in the Data properties dialog. The Contents pane also provides a Data Type column.

2 In the Data type field, enter an expression that evaluates to a data type. The following expressions are allowed:

- Alias type from the MATLAB workspace, as described in “Creating a Data Type Alias”.
- fixdt function to create a `Simulink.NumericType` object describing a fixed-point or floating-point data type
- `type` operator, to base the type on previously defined data

  In the following example, the data type of input argument `data1` is **int32**. The data type of input argument `data2` is based on `data1` using the expression `type(data1)`. Click the Build icon to compile and build the model from the **Embedded MATLAB Editor**.

  When the model is compiled, the actual type of `data2` appears in the Compiled Type column in the Contents pane:



Compiled data types

Contents of: call_stats_block2/Embedded MATLAB Function

Name	Port	Scope	Data Type Mode	Data Type	Compiled Type
vals	1	Input	Inherited		unknown
mean	1	Output	Inherited		unknown
stdev	2	Output	Inherited		unknown
data1	2	Input	Built-in	int32	int32
data2	3	Input	Expression	type(data1)	int32

Data data2

Name: data2

Scope: Input Port: 3

Size: -1

Complexity: Inherited

Type

Data type mode: Expression Data type: type(data1)

Scaling mode: None

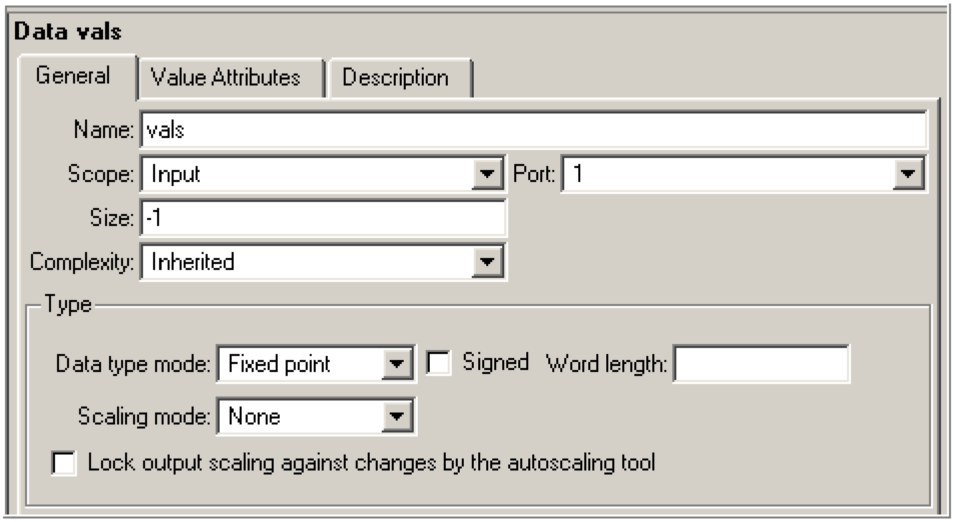
□ Lock output scaling against changes by the autoscaling tool

Specifying Fixed-Point Data Properties

Embedded MATLAB Function blocks can represent signals and parameter values as fixed-point numbers. To simulate models that use fixed-point data in Embedded MATLAB Function blocks, you must install the Simulink Fixed

Point product on your system (see “What Is Simulink Fixed Point?” in the Simulink Fixed Point online documentation).

When you select the data type mode **Fixed point**, the Type panel in the Data properties dialog displays new fields for specifying additional information about your fixed-point data, as in this example:



Data vals

Name: vals

Scope: Input

Port: 1

Size: -1

Complexity: Inherited

Type

Data type mode: Fixed point

□ Signed

Word length:

Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

You can set the following fixed-point properties:

Signed. Use this check box to indicate whether you want the fixed-point data to be signed or unsigned. Signed data can represent positive and negative quantities. Unsigned data represents positive values only.

Word length. Specify the size in bits of the word that will hold the quantized integer. Large word sizes represent large quantities with greater precision than small word sizes. Word length can be any integer between 0 and 32. If you do not specify a value, the default is 16.

Scaling mode. Specify the method for scaling your fixed point data to avoid overflow conditions and minimize quantization errors. Scaling is disabled by default. However, you can select two scaling modes:

Scaling Mode	Description
Binary point	If you select this mode, the **Data** properties dialog displays a field for entering fraction length that specifies the binary point location. Binary points can be positive or negative integers. If you do not specify a value, the default is 0. A positive integer entry moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer entry moves the binary point further right of the rightmost bit by that amount.
Slope and bias	If you select this mode, the **Data** properties dialog displays fields for entering separate values for the slope and bias. Slope can be any *positive* real number. If you do not specify a value, the default is 1.0. Bias can be any real number. If you do not specify a value, the default value is 0.0. You can enter slope and bias as expressions that contain parameters defined in the MATLAB workspace.

Lock output scaling against changes by the autoscaling tool. Use this check box to indicate whether you want to prevent Simulink from replacing the current fixed-point type with a type chosen by the autoscaling tool. See "Automatic Scaling" in Simulink Fixed Point documentation for instructions on autoscaling fixed-point data in Simulink.

Sizing Function Arguments

You specify the size of arguments of Embedded MATLAB Function blocks in the Model Explorer or Ports and Data Manager.

Specifying Argument Size (p. 17-81)	Describes how to specify the size of data arguments
Inheriting Argument Sizes from Simulink (p. 17-81)	Explains how a data argument can inherit its size from Simulink inputs, outputs, or parameters
Specifying Argument Sizes with Expressions (p. 17-83)	Describes valid expressions for specifying the size of data arguments

Specifying Argument Size

To examine or specify the size of an argument, follow these steps:

1. From the **Embedded MATLAB Editor**, select **Model Explorer** or **Edit Data/Ports** from the **Tools** menu.

2. In the **Contents** pane, click the row that contains the data argument.

3. Enter the size of the argument in one of two places in the Model Explorer or Ports and Data Manager:

   - Size field of the Data properties dialog, located in the Dialog pane
   - Size column in the row that contains the data argument, located in the Contents pane

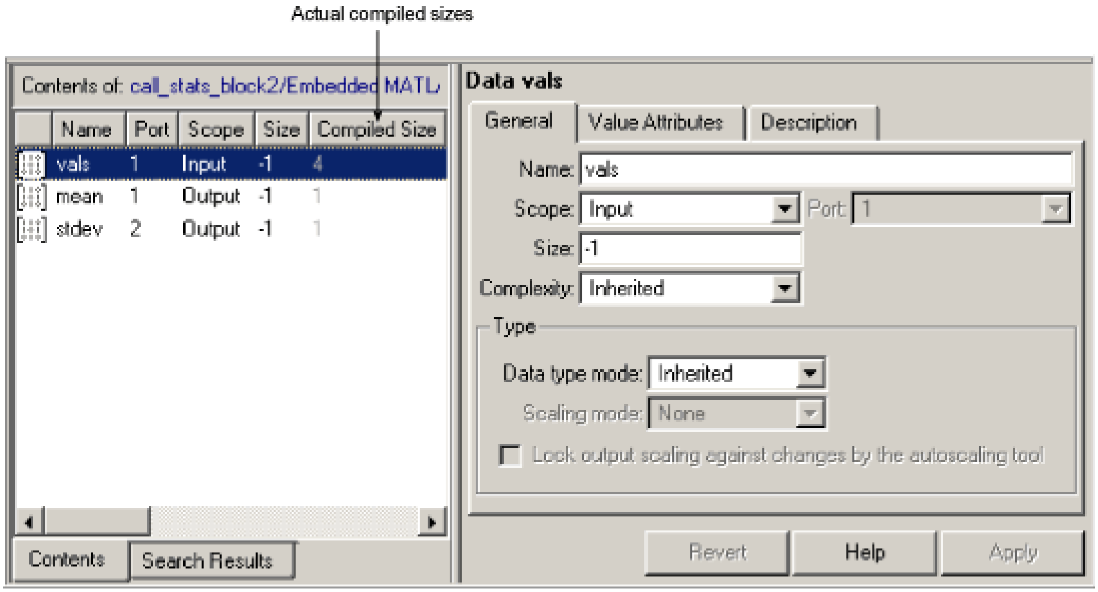
Note The default value is `-1`, indicating that size is inherited, as described in "Inheriting Argument Sizes from Simulink" on page 17-81.

Inheriting Argument Sizes from Simulink

Size defaults to 1, which means that the data argument inherits its size from Simulink based on its scope, as follows:

For Scope	Inherits Size
Input	From the Simulink input signal connected to the argument
Output	From the Simulink output signal connected to the argument
Parameter	From the Simulink or MATLAB parameter to which it is bound. See "Parameter Arguments in Embedded MATLAB Functions" on page 17-85.

After you compile the model, the **Compiled Size** column of the Model Explorer or Ports and Data Manager displays the actual size used in the compiled simulation application:



Actual compiled sizes

Contents of: call_stats_block2/Embedded MATL

Name	Port	Scope	Size	Compiled Size
vals	1	Input	-1	4
mean	1	Output	-1	1
stdev	2	Output	-1	1

Data vals

Name: vals

Scope: Input

Port: 1

Size: -1

Complexity: Inherited

Type

Data type mode: Inherited

Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

To conveniently compile the model from the **Embedded MATLAB Editor**, click the Build icon:

The size of an output argument is the size of the value that is assigned to it. If the expected size in Simulink does not match, a mismatch error occurs during compilation of the model.

Note No arguments with inherited sizes are allowed for Embedded MATLAB Function blocks in a library.

Specifying Argument Sizes with Expressions

The size of a data argument can be a scalar value or a MATLAB vector of values.

To specify size as a scalar, set the Size property to 1 or leave it blank. To specify size as a vector, enter an array of up to two dimensions in `[row column]` format where

- The number of dimensions equals the length of the vector
- The size of each dimension corresponds to the value of each element of the vector

For example, a value of `[2 4]` defines a 2-by-4 matrix. To define a row vector of size 5, set the **Size** field to `[1 5]`. To define a column vector of size 6, set the **Size** field to `[6 1]` or just `6`. You can enter a MATLAB expression for each `[row column]` element in the **Size** field. Each expression can use one or more of the following elements:

- Numeric constants
- Arithmetic operators, restricted to `+`, `-`, `*`, and `/`
- Parameters defined in the MATLAB workspace or the parent Simulink masked subsystem
- Calls to the MATLAB functions `min`, `max`, and `size`

The following examples are valid expressions for size:

```
k+1
size(x)
min(size(y),k)
```

In these examples, `k`, `x`, and `y` are variables of scope **Parameter**.

Once you build the model, the **Compiled Size** column of the Model Explorer or Ports and Data Manager displays the actual size used in the compiled simulation application.

Parameter Arguments in Embedded MATLAB Functions

Parameter arguments for Embedded MATLAB Function blocks do not take their values from Simulink signals. Instead, their values come from parameters defined in a parent Simulink masked subsystem or variables defined in the MATLAB base workspace. Using parameters allows you to pass a read-only constant in Simulink to the Embedded MATLAB Function block.

Use the following procedure to add a parameter argument to a function for an Embedded MATLAB Function block.

1. In the **Embedded MATLAB Editor**, add an argument to the function header of the Embedded MATLAB Function block.

   The name of the argument must be identical to the name of the masked subsystem parameter or MATLAB variable that you want to pass to the Embedded MATLAB Function block. For information on declaring parameters for masked subsystems in Simulink, see “Mask Editor” on page 13-17.

2. Bring focus to the Embedded MATLAB Function block in Simulink.

   The new argument appears as an input port in the Simulink diagram.

3. In the **Embedded MATLAB Editor**, select **Model Explorer** or **Edit Data/Ports** from the **Tools** menu.

4. In the **Contents** pane, click the row that contains the new argument.

5. Set the scope of the data argument to **Parameter**, either in the Data properties dialog in the Dialog pane or in the Scope column in the Contents pane.

6. Examine the Embedded MATLAB Function block in Simulink.

   The input port no longer appears for the parameter argument.

Note Parameter arguments appear as arguments in the function header of the Embedded MATLAB Function block to maintain MATLAB consistency. This lets you test functions in an Embedded MATLAB Function block by copying and pasting them to MATLAB.

Working with Structures and Bus Signals

The Embedded MATLAB structure is a data type that is based on the MATLAB structure (see "Structures" in the MATLAB Programming documentation). In Embedded MATLAB Function blocks, you can define structure data as inputs and outputs that interact with Simulink bus signals.

Note You can also define structures inside Embedded MATLAB functions that are not part of Embedded MATLAB Function blocks (see "Using Structures" in the Embedded MATLAB documentation).

About Structures in Embedded MATLAB Function Blocks

You can create structures in the primary function of Embedded MATLAB Function blocks to interface with Simulink bus signals at input and output ports. You can also create structures as local and persistent variables in primary functions and subfunctions of Embedded MATLAB Function blocks (see "Using Structures" in the Embedded MATLAB documentation).

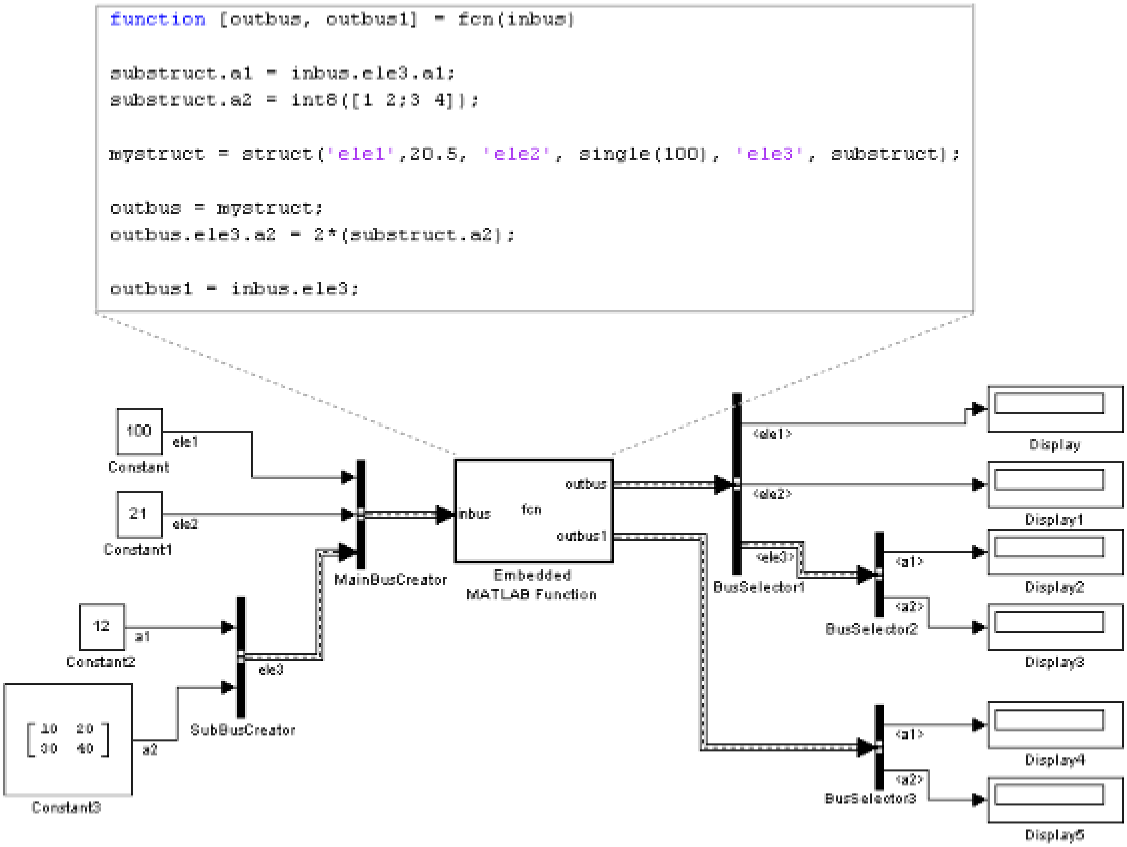
The following table summarizes how to create different types of structures in Embedded MATLAB Function blocks:

Scope	How to Create	Details
Input	Create structure data with scope of `Input` in Ports and Data Manager or Model Explorer	You can create structure data as inputs and outputs in the top-level Embedded MATLAB function for interfacing to other environments. See "Workflow for Creating Structures in Embedded MATLAB Function Block" on page 17-93.
Output	Create structure data with scope of `Output` in Ports and Data Manager or Model Explorer	
Local	Create local variable implicitly in Embedded MATLAB function	See "Defining Local Structure Variables" in the Embedded MATLAB documentation.
Persistent	Declare variable to be persistent in Embedded MATLAB function	See "Making Structures Persistent" in the Embedded MATLAB documentation.

Structures in Embedded MATLAB Function blocks can contain fields of any type and size, including composite data (such as muxed signals or buses) and arrays of structures, as described in "Elements of Embedded MATLAB Structures" in the Embedded MATLAB documentation.

Example of Structures in an Embedded MATLAB Function Block

The following example shows how to use structures in an Embedded MATLAB function block:



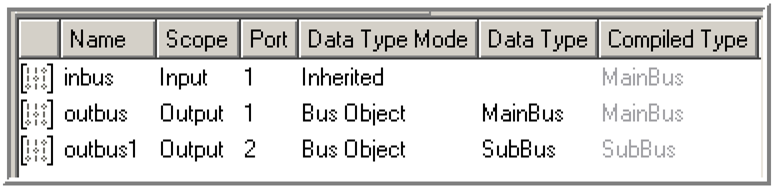
```
function [outbus, outbus1] = fcn(inbus)

substruct.a1 = inbus.ele3.a1;
substruct.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5, 'ele2', single(100), 'ele3', substruct);

outbus = mystruct;
outbus.ele3.a2 = 2*(substruct.a2);

outbus1 = inbus.ele3;
```

In this model, an Embedded MATLAB Function block receives a bus signal using the structure `inbus` at input port 1 and outputs two bus signals from the structures `outbus` at output port 1 and `outbus1` at output port 2. The input signal comes from the Simulink Bus Creator block `MainBusCreator`, which bundles signals `ele1`, `ele2`, and `ele3`. The signal `ele3` is the output of another Bus Creator block `SubBusCreator`, which bundles the signals `a1` and `a2`. The structure `outbus` connects to a Simulink Bus Selector block `BusSelector1`; the structure `outbus1` connects to another Simulink Bus Selector block `BusSelector2`.

Like other outputs in Embedded MATLAB, structure outputs must be initialized. The Embedded MATLAB function in this example implicitly defines a local structure variable mystruct using the struct function, and uses this local structure variable to initialize the value of the first output outbus. It initializes the second output outbus1 to the value of field ele3 of structure inbus.

Structure Definitions in Example

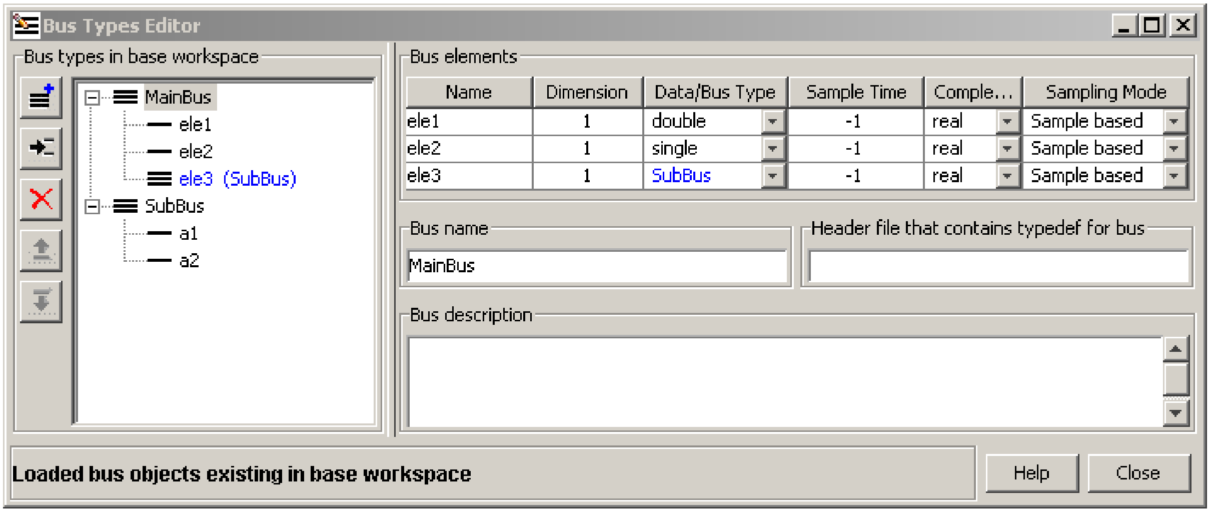
Here are the definitions of the structures in the Embedded MATLAB Function block in the example, as they appear in the Ports and Data Manager:



Name	Scope	Port	Data Type Mode	Data Type	Compiled Type
inbus	Input	1	Inherited		MainBus
outbus	Output	1	Bus Object	MainBus	MainBus
outbus1	Output	2	Bus Object	SubBus	SubBus

Simulink Bus Objects Define Structure Inputs and Outputs

Each structure input and output must be defined by a Simulink.Bus object in the base workspace (see “Workflow for Creating Structures in Embedded MATLAB Function Block” on page 17-93). This means that the structure shares the same properties as the bus object, including number, name, and type of fields. In this example, the following bus objects define the structure inputs and outputs:



Bus types in base workspace

- MainBus
  - ele1
  - ele2
  - ele3 (SubBus)
- SubBus
  - a1
  - a2

Bus elements

Name	Dimension	Data/Bus Type	Sample Time	Comple...	Sampling Mode
ele1	1	double	-1	real	Sample based
ele2	1	single	-1	real	Sample based
ele3	1	SubBus	-1	real	Sample based

Bus name: MainBus

Header file that contains typedef for bus:

Bus description:

Loaded bus objects existing in base workspace

The `Simulink.Bus` object `MainBus` defines structure input `inbus` and structure output `outbus`. The `Simulink.Bus` object `SubBus` defines structure output `outbus1`. Based on these definitions, `inbus` and `outbus` have the same properties as `MainBus` and, therefore, reference their fields by the same names as the fields in `MainBus`, using dot notation (see “Indexing Substructures and Fields” on page 17-95). Similarly, `outbus1` references its fields by the same names as the fields in `SubBus`. Here are the field references for each structure in this example:

Structure	First Field	Second Field	Third Field
`inbus`	`inbus.ele1`	`inbus.ele2`	`inbus.ele3`
`outbus`	`outbus.ele1`	`outbus.ele2`	`outbus.ele3`
`outbus1`	`outbus1.a1`	`outbus1.a2`	—

To learn how to define structures in Embedded MATLAB, see “Workflow for Creating Structures in Embedded MATLAB Function Block” on page 17-93.

How Structure Inputs and Outputs Interface with Bus Signals

Simulink buses appear inside the Embedded MATLAB Function block as structures; structure outputs from the Embedded MATLAB Function block appear as buses in Simulink. When you create structure inputs in Embedded MATLAB function blocks, Embedded MATLAB determines the type, size, and complexity of the structure from the Simulink input signal. When you create structure outputs, you must define their type, size, and complexity in the Embedded MATLAB function.

You can connect structure inputs and outputs from Embedded MATLAB Function blocks to any Simulink bus signal, including:

- Simulink blocks that output bus signals— such as Bus Creator blocks
- Simulink blocks that accept bus signals as input — such as Bus Selector and Gain blocks
- S-Function blocks
- Other Embedded MATLAB Function blocks

Working with Virtual and Nonvirtual Buses

Embedded MATLAB Function blocks supports nonvirtual buses only (see "Virtual and Nonvirtual Buses" on page 6-15 in the Simulink User's Guide). When Simulink builds models that contain Embedded MATLAB function inputs and outputs, it uses a hidden converter block to convert bus signals for use with Embedded MATLAB, as follows:

- Converts incoming virtual bus signals to nonvirtual buses for Embedded MATLAB structure inputs
- Converts outgoing nonvirtual bus signals from Embedded MATLAB to virtual bus signals, if necessary

Rules for Defining Structures in Embedded MATLAB Function Blocks

Follow these rules when defining structures in Embedded MATLAB Function blocks:

- For each structure input or output in an Embedded MATLAB Function block, you must define a `Simulink.Bus` object in the base workspace to specify its type to Simulink (see Simulink.Bus in the Simulink Reference documentation.
- Embedded MATLAB Function blocks support nonvirtual buses only (see “Working with Virtual and Nonvirtual Buses” on page 17-92).
- Structures cannot have scopes defined as **Parameter**.

Workflow for Creating Structures in Embedded MATLAB Function Block

Here is the workflow for creating a structure in Embedded MATLAB:

1. Decide on the type (or scope) of the structure (see “About Structures in Embedded MATLAB Function Blocks” on page 17-87).
2. Based on the scope, follow these guidelines for creating the structure:

For Structure Scope:	Requirements
Input	You must: **1** Create a `Simulink.Bus` object in the base workspace to define the structure input. **2** Add data to the Embedded MATLAB Function block, as described in "Adding Data to an Embedded MATLAB Function Block" on page 17-51. The data should have the following properties • Scope = **Input** • Data type mode = **Bus Object** • Data type = name of the `Simulink.Bus` object that defines the structure input See "Rules for Defining Structures in Embedded MATLAB Function Blocks" on page 17-92.
Output	You must: **1** Create a `Simulink.Bus` object in the base workspace to define the structure output. **2** Add data to the Embedded MATLAB Function block with the following properties: • Scope = **Output** • Data type mode = **Bus Object** • Data type = name of the `Simulink.Bus` object that defines the structure input **3** Define and initialize the output structure implicitly as a variable in the Embedded MATLAB function, as described in "Defining Outputs as Structures" in the Embedded MATLAB documentation. **4** Make sure the number, type, and size of fields in the output structure variable definition match the properties of the `Simulink.Bus` object.

For Structure Scope:	Requirements
Local	You must define the structure implicitly as a local variable in the Embedded MATLAB function, as described in "Defining Local Structure Variables" in the Embedded MATLAB documentation. By default, local variables in Embedded MATLAB are temporary variables.
Persistent	You must define the structure implicitly as a persistent variable in the Embedded MATLAB function, as described in "Making Structures Persistent" in the Embedded MATLAB documentation.

Indexing Substructures and Fields

As in MATLAB, you index substructures and fields of Embedded MATLAB structures by using dot notation. Unlike MATLAB, you must reference field values individually (see "Reference Field Values Individually from Structure Arrays" in the Embedded MATLAB documentation).

For example, in the model described in "Example of Structures in an Embedded MATLAB Function Block" on page 17-88, the Embedded MATLAB function uses dot notation to index fields and substructures:

```
function [outbus, outbus1] = fcn(inbus)

substruct.a1 = inbus.ele3.a1;
substruct.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                  'ele3',substruct);

outbus = mystruct;
outbus.ele3.a2 = 2*(substruct.a2);

outbus1 = inbus.ele3;
```

The following table shows how Embedded MATLAB resolves symbols in dot notation for indexing elements of the structures in this example:

Dot Notation	Symbol Resolution
`substruct.a1`	Field `a1` of local structure `substruct`
`inbus.ele3.a1`	Value of field `a1` of field `ele3`, a substructure of structure input`inbus`
`inbus.ele3.a2(1,1)`	Value in row 1, column 1 of field `a2` of field `ele3`, a substructure of structure input `inbus`

Assigning Values to Structures and Fields

You can assign values to any Embedded MATLAB structure, substructure, or field. Here are the guidelines:

Operation	Conditions
Assign one structure to another structure	You must define each structure with the same number, type, and size of fields, either as `Simulink.Bus` objects in the base workspace or locally as implicit structure declarations (see “Workflow for Creating Structures in Embedded MATLAB Function Block” on page 17-93).
Assign one structure to a substructure of a different structure and vice versa	You must define the structure with the same number, type, and size of fields as the substructure, either as `Simulink.Bus` objects in the base workspace or locally as implicit structure declarations.
Assign an element of one structure to an element of another structure	The elements must have the same type and size.

For example, the following table presents valid and invalid structure assignments based on the specifications for the model described in "Example of Structures in an Embedded MATLAB Function Block" on page 17-88:

Assignment	Valid or Invalid?	Rationale
outbus = mystruct;	Valid	Both outbus and mystruct have the same number, type, and size of fields. The structure outbus is defined by the Simulink.Bus object MainBus and mystruct is defined locally to match the field properties of MainBus.
outbus = inbus;	Valid	Both outbus and inbus are defined by the sameSimulink.Bus object, MainBus.
outbus1 = inbus.ele3;	Valid	Both outbus1 and inbus.ele3 have the same type and size because each is defined by the Simulink.Bus object SubBus.
outbus1 = inbus;	Invalid	The structure outbus1 is defined by a different Simulink.Bus object than the structure inbus.

Limitations of Structures in Embedded MATLAB Function Blocks

Structures in Embedded MATLAB Function Blocks support a subset of the operations available for MATLAB structures (see "Limitations with Structures" in the Embedded MATLAB documentation.

Working with Frame-Based Signals

Embedded MATLAB Function blocks can input and output frame-based signals in Simulink models. A frame of data is a collection of sequential samples from a single channel or multiple channels. To generate frame-based signals, you must install Signal Processing Blockset. For more information about using frame-based signals in Simulink, see "Frame-Based Signals" in the Signal Processing Blockset documentation.

Embedded MATLAB Function blocks automatically convert incoming frame-based signals as follows:

- Converts a single-channel frame-based signal to a MATLAB column vector
- Converts a multichannel frame-based signal to a two-dimensional MATLAB matrix

An M-by-N frame-based signal represents M consecutive samples from each of N independent channels. N-D signals are not supported for frames.

To convert matrix or vector data to a frame-based output, Embedded MATLAB provides a data property called sampling mode that lets you specify whether your output is a frame-based or sample-based signal for downstream processing.

The following topics explain how to work with frame-based signals in Embedded MATLAB Function blocks.

- "Supported Types for Frame-Based Data" on page 17-98
- "Adding Frame-Based Data in Embedded MATLAB Function Blocks" on page 17-99
- "Examples of Frame-Based Signals in Embedded MATLAB Function Blocks" on page 17-99

Supported Types for Frame-Based Data

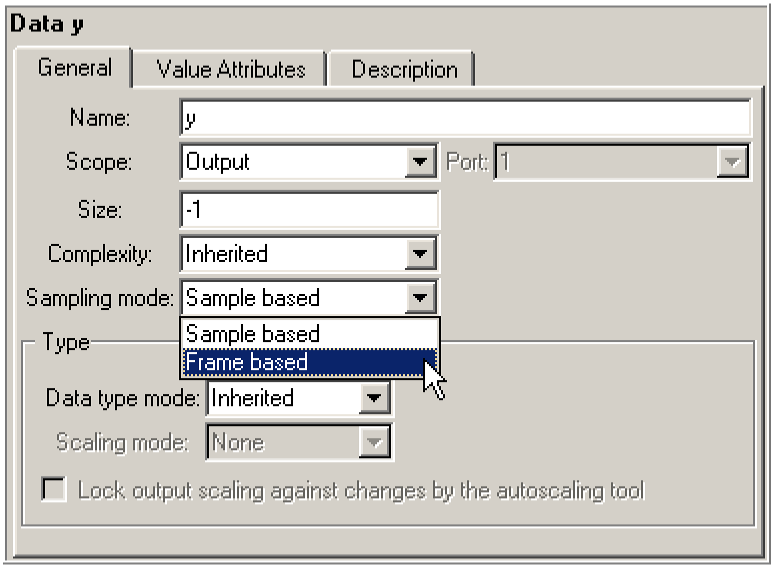
Embedded MATLAB Function blocks accept frame-based signals of any data type **except** bus objects. For a list of supported types, see "Supported Variable Types" in the Embedded MATLAB documentation.

Adding Frame-Based Data in Embedded MATLAB Function Blocks

To add frame-based data to an Embedded MATLAB Function block, follow these steps:

1. Add an input or output, as described in “Adding Data to an Embedded MATLAB Function Block” on page 17-51.

2. If your data is an output, set the general property **Sampling mode** to **Frame based**, as in this example:



Data y

Name: y

Scope: Output

Port: 1

Size: -1

Complexity: Inherited

Sampling mode: Sample based

Type

Data type mode: Inherited

Scaling mode: None

□ Lock output scaling against changes by the autoscaling tool

Note For more information on how to set data properties, see “The Data Properties Dialog” on page 17-53.

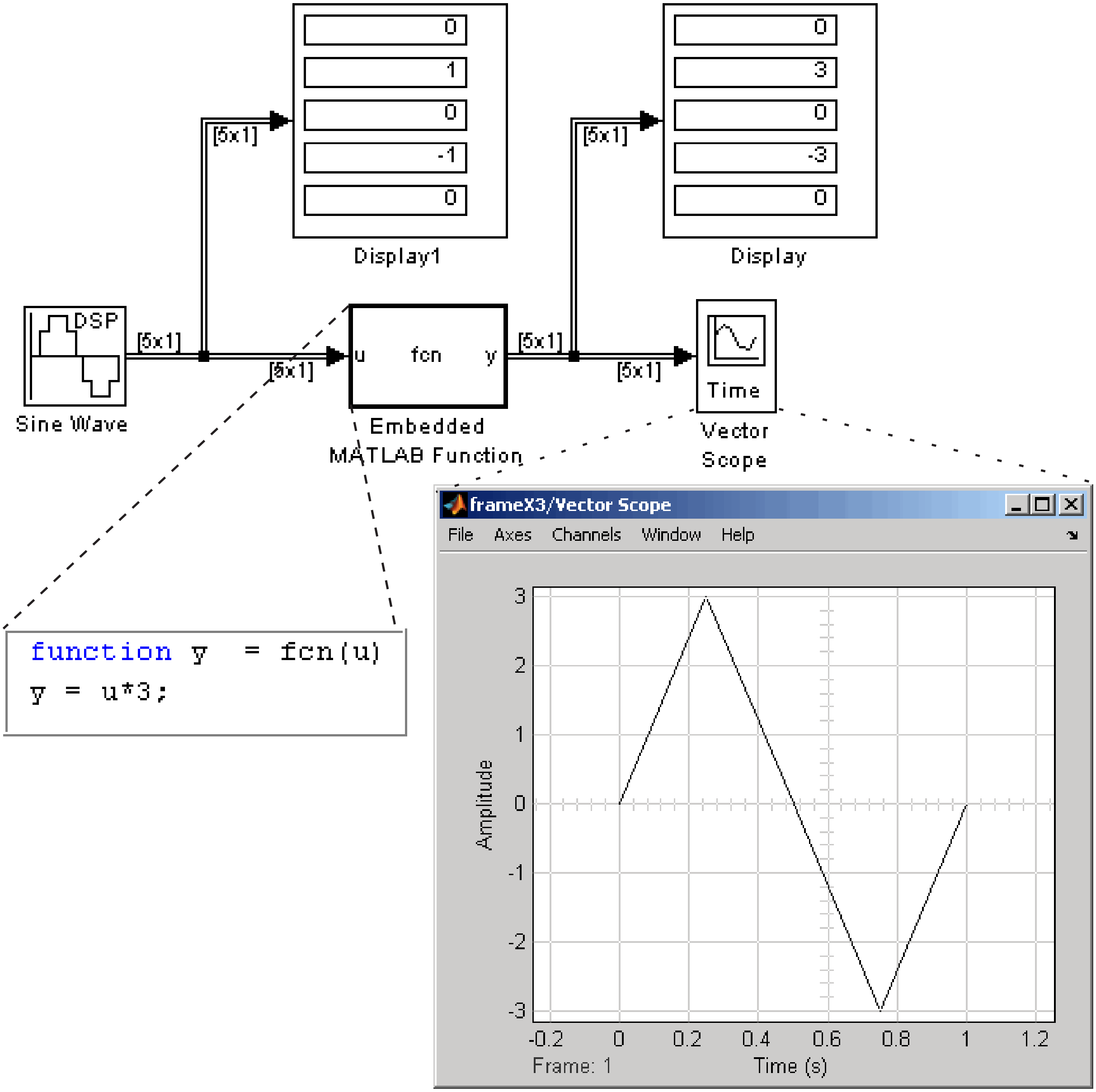
Examples of Frame-Based Signals in Embedded MATLAB Function Blocks

This topic presents examples of how to work with frame-based signals in Embedded MATLAB Function blocks.

- “Multiplying a Frame-Based Signal by a Constant Value” on page 17-100
- “Adding a Channel to a Frame-Based Signal” on page 17-102

Multiplying a Frame-Based Signal by a Constant Value

In the following example, an Embedded MATLAB Function block multiplies all the signal values in a frame-based single-channel input by a constant value and outputs the result as a frame. The input signal is a sine wave that contains 5 samples per frame. Here is the model:

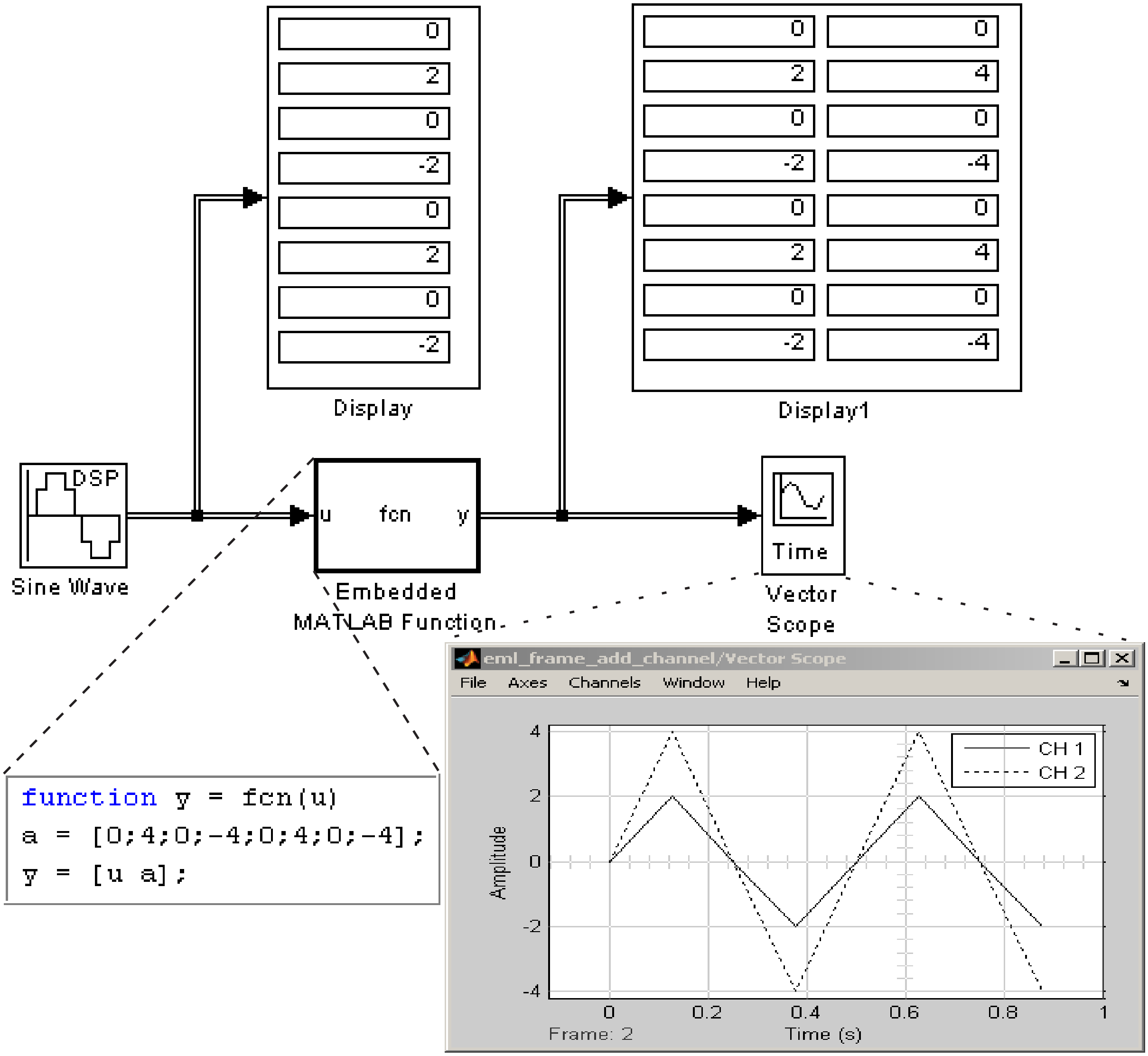


```
function y  = fcn(u)
y = u*3;
```

In the Embedded MATLAB Function block, input u and output y inherit their size, complexity, and data type from the sine wave signal, a 5-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set its sampling mode to **Frame based** (see "Adding Frame-Based Data in Embedded MATLAB Function Blocks" on page 17-99). When you simulate this model, the Embedded MATLAB Function block multiplies each input signal by 3 and outputs the result as a frame.

Adding a Channel to a Frame-Based Signal

In the following example, an Embedded MATLAB Function block adds a channel to a frame-based single-channel input and outputs the multichannel result. The input signal is a sine wave that contains 8 samples per frame. Here is the model:



```
function y = fcn(u)
a = [0;4;0;-4;0;4;0;-4];
y = [u a];
```

In the Embedded MATLAB Function block, input u and output y inherit their size, complexity, and data type from the sine wave signal, an 8-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set its sampling mode to **Frame based** (see "Adding

Frame-Based Data in Embedded MATLAB Function Blocks" on page 17-99). Local variable a defines a second column on the matrix which will be output as a frame and interpreted as a second channel by downstream blocks. When you simulate this model, the Embedded MATLAB Function block outputs the new multichannel signal.

18

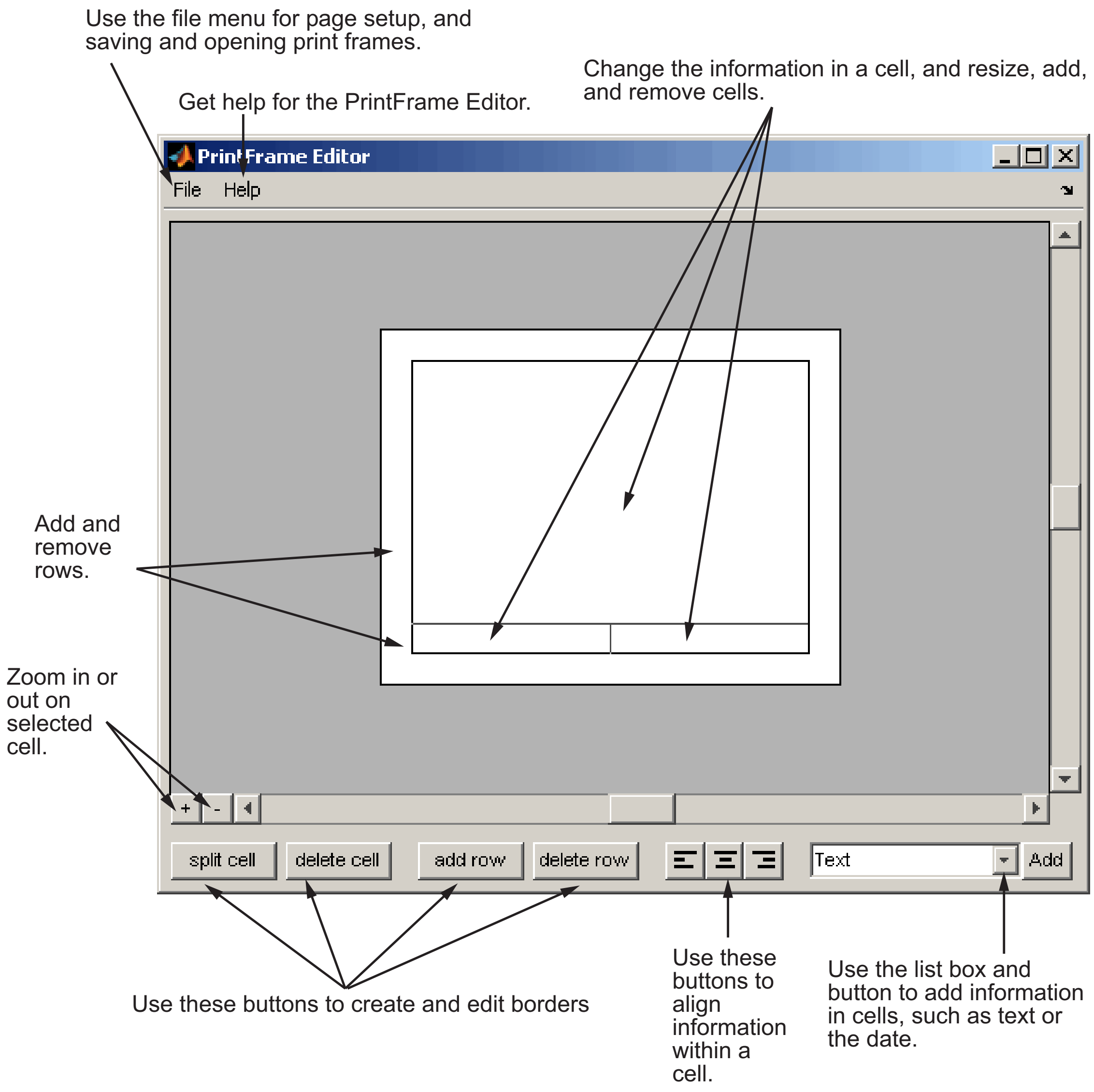
PrintFrame Editor

The PrintFrame Editor is a graphical user interface you use to create and edit print frames for Simulink and Stateflow block diagrams.

PrintFrame Editor Overview

The PrintFrame Editor is a graphical user interface you use to create and edit print frames for Simulink and Stateflow block diagrams. This chapter outlines the PrintFrame Editor, accessible with the `frameedit` command.

The following figure describes the general layout of the PrintFrame Editor.



Use the file menu for page setup, and saving and opening print frames.

Get help for the PrintFrame Editor.

Change the information in a cell, and resize, add, and remove cells.

Add and remove rows.

Zoom in or out on selected cell.

Use these buttons to create and edit borders

Use these buttons to align information within a cell.

Use the list box and button to add information in cells, such as text or the date.

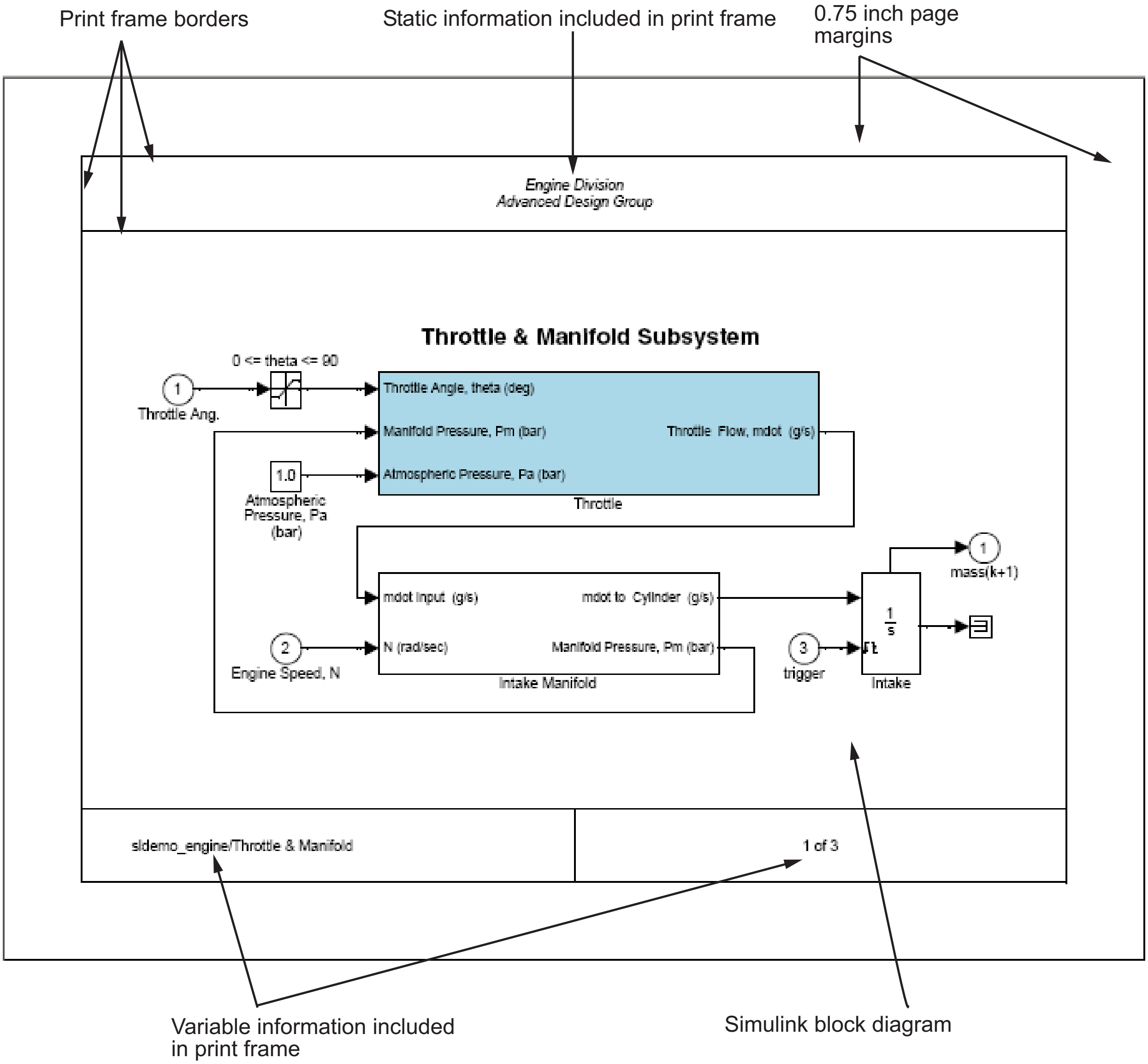
The following topics provide more information about PrintFrames:

- "What PrintFrames Are" on page 18-4
- "Starting the PrintFrame Editor" on page 18-6
- "Getting Help for the PrintFrame Editor" on page 18-7
- "Closing the PrintFrame Editor" on page 18-7
- "Print Frame Process" on page 18-7

What PrintFrames Are

Print frames are borders containing information relevant to the block diagram, for example, the name of the block diagram. After creating a print frame, you can use Simulink or Stateflow to print a block diagram with a print frame.

This illustration shows an example of a print frame with the major elements labeled.


Print frame borders
Static information included in print frame
0.75 inch page margins
Engine Division
Advanced Design Group
Throttle & Manifold Subsystem
0 <= theta <= 90
1
Throttle Ang.
Throttle Angle, theta (deg)
Manifold Pressure, Pm (bar)
Throttle Flow, mdot (g/s)
1.0
Atmospheric Pressure, Pa (bar)
Atmospheric Pressure, Pa (bar)
Throttle
mdot input (g/s)
mdot to Cylinder (g/s)
2
Engine Speed, N
N (rad/sec)
Manifold Pressure, Pm (bar)
Intake Manifold
3
trigger
1/s
Intake
1
mass(k+1)
sldemo_engine/Throttle & Manifold
1 of 3
Variable information included in print frame
Simulink block diagram

See the “Example” on page 18-25 for specific instructions to create this print frame.

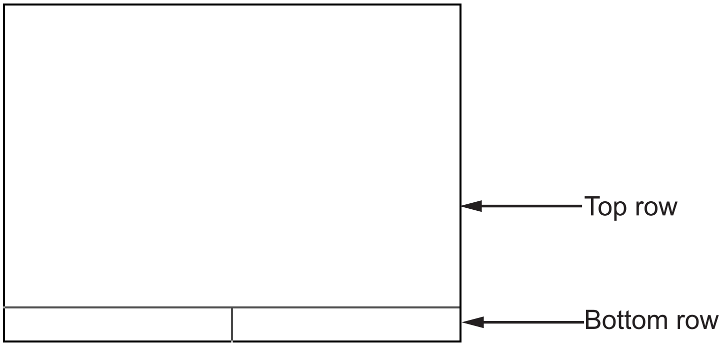
Starting the PrintFrame Editor

Type `frameedit` at the MATLAB prompt. The **PrintFrame Editor** window appears. The **PrintFrame Editor** window opens with the default print frame.

You can use `frameedit filename` to open the **PrintFrame Editor** window with the specified filename, where `filename` is a figure file you previously created and saved using `frameedit`.

Default Print Frame

The default print frame has two rows. The top row consists of one cell and the bottom row has two cells.



You can add information entries to these cells. You can also add new rows and cells and add information in them, or change entries to different ones.

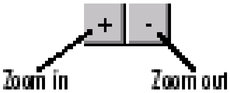
Zooming In and Out

While using the PrintFrame Editor, you might need to zoom in on an area to better see the information or cell.

1 Click in the area you want to zoom in on.

   This selects a cell.

2 Click the zoom in button.



The area is magnified.

3 Click the zoom in button repeatedly to continue zooming in.

To zoom out, reducing magnification in an area, click the zoom out button. Click the zoom out button repeatedly to continue zooming out.

Getting Help for the PrintFrame Editor

Select **PrintFrame Editor Help** from the **Help** menu in the PrintFrame Editor window to access this online help.

Closing the PrintFrame Editor

To close the **PrintFrame Editor** window, click the close box in the upper right corner, or select **Close** from the **File** menu.

Print Frame Process

These are the basic steps for creating and using print frames:

- “Designing the Print Frame” on page 18-8
- “Specifying the Print Frame Page Setup” on page 18-9
- “Creating Borders (Rows and Cells)” on page 18-11
- “Adding Information to Cells” on page 18-13
- “Changing Information in Cells” on page 18-17
- “Saving and Opening Print Frames” on page 18-21
- “Printing Block Diagrams with Print Frames” on page 18-22

See also the “Example” on page 18-25.

Designing the Print Frame

Before you create a print frame using the PrintFrame Editor, consider the type of information you want to include in it and how you want the information to appear. You might want to make a sketch of how you want the print frame to look, and note the wording you want to use.

See the following for more information:

Variable and Static Information

In a print frame, you can include variable and static information. Variable information is automatically supplied at the time of printing, for example, the date the block diagram is being printed. Static information always prints exactly as you entered it, for example, the name and address of your organization.

Single Use or Multiple Use Print Frames

You can design a print frame for one particular block diagram, or you can design a more generic print frame for printing with different block diagrams.

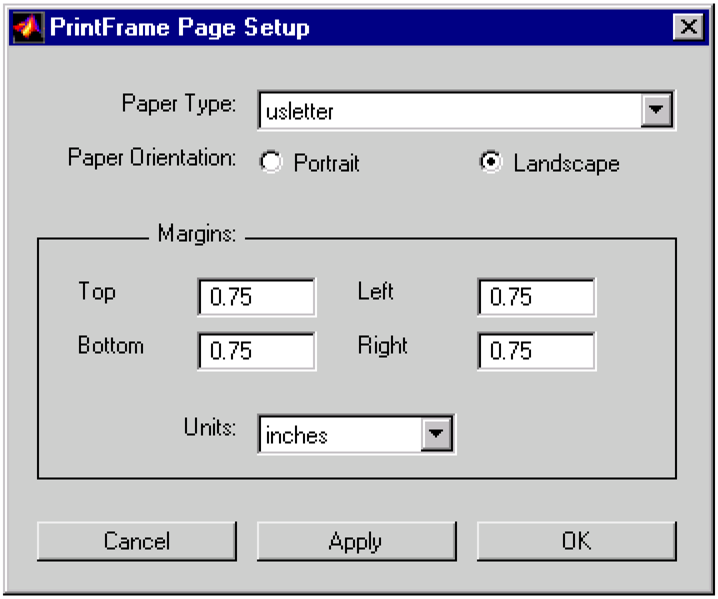
Specifying the Print Frame Page Setup

After you have an idea of the design of your print frame, specify the page setup for the print frame.

Note Always begin creating a new print frame with **PrintFrame Page Setup**. If, instead, you begin by creating borders and adding information, and then later change the page setup, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the page setup paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

1 In the **PrintFrame Editor** window, select **Page Setup** from the **File** menu.

The **PrintFrame Page Setup** dialog box appears.



Paper Type: usletter

Paper Orientation: ○ Portrait ● Landscape

Margins:

Top	0.75	Left	0.75
Bottom	0.75	Right	0.75

Units: inches

2 In the dialog box, specify:

- **Paper Type** – for example, usletter
- **Paper Orientation** – portrait or landscape
- **Margins** for the print frame and the **Units** in which to specify the margins

3 Click **Apply** to see the effects of the changes you made. Then click **OK** to close the dialog box.

Creating Borders (Rows and Cells)

After setting up the page, specify borders (cells) in which the block diagram and information will be placed.

Important Always specify the PrintFrame page setup before creating borders and adding information (see “Specifying the Print Frame Page Setup” on page 18-9). Otherwise, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

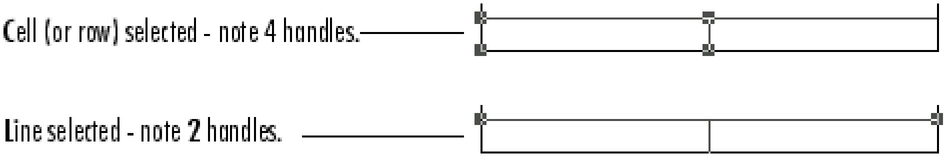
In the PrintFrame Editor, you create borders using rows and then create cells within the rows.

Adding and Removing Rows

You can add and remove rows in a print frame.

1 Click within an existing row to select it. If a row consists of multiple cells, click in any of the cells in the row to select that row.

When a row is selected, handles appear on all four corners. If handles appear on only two corners, you clicked on and only selected the line, not the row.


Cell (or row) selected - note 4 handles.
Line selected - note 2 handles.

2 Click the **add row** button to create a new row.

The new row appears above the row you selected.

3 To remove a row, select the row and click the **delete row** button.

Adding and Removing Cells

You can create multiple cells within a row.

1 Select the row in which you want multiple cells.

2 Click the **split cell** button.

The row splits into two cells. If the row already consists of more than one cell, the selected cell splits into two cells.

3 To remove a cell, select the cell and click the **delete cell** button.

Resizing Rows and Cells

You can change the dimensions of a row or cell.

1 Click on the line you want to move.

A handle appears on both ends of the line.

2 Drag the line to the new location.

For example, to make a row taller, click on the top line that forms the row. Then drag the line up and the height of the row increases.

Print Frame Size

Note that the overall size of the print frame is based on the options you specify using the page setup feature. Therefore, when you change the dimensions of one row or cell, the dimensions of the row or cell next to it change in an inverse direction. For example, if you drag the top line of a row to make it taller, the row above it becomes shorter by the same amount.

To change the overall dimensions of the print frame, use the page setup feature. See “Specifying the Print Frame Page Setup” on page 18-9.

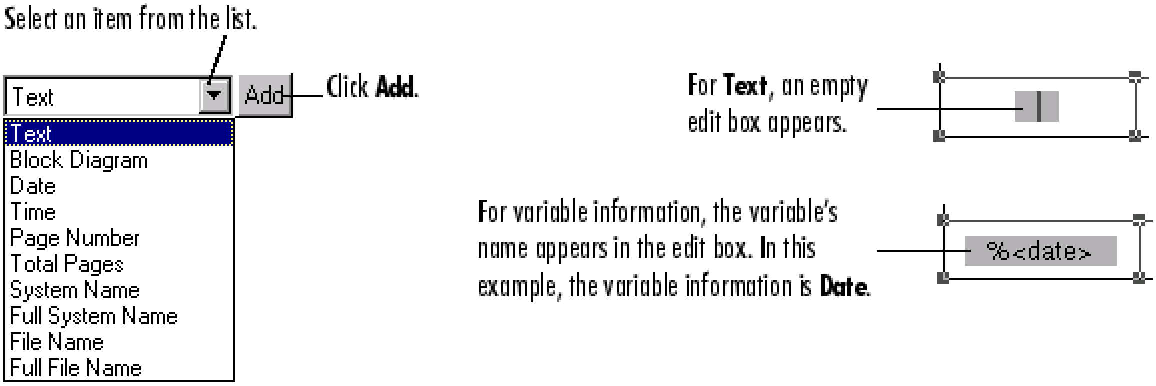
Adding Information to Cells

1 Select the cell where you want to add information.

2 From the list box, select the type of information you want to add.

3 Click the **Add** button.

   An edit box containing that information appears in the cell. (The edit boxes for your platform might look slightly different from those in the figure below.)


Select an item from the list.
Text
Add
Click **Add**.
Text
Block Diagram
Date
Time
Page Number
Total Pages
System Name
Full System Name
File Name
Full File Name
For **Text**, an empty edit box appears.
For variable information, the variable's name appears in the edit box. In this example, the variable information is **Date**.
%<date>

4 Click outside of the edit box to end editing mode.

Note If you click the **Add** button and nothing happens, it might be because you did not select a cell first.

For information on the types of information you can add to cells, see:

- "Text Information" on page 18-14
- "Variable Information" on page 18-14
- "Multiple Entries in a Cell" on page 18-15

Text Information

For **Text**, type the text you want to include in that cell, for example, the name of your organization. Press the **Enter** key if you want to type additional text on a new line. Note that you can type special characters, for example, superscripts and subscripts, Greek letters, and mathematical symbols. For special characters, use embedded TeX sequences (see the `text` command `String` property (in Text Properties of the online MATLAB reference documentation) for a list of allowable sequences). Click outside of the edit box when you are finished to end editing mode.

Variable Information

All of the items in the information list box, except for the **Text** item, are for adding variable information, which is supplied at the time of printing. When you print a block diagram with a print frame that contains variable information, the information for that particular block diagram prints in those fields.

Types of Variable Information

The variable entries you can include are:

- **Block Diagram** — This entry indicates where the block diagram is to be printed. **Block Diagram** is a mandatory entry. If **Block Diagram** is not in one of the cells, you cannot save the print frame and therefore cannot print a block diagram with it.
- **Date** — The date that the block diagram and print frame are printed, in `dd-mmm-yyyy` format, for example, `05-Dec.-1997`.
- **Time** — The time that the block diagram and print frame are printed, in `hh:mm` format, for example, `14:22`.
- **Page Number** — The page of the block diagram being printed.
- **Total Pages** — The total number of pages being printed for the block diagram, which depends on the printing options specified.
- **System Name** — The name of the block diagram being printed.
- **Full System Name** — The name of the block diagram being printed, including its position from the root system through the current system, for example, `engine/Throttle & Manifold`.

- **File Name** — The filename of the block diagram, for example, `sldemo_engine.mdl`.
- **Full File Name** — The full path and filename for the block diagram, for example, `\\matlab\toolbox\simulink\simdemos\automotive\sldemo_engine.mdl`.

Note: Adding the system name or filename does not mean that you can then specify a Simulink or Stateflow system or filename in the PrintFrame Editor. It means that when you print a block diagram from Simulink or Stateflow and specify that it print with a print frame, the system name or filename of that Simulink or Stateflow block diagram prints in the specified cell of the print frame.

Format for Variable Information

When you add a variable entry, a percent sign, `%`, is automatically included to identify the entry as variable information rather than a text string. In addition, the type of entry, for example, `page`, appears in angle brackets, < >. The entry consists of the entire string, for example, `%<page>`, for **Page Number**.

Multiple Entries in a Cell

You can include multiple entries in one cell.

1. Select the cell.
2. Add another item from the list box.

   The new entry is added after the last entry in that cell.

You can also type descriptive text to any of the variable entries without using the **Text** item in the information list box.

1. Double-click in the cell.

   An edit box appears around the entry.

2 Type text in the edit box before or after the entry.

3 Click anywhere outside of the edit box to end editing mode.

Note You cannot include multiple entries or text in the cell that contains the block diagram entry. `%<blockdiagram>` must be the only information in that cell. If there is any other information in that cell, you cannot save the print frame and therefore cannot print it with a block diagram.

Changing Information in Cells

Use these features to change information in cells:

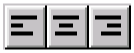
- “Aligning the Information in a Cell” on page 18-17
- “Editing Text Strings” on page 18-17
- “Removing and Copying Entries” on page 18-18
- “Changing the Font Characteristics” on page 18-19

Aligning the Information in a Cell

To align the information within a cell:

1 Click within the cell to select it.

2 Click on one of the **Align** buttons for left, center, or right alignment.



The information aligns within the cell.

Alignment does not apply to the cell that contains the `%<blockdiagram>` entry. The block diagram is automatically scaled and centered to fit in that cell at the time of printing.

Editing Text Strings

You can change text you typed in a cell:

1 Double-click the information you want to edit.

An edit box appears around all of the information in that cell.

2 Click at the start of the text you want to change and drag to the end of the text to be changed.

This highlights the text.

3 Type the replacement text.

It automatically replaces the highlighted text.

4 Click anywhere outside of the edit box to end editing mode.

Note Be careful not to edit the text of a variable entry, because then the variable information will not print. For example, if you accidentally remove the `%` from the `%<page>` entry, the text `<page>` will print instead of the actual page number.

Removing and Copying Entries

You can cut, copy, paste, or delete an entry:

1 Double-click the information you want to remove or copy.

An edit box appears around all of the information in that cell.

2 Click at the start of the entry you want to edit and drag to the end of that entry. This highlights the entry.

For variable information, be sure to include the entire string, for example, `%<page>`.

Note that for Microsoft Windows, you can select all of the entries in a cell by right-clicking the information and choosing **Select All** from the pop-up menu.

3 Use the standard editing techniques for your platform to cut, copy, or delete the highlighted information.

- For Windows, right-click in the edit box and select **Cut**, **Copy**, or **Delete** from the pop-up menu.
- For UNIX, highlighting the information automatically copies it to the clipboard. If you want to remove it, press the **Delete** key.

If you make a mistake, use your platform's standard undo technique. For example, for Windows, right-click in the edit box and select **Undo** from the pop-up menu.

4 If you cut or copied the information to the clipboard and want to paste it, double-click the entry where you want to paste it and position the cursor at the new location in that edit box. Then use the standard paste technique for your platform.

- For Windows, right-click at the new location and select **Paste** from the pop-up menu.
- For UNIX, click at the new location and then click the middle mouse button.

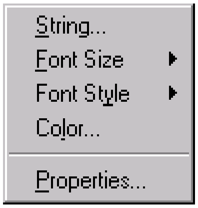
5 Click somewhere outside of the edit box to end editing mode.

Changing the Font Characteristics

You can change the font characteristics for the information in any cell. Specifically, you can specify the font size, style, color, and family.

1 Right-click the information in the cell.

The information in the cell is selected and the pop-up menu for changing font characteristics appears.



If this pop-up menu does not appear, it is because you were in edit mode. To get the font pop-up menu, click somewhere outside of the edit box surrounding the information and then right-click.

2 Select an item from the pop-up menu. Choose **Properties** if you want to change the font family or if you want to change multiple characteristics at once.

Note that you can also select **String** from the pop-up menu, which allows you to edit the text string.

3 Select the new font characteristic(s) for that cell. For example, for **Font Size**, select the new size from its pop-up menu.

Note that changing the font characteristics for the `%<blockdiagram>` entry is not relevant and does nothing.

Saving and Opening Print Frames

This section describes how to save and open print frames.

Saving a Print Frame

You must save a print frame to print a block diagram with that print frame. To save a print frame:

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

   The **Save As** dialog box appears.

2. Type a name for the print frame in the **File name** edit box.

3. Click the **Save** button.

The print frame is saved as a figure file, which has the `.fig` extension. A figure file is a binary file used for print frames.

Opening a Print Frame

You can open a saved print frame in the PrintFrame Editor, make changes to it, and save it under the same or a different name. To open an existing print frame:

1. Select **Open** from the **File** menu.

2. Select the print frame you want to open.

   All print frames are figure files.

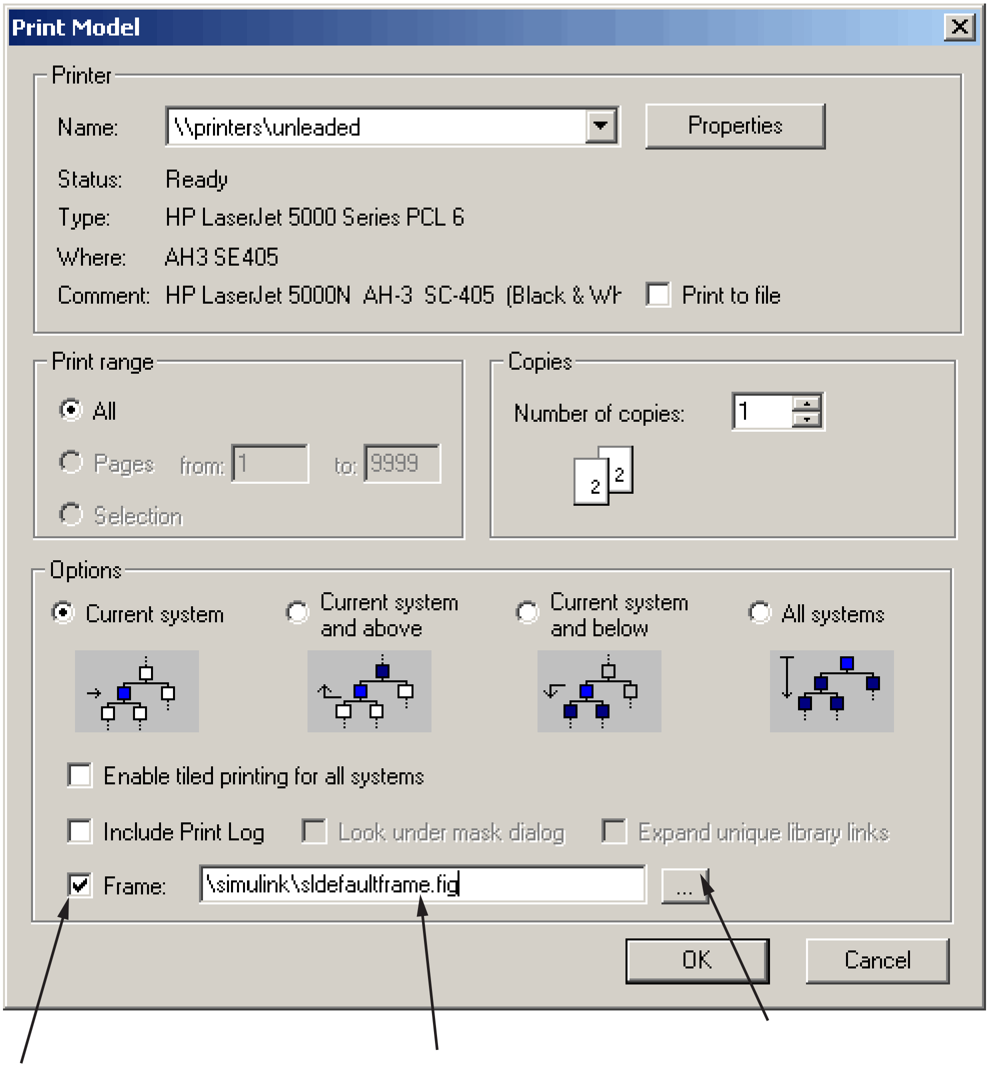
Alternatively, you can open a print frame from the MATLAB prompt. Type `frameedit filename` and the PrintFrame Editor opens with the print frame file you specified.

Printing Block Diagrams with Print Frames

When using Simulink or Stateflow, you can print a block diagram with the print frame.

1 Select **Print** from the Simulink or Stateflow **File** menu.

   The **Print Model** dialog box appears. The dialog box shown below is for the Windows platform. The dialog box for your platform might look slightly different.



Print Model

Printer

Name: \\printers\unleaded

Status: Ready

Type: HP LaserJet 5000 Series PCL 6

Where: AH3 SE405

Comment: HP LaserJet 5000N AH-3 SC-405 (Black & Wh

□ Print to file

Print range

● All

○ Pages from: 1 to: 9999

○ Selection

Copies

Number of copies: 1

Options

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

□ Enable tiled printing for all systems

□ Include Print Log □ Look under mask dialog □ Expand unique library links

☑ Frame: \simulink\sldefaultframe.fig

Check this box to print the block diagram with a print frame.

Type the path and filename of the print frame you want the block diagram to print with,

or

click the ... button and then select the print frame file.

In the **Print Model** dialog box:

1 Select the **Frame** check box.

2 Supply the filename for the print frame you want to use. Either type the path and filename directly in the edit box, or click the **...** button and select the print frame file you saved using the PrintFrame Editor.

Note that the default print frame filename, `sldefaultframe.fig`, appears in the filename edit box until you specify a different filename.

3 Specify other printing options in the **Print** dialog box. For example, for Windows, specify options under **Properties**.

Note Specify the paper orientation for printing the way you normally would. The paper orientation you specified in the PrintFrame Editor's **PrintFrame Page Setup** dialog box is not the same as the paper orientation used for printing. For example, assume you specified a landscape-oriented print frame in the PrintFrame Editor. If you want the printed page to have a landscape orientation, you must specify that at the time of printing. For example, for Windows, click the **Properties** button in the Simulink or Stateflow **Print Model** dialog box, and for **Page Setup**, specify the **Orientation** as **Landscape**.

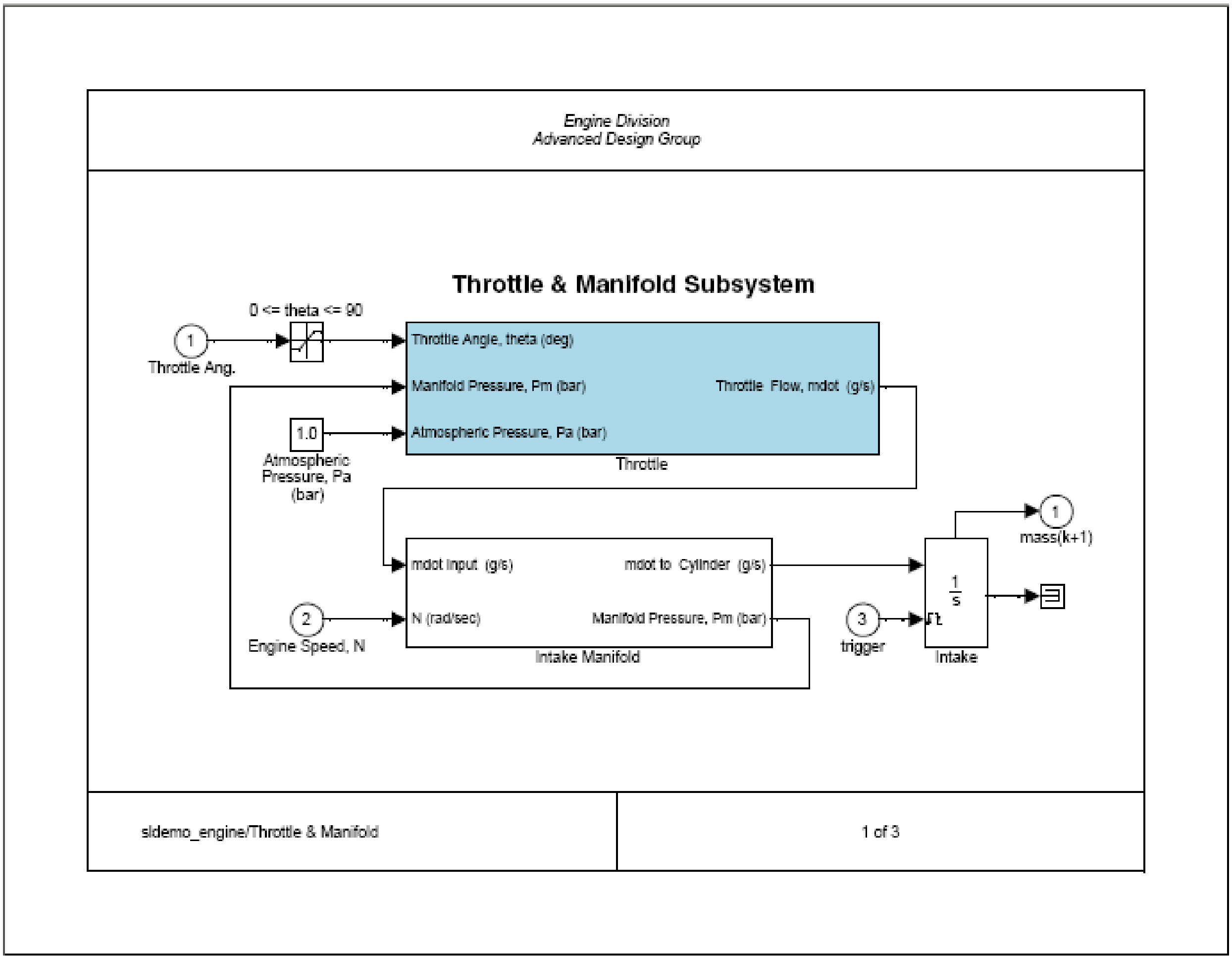
4 Click **OK** in the **Print** dialog box.

The block diagram prints with the print frame you specified.

See also the example, “Print the Block Diagram with the Print Frame” on page 18-28.

Example

This example uses a Simulink demo engine model. It involves two parts — first creating a print frame, and then printing the engine model with that print frame. The result looks similar to the figure below.


Engine Division
Advanced Design Group
Throttle & Manifold Subsystem
0 <= theta <= 90
1
Throttle Ang.
Throttle Angle, theta (deg)
Manifold Pressure, Pm (bar)
Throttle Flow, mdot (g/s)
1.0
Atmospheric Pressure, Pa (bar)
Atmospheric Pressure, Pa (bar)
Throttle
mdot input (g/s)
mdot to Cylinder (g/s)
2
Engine Speed, N
N (rad/sec)
Manifold Pressure, Pm (bar)
Intake Manifold
3
trigger
1/s
Intake
1
mass(k+1)
sldemo_engine/Throttle & Manifold
1 of 3

For more information, see:

- "Create the Print Frame" on page 18-26
- "Print the Block Diagram with the Print Frame" on page 18-28

Create the Print Frame

1. At the MATLAB prompt, type `frameedit`.

   The **PrintFrame Editor** window appears.

2. Set up the page:

   a. Select **Page Setup** from the **File** menu.

      The **PrintFrame Page Setup** dialog box opens.

   b. For **Paper Type**, select a size that is appropriate for your printer.

   c. For this example, keep the **Paper Orientation** as **Landscape** and the **Margins** set to `0.75` **inches**.

   d. Click the **OK** button.

      The dialog box closes. The print frame you see in the **PrintFrame Editor** window will reflect your changes.

3. Add the information entries `%<blockdiagram>`, `%<fullsystem>`, and `%<page>`.

   a. Click within the upper row in the print frame.

   b. From the bottom right list box, select `Block Diagram`, then click **Add**. The `%<blockdiagram>` appears in the row.

   c. Click within the lower left cell in the print frame.

   d. From the bottom right list box, select `Full System Name`, then click **Add**. The `%<fullsystem>` appears in the cell.

   e. Click within the lower right cell in the print frame.

   f. From the bottom right list box, select `Page Number`, then click **Add**. The `%<page>` appears in the cell.

4. Add a row at the top.

- **a** Click within the upper row in the print frame, the row that contains the `%<blockdiagram>` entry.

  Be sure that handles appear on all four corners of the row.
- **b** Click the **add row** button.

  A new row appears at the top, above the row you selected.

5 Make the new row shorter.

- **a** Click on the horizontal line that separates the top row (the row you just added) from the row beneath it (the row containing the `%<blockdiagram>` entry).

  Be sure that only two handles appear, one at each end of the line. If you see four handles in either row, click directly on the horizontal line and the other two handles disappear.
- **b** Drag the line up until the top row is about the same height as the row at the bottom of the print frame.

6 Add information in the top row.

- **a** Click anywhere within the top row (the row you just added).
- **b** Select **Text** from the information list box.
- **c** Click the **Add** button.

  An edit box appears in the cell.
- **d** Type `Engine Division`, press the **Enter** key to advance the cursor to the next line, and then type `Advanced Design Group`.

  Click the zoom in button if you need to magnify the entry.
- **e** Click outside of the edit box to end editing mode.

7 In the left cell of the bottom row, align the information on the left.

- **a** Click the zoom out button if you need to.
- **b** Click within the left cell of the bottom row to select it.
- **c** Click the left alignment button.

The entry moves to the left.

8 Make the information in the top row appear in italics.

   a Right-click on the entry in the row.

   b Select **Font Style** from the pop-up menu.

   If the pop-up menu for font properties does not appear, you are in editing mode. Click outside of the edit box to end editing mode and then right-click the text to access the pop-up menu.

   c From the **Font Style** pop-up menu, select **italic**.

The entry in the cell appears in italics and the information will appear in italics when the print frame is printed with a Simulink diagram.

9 Add the total number of pages to the right cell in the bottom row.

   a Click within the cell to select it.

   b Add the total pages entry: select **Total Pages** from the information list box and click the **Add** button.

   The `%<npages>` entry appears after the `%<page>` entry. If you need to, zoom in to see the entry.

   c Add the text `of` after the page number entry. Click the cursor after the `%<page>` entry, and then type `of` (type a space before and after the word).

The information in the cell now is: `%<page> of %<npages>`.

10 Save the print frame: select **Save As** from the **File** menu. In the **Save Frame** dialog box, type `engdivl` for the **File name**. Click the **Save** button.

The print frame is saved as a figure file.

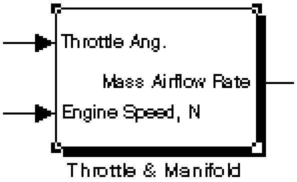
11 You can close the **PrintFrame Editor** window by clicking the close box.

Print the Block Diagram with the Print Frame

1 To view the Simulink engine model, type `sldemo_engine` at the MATLAB prompt.

The engine model appears in a Simulink window.

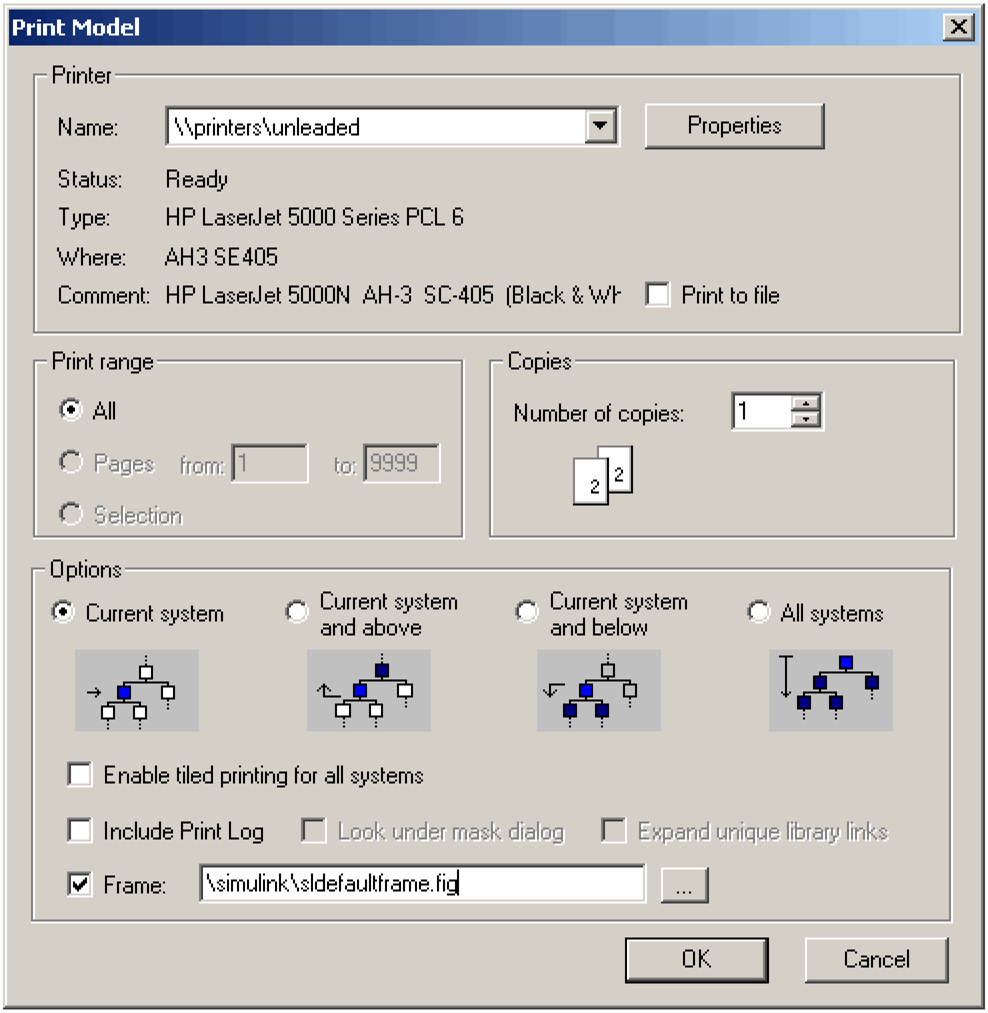
2 Double-click the Throttle & Manifold block.



The Throttle & Manifold subsystem opens in a new window.

3 In the **Throttle & Manifold** window, select **Print** from the **File** menu.

The **Print Model** dialog box opens with the default settings as shown here.



Print Model

Printer

Name: \\printers\unleaded

Status: Ready

Type: HP LaserJet 5000 Series PCL 6

Where: AH3 SE405

Comment: HP LaserJet 5000N AH-3 SC-405 (Black & Wh

□ Print to file

Print range

● All

○ Pages from: 1 to: 9999

○ Selection

Copies

Number of copies: 1

Options

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

□ Enable tiled printing for all systems

□ Include Print Log □ Look under mask dialog □ Expand unique library links

☑ Frame: \simulink\sldefaultframe.fig

4 In the **Print Model** dialog box, set the page orientation to landscape. This example uses the techniques for the Windows platform. Use the methods for your own platform to change the page orientation for printing.

a Click the **Properties** button.

The **Document Properties** dialog box opens.

- **b** Go to the **Page Setup** tab.
- **c** For **Orientation**, select **Landscape**.
- **d** Click **OK**.

  The **Document Properties** dialog box closes.

5 In the **Print Model** dialog box, under **Options**, select **Current system and below**.

This specifies that the Throttle & Manifold block diagram and its subsystems will print.

6 Check the **Frame** check box.

7 Specify the print frame to use.

- **a** Click the **...** button.
- **b** In the **Frame File Selection** dialog box, find the filename of the print frame you just created, `engdivl.fig`, and select it.
- **c** Click the **Open** button.

  The path and filename appear in the **Frame** edit box.

8 Click **OK** in the **Print Model** dialog box.

The Throttle & Manifold block diagram prints with the print frame; it should look similar to the figure shown at the start of this example.

In addition, the Throttle block diagram and the Intake Manifold block diagram print because you specified printing of the current system and its subsystems. These block diagrams also print with the `engdivl` print frame, but note that their variable information in the print frame is different.

Examples

Use this list to find examples in the documentation.

How Simulink Works

Simulink Basics

Creating a Model

Working with Blocks

Working with Lookup Tables

"Example of a Logarithm Lookup Table" on page 8-35

Index