Simulink® 7
Using Simulink

MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science
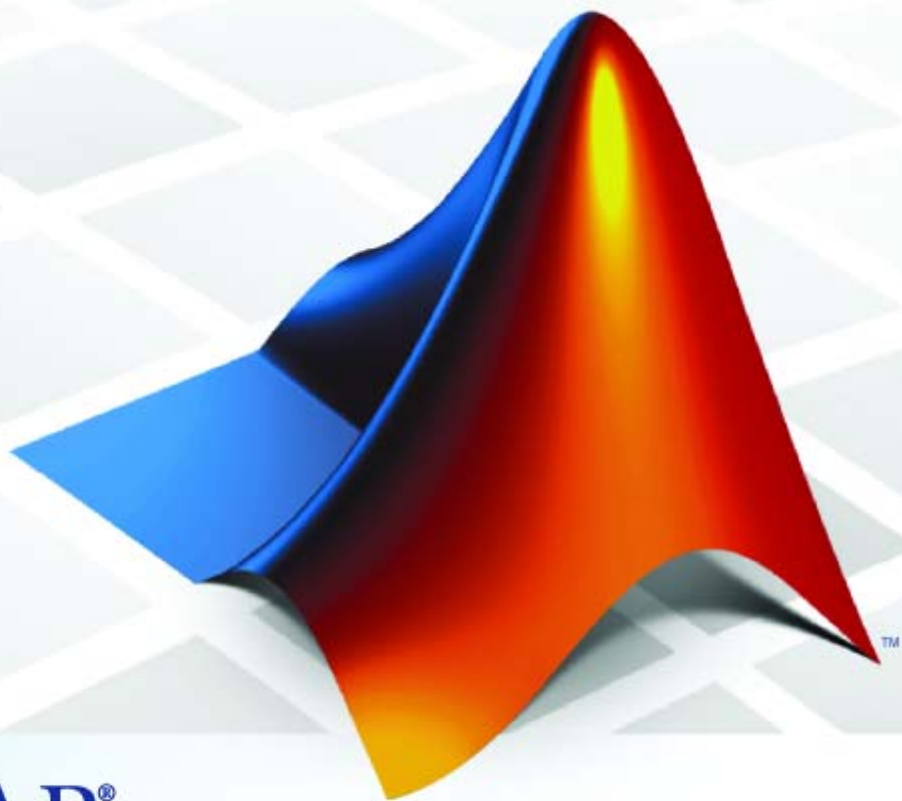
How to Contact The MathWorks

`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®

© COPYRIGHT 1990–2008 by The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Contents

Simulink® Basics

1

How Simulink® Works

2

Creating a Model

3

Creating Conditional Subsystems

4

Referencing a Model

5

Working with Blocks

6

Working with Block Libraries

7

<div align="right">

Working with Signals

</div>

8

Using Composite Signals

9

Working with Data

10

Working with Lookup Tables

11

Modeling with Simulink®

12

Exploring, Searching, and Browsing Models

13

Running Simulations

14

Visualizing Simulation Results

15

Analyzing Simulation Results

16

Creating Block Masks

17

Simulink® Debugger

18

Accelerating Models

19

Customizing the Simulink® User Interface

20

Creating Custom Blocks

21

Using the Embedded MATLAB™ Function Block

22

PrintFrame Editor

23

Glossary

Examples

A

Index

Simulink® Basics

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

Starting the Simulink® Engine

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

- On the toolbar, click the Simulink icon.

- Enter the `simulink` command at the MATLAB prompt.

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

The Simulink library window displays icons representing the pre-installed block libraries. You can create models by copying blocks from the library into a model window.
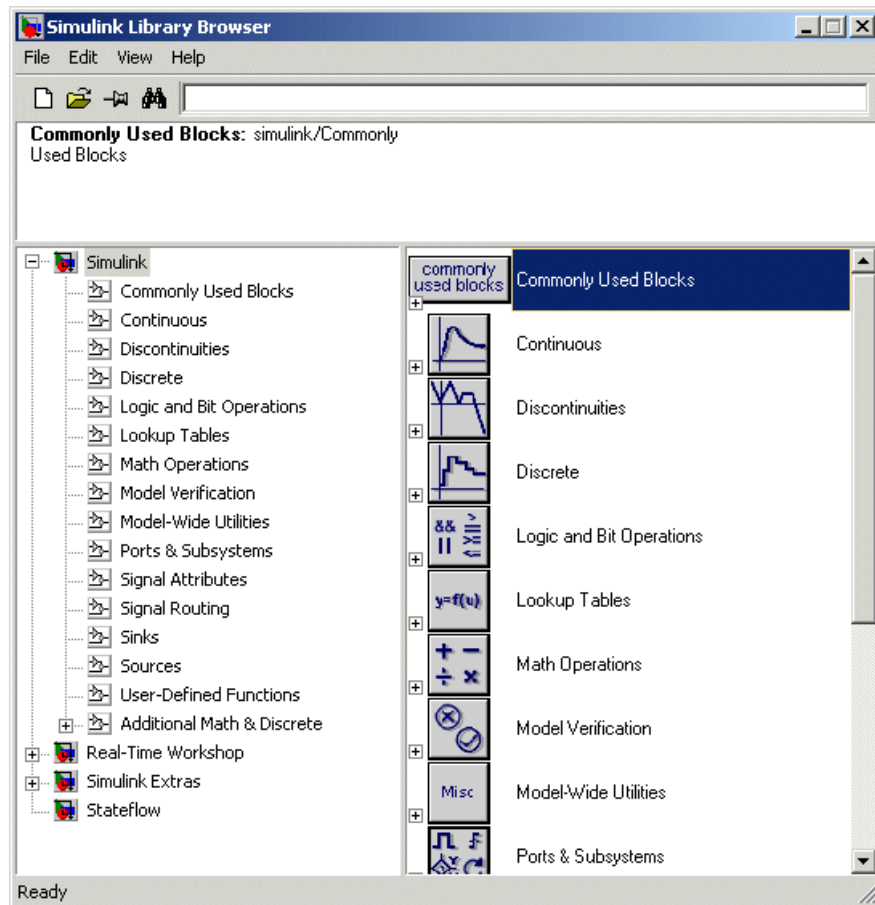
Note On computers running the Windows® operating system, you can display the Simulink library window by right-clicking the Simulink node in the Library Browser window.

Opening Models

In this section...
"Editing an Existing Model" on page 1-4
"Opening Models with Different Character Encodings" on page 1-4
"Avoiding Initial Model Open Delay" on page 1-5

Editing an Existing Model

To edit an existing model diagram, either

- Click the **Open** button on the Library Browser's toolbar (Windows® operating systems only) or select **Open** from the Simulink® library window's **File** menu and then choose or enter the file name for the model to edit.

- Enter the name of the model (without the .mdl extension) in the MATLAB® software Command Window. The model must be in the current directory or on the path.

Note If you have an earlier version of the Simulink software, and you want to open a model that was created in a later version, you must first use the later version to save the model in a format compatible with the earlier version. You can then open the model in the earlier version. See "Saving a Model in Earlier Formats" on page 1-17 for details.

Opening Models with Different Character Encodings

If you open a model created in a MATLAB software session configured to support one character set encoding, for example, Shift_JIS, in a session configured to support another character encoding, for example, US_ASCII,

the Simulink software displays a warning or an error message, depending on whether it can or cannot encode the model, using the current character encoding, respectively. The warning or error message specifies the encoding of the current session and the encoding used to create the model. To avoid corrupting the model (see "Saving Models with Different Character Encodings" on page 1-16) and ensure correct display of the model's text, you should:

1 Close all models open in the current session.

2 Use the `slCharacterEncoding` command to change the character encoding of the current MATLAB software session to that of the model as specified in the warning message.

3 Reopen the model.

You can now safely edit and save the model.

Avoiding Initial Model Open Delay

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

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

Similarly, the following command loads the Simulink software when the MATLAB software starts up on UNIX® systems, systems:

```
matlab -r load_simulink
```

Model Editor

Opening the Simulink® Model or Library

When you open a Simulink® model or library, the model or library is displayed in an instance of the Model Editor.
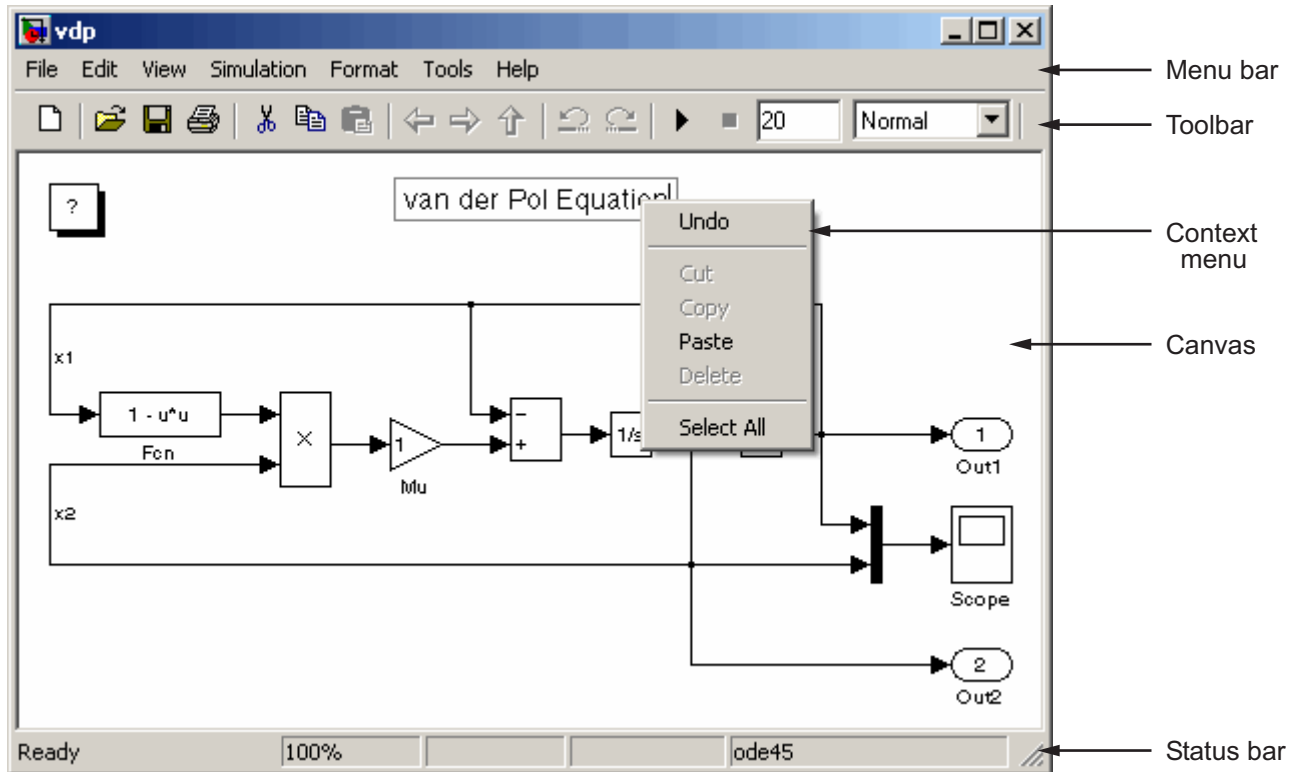
Editor Components

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 "Creating a Model" and "Running Simulations" for more information.

Toolbar

The toolbar allows you to execute the most frequently used commands with a click of a mouse button. For example, to open a Simulink software 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 **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 "Working with Blocks" for more information.

Context Menus

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

Status Bar

The status bar appears only in the Windows® operating system version of the Model Editor. 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 **View** menu.

Undoing a Command

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

• Adding, deleting, or moving a block

• Adding, deleting, or moving a line

• Adding, deleting, or moving a model annotation

- Editing a block name

- Creating a subsystem (see "Undoing Subsystem Creation" for more information)

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

Zooming Block Diagrams

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

- Select **Zoom In** from the **View** menu (or type **r**) to enlarge the view.

- Select **Zoom Out** from the **View** menu (or type **v**) to shrink the view.

- Select **Fit System To View** from the **View** menu (or press the space bar) to fit the diagram to the view.

- Select **Normal** from the **View** menu (or type **1**) to view the diagram at actual size.

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

Panning Block Diagrams

You can use your keyboard alone (see "Model Viewing Shortcuts" on page 1-33) 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
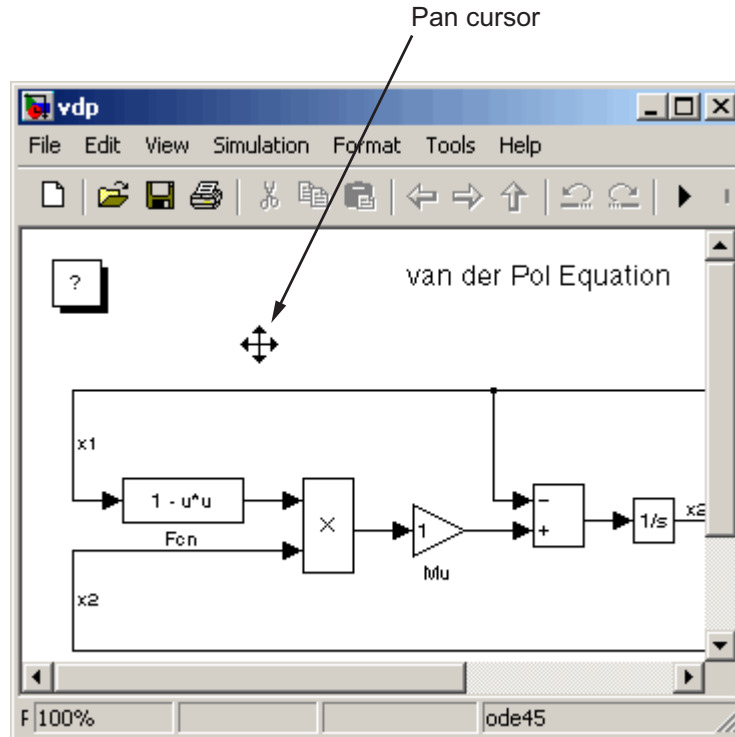
Moving the mouse now pans the model diagram in the editor window.

View Command History

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

- **Back** (⇐) — Displays the previous view in the view history.

- **Forward** (⇒) — Displays the next view in the view history.

- **Go To Parent** ( ) — Opens, if necessary, the parent of the current subsystem and brings its window to the top of the desktop.

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

Bringing the MATLAB® Software Desktop Forward

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

Copying Models to Third-Party Applications

On a computer running the Microsoft® Windows operating system, you can copy a Simulink product model to the Windows operating system clipboard, then paste it to a third-party application such as word processing software. The Simulink product allows you to copy a model in either bitmap or metafile format. You can then paste the clipboard model to an application that accepts figures in bitmap or metafile format. See "Exporting to the Windows or Macintosh® Clipboard" for a description of how to set up the figure settings and save a figure to the clipboard.

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

1 Set the figure copying options.

   a Select **File > Preferences**. The Preferences dialog box appears.

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

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

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

d Click **OK**.

2 Click **OK**.

3 Open the vdp model.

4 In the Model Editor, select **Edit > Copy Model to Clipboard**.

5 Open a document in Microsoft Word and paste the contents of the clipboard.

Updating a Block Diagram

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

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

For example:

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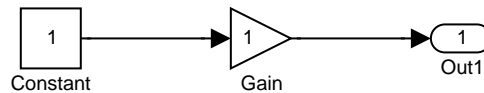
2 Select **Format > Port/Signal Displays > Port Data Types** from the Model Editor's menu bar.

The data types of the output ports of the Constant and Gain blocks are displayed. Note that the data type of both ports is double, the default value.
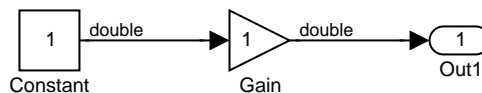
3 Set the **Signal Data Type** parameter of the Constant block (see Constant) to single.

1-13

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**.

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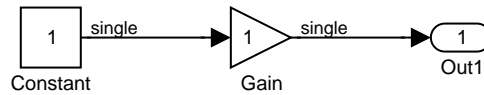
Note that the Simulink software has inferred a data type for the output of the Gain block. This is because you did not specify a data type for the block. The data type inferred is single because single precision is all that is necessary to simulate the model accurately, given that the precision of the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

Saving a Model

In this section...
"About Saving a Model" on page 1-15
"Saving Models with Different Character Encodings" on page 1-16
"Saving a Model in Earlier Formats" on page 1-17
"Opening Models Originally Created in an Older Version of Simulink®" on page 1-19

About Saving a Model

You can save a model by choosing either the **Save** or **Save As** command from the **File** menu. The model is saved by generating a specially formatted file called the *model file* (with the .mdl extension) that contains the block diagram and block properties.

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

If you are saving a model whose model file was previously saved, use the **Save** command to replace the file's contents or the **Save As** command to save the model with a new name or location. You can also use the **Save As** command to save the model in a format compatible with previous releases of the Simulink® product (see "Saving a Model in Earlier Formats" on page 1-17).

The Simulink software follows this procedure while saving a model:

1 If the mdl file for the model already exists, it is renamed as a temporary file.

2 All block PreSaveFcn callback routines are executed first, then the block diagram's PreSaveFcn callback routine are executed.

3 The model file is written to a new file using the same name and an extension of `mdl`.

4 All block `PostSaveFcn` callback routines are executed, then the block diagram's `PostSaveFcn` callback routine is executed.

5 The temporary file is deleted.

If an error occurs during this process, the Simulink software renames the temporary file to the name of the original model file, writes the current version of the model to a file with an `.err` extension, and issues an error message. If an error occurs in step 2, step 3 is omitted and steps 4 and 5 are performed.

Saving Models with Different Character Encodings

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

For example, it's possible you could have introduced characters that cannot be represented in the model's original encoding. If this is the case, the model is saved as **model**.err where **model** is the model's name, leaving the original model file unchanged. The Simulink software also displays an error message that specifies the line and column number of the first 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, find and delete the corresponding character in the open model and resave the model . Repeat this process until you are able to save the model without error.

It's possible that your model's original encoding can represent all the text changes that you've made in the current session, albeit incorrectly. For example, suppose you open a model whose original encoding is A in a session whose current encoding is B. Further, suppose 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, the Simulink software will display x as y. To alert you to the possibility of such corruptions, the software 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 the Simulink software in formats used by earlier versions, 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 the Simulink product.

To save a model in earlier format:

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

2 Select **Save As** from the **File** menu.

The **Save As** dialog box is displayed.
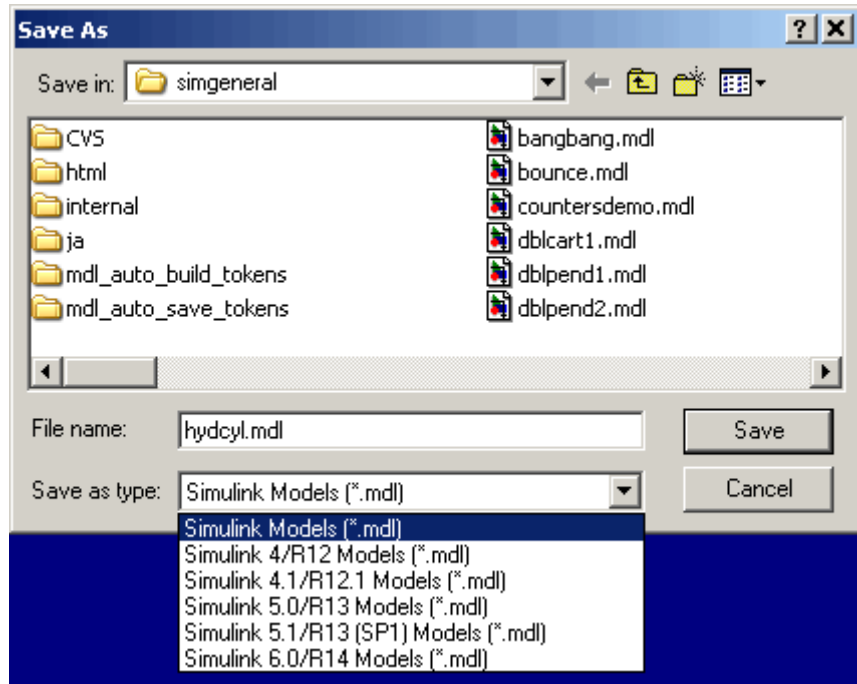
3 Select a format from the **Save as type** list on the dialog box.

4 Click the **Save** button.

When saving a model in an earlier version's format, the model is saved in the earlier format regardless of whether the model contains blocks and features that were introduced after that version. If the model does contain blocks or use features that postdate the earlier version, the model might not give correct results when run by the earlier version. 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 blocksets appear as unlinked blocks.

Opening Models Originally Created in an Older Version of Simulink®

Opening models originally created in an older version of Simulink and then using **Save as** can cause compatibility problems unless they are first saved with the most recent Simulink version and then opened and saved with the older version of Simulink.

Use the following procedure if you wish to open a model created in an older version of Simulink, and wish to save it to some other version:

1 Open the older model with the most recent version of Simulink available to you

2 Before making any changes, use **Save** to save the model in the most recent version of Simulink available to you

3 If you wish to modify or run the model, do so at this step

1-19

4 Use **Save as** to save the model in an older Simulink version. Start the older version of Simulink and use it to open the just saved model

5 While still in the older version of Simulink, but before making any changes or running the model, use **Save** to save the model. You can now run the model in the older version of Simulink.

Printing a Block Diagram

In this section...
"About Printing" on page 1-21
"Print Dialog Box" on page 1-21
"Specifying Paper Size and Orientation" on page 1-23
"Positioning and Sizing a Diagram" on page 1-23
"Tiled Printing" on page 1-24
"Print Command" on page 1-28

About Printing

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

Print Dialog Box

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

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

The portion of the **Print** dialog box that supports selective printing is similar on supported platforms. This figure shows how it looks on a computer running the Microsoft® Windows® operating system. In this figure, only the current system is to be printed.
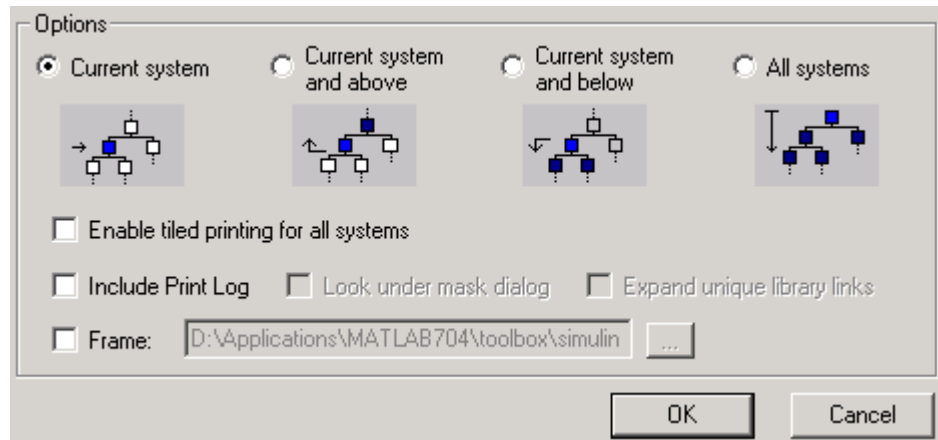
When you select either the **Current system and below** or **All systems** option, two check boxes become enabled. In this figure, **All systems** is selected.
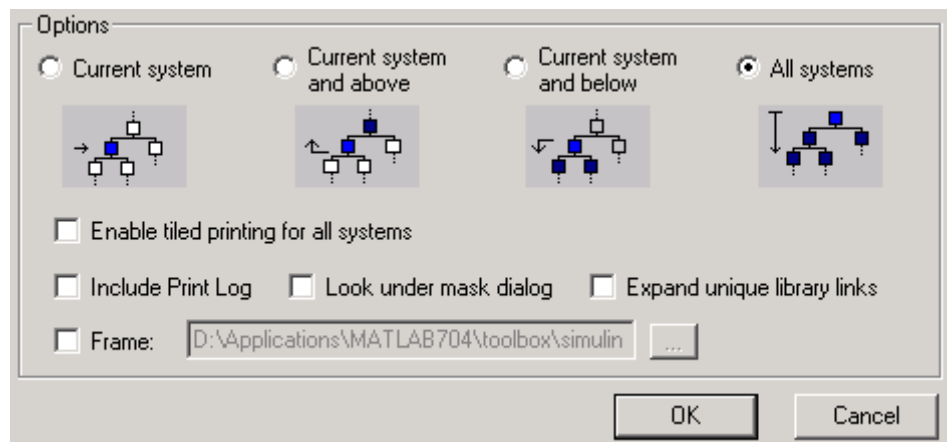
Selecting the **Look under mask dialog** check box prints the contents of masked subsystems when encountered at or below the level of the current block. When you are printing all systems, the top-level system is considered the current block, so the Simulink® software looks under any masked blocks encountered.

Selecting the **Expand unique library links** check box prints the contents of library blocks when those blocks are systems. Only one copy is printed regardless of how many copies of the block are contained in the model. For more information about libraries, see Chapter 7, "Working with Block Libraries".

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

Selecting the **Enable tiled printing for all systems** check box overrides the tiled-print settings for individual subsystems in a model. See "Tiled Printing" on page 1-24 for more information.

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

Specifying Paper Size and Orientation

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

Positioning and Sizing a Diagram

You can use a model's `PaperPositionMode` and `PaperPosition` parameters to position and size the model's diagram on the printed page. The value of the

PaperPosition parameter is a vector of form [left bottom width height]. The first two elements specify the bottom-left corner of a rectangular area on the page, measured from the page's bottom-left corner. The last two elements specify the width and height of the rectangle. When the model's PaperPositionMode is manual, the Simulink software positions (and scales, if necessary) the model's diagram to fit inside the specified print rectangle. For example, the following commands

```
vdp
set_param('vdp', 'PaperType', 'usletter')
set_param('vdp', 'PaperOrientation', 'landscape')
set_param('vdp', 'PaperPositionMode', 'manual')
set_param('vdp', 'PaperPosition', [0.5 0.5 4 4])
print -svdp
```

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

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

Tiled Printing

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

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

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

Enabling Tiled Printing

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

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

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

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

To enable tiled printing for all systems in your model, select the **Enable tiled printing for all systems** check box on the **Print** dialog box (see "Print Dialog Box" on page 1-21). If you select this option, the Simulink software overrides the individual tiled-print settings for all systems in your model.

Displaying Page Boundaries

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

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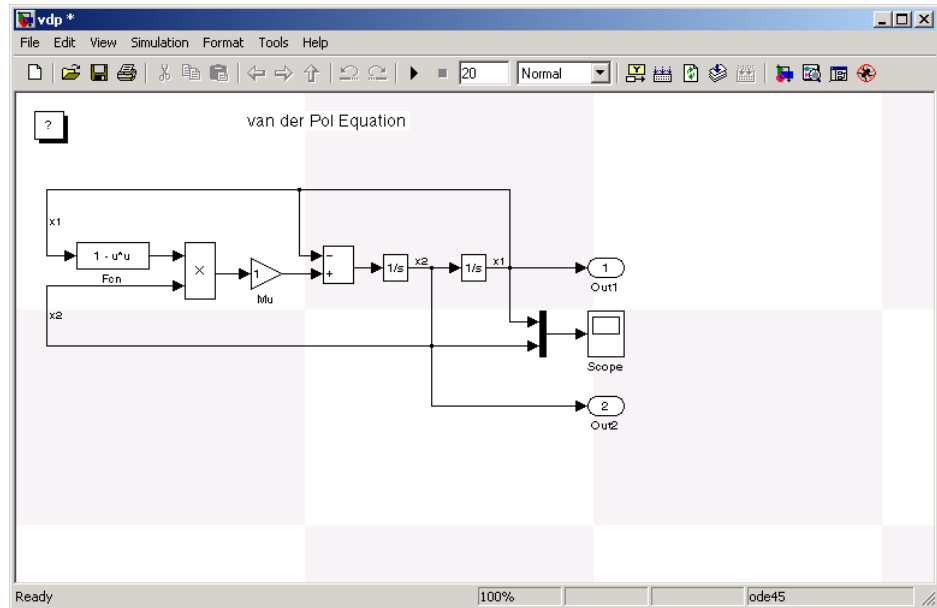
If tiled printing is disabled, only a single page is displayed on the Model Editor's canvas.

Specifying Tiled Print Settings

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

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

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

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

Printing Tiled Pages

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

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

- On a computer running the Microsoft Windows operating system, you can specify a range of tiled page numbers to be printed using the **Print range** portion of the **Print** dialog box. This field is accessible if you select both the **Current system** and **Enable tiled printing for all systems** options (see "Print Dialog Box" on page 1-21).
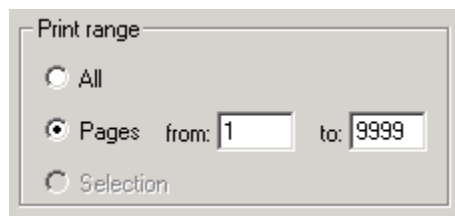
- On all platforms, you can specify a range of tiled page numbers to be printed using the print command at the MATLAB software prompt. The print command's tileall option enables tiled printing for the system, and its pages option indicates the range of tiled page numbers to be printed (see "Print Command" on page 1-28). For example, the following commands

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

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

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

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

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

Print Command

The format of the print command is

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

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

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

tileall specifies the tiled printing option (see "Tiled Printing" on page 1-24).

p is a two-element vector specifying the range of tiled page numbers to be printed. The vector must be preceded by the pages switch identifier. This option is valid only when you enable tiled printing using the tileall switch. For an example of its usage, see "Printing Tiled Pages" on page 1-27.

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

For example, this command prints a system named untitled.

```
print -suntitled
```

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

```
print -sSub1
```

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

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

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

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

To print the currently selected subsystem, enter

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

Generating a Model Report

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

To generate a report for the current model:

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

The **Print Details** dialog box appears.
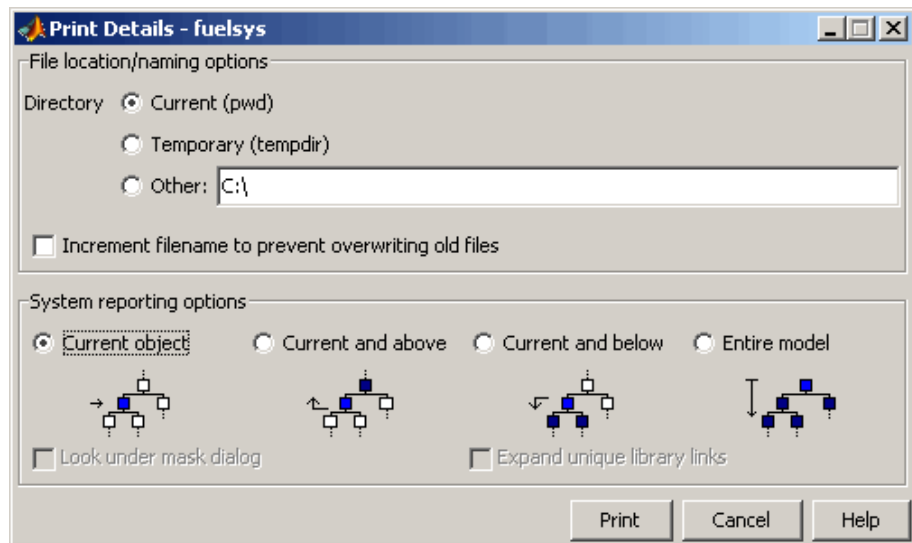
The dialog box allows you to select various report options (see "Model Report Options" on page 1-31).

2 Select the desired report options on the dialog box.

3 Select **Print**.

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

While generating the report, the Simulink software displays status messages on a messages pane that replaces the options pane on the **Print Details** dialog box.
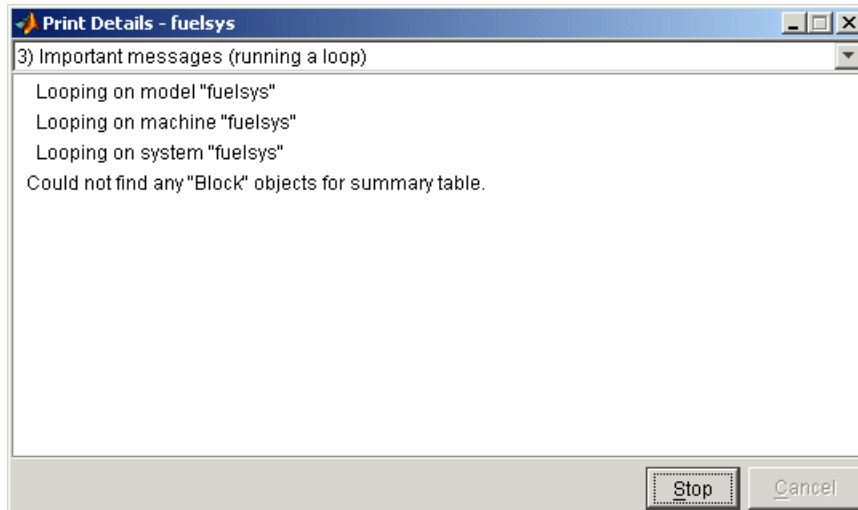
You can select the detail level of the messages from the list at the top of the messages pane. When the report generation process begins, the **Print** button on the **Print Details** dialog box changes to a **Stop** button. Clicking this button terminates the report generation. When the report generation process finishes, the **Stop** button changes to an **Options** button. Clicking this button 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 the HTML report is stored. The options include your system's temporary directory (the default), your system's current directory, or another directory whose path you specify in the adjacent edit field.

1-31

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

In this section...
"Model Viewing Shortcuts" on page 1-33
"Block Editing Shortcuts" on page 1-34
"Line Editing Shortcuts" on page 1-35
"Signal Label Editing Shortcuts" on page 1-35
"Annotation Editing Shortcuts" on page 1-36

Model Viewing Shortcuts

The following table lists keyboard shortcuts for viewing models.

Task	Microsoft® Windows® Operating System	UNIX® System
Zoom in	**r**	**r**
Zoom out	**v**	**v**
Zoom to normal (100%)	**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
Go back in pan/zoom history	**b** or **Shift+Left Arrow**	**b** or **Shift+Left Arrow**

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

Block Editing Shortcuts

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

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

Line Editing Shortcuts

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

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

Signal Label Editing Shortcuts

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

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

Annotation Editing Shortcuts

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

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

Ending a Simulink® Session

Terminate a Simulink® software session by closing all Simulink windows.

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

- On a computer running the Microsoft® Windows® operating system: **Exit MATLAB**

- On a UNIX® system, system: **Quit MATLAB**

2

How Simulink® Works

Introduction

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

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

Modeling Dynamic Systems

Block Diagram Semantics

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

The Simulink® product extends these classic block diagram models by introducing the notion of two classes of blocks, nonvirtual blocks and virtual blocks. Nonvirtual blocks represent elementary systems. 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, blocks and lines can be used to describe many "models of computations." One example would be a flow chart. A flow chart consists of blocks and lines, but one cannot describe general dynamic systems using flow chart semantics.

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

To summarize the meaning of time-based block diagrams:

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

- Signals represent quantities that change over time and are defined for all points in time between the block diagram's start and stop time.

- The relationships between signals and state variables are defined by a set of equations represented by blocks. Each block consists of a set of equations (block methods). These equations define a relationship between the input signals, output signals and the state variables. Inherent in the definition of a equation is the notion of parameters, which are the coefficients found within the equation.

Creating Models

The Simulink product provides a graphical editor that allows you to create and connect instances of block types (see Chapter 3, "Creating a Model") selected from libraries of block types (see) via a library browser. Libraries of blocks are provided representing elementary systems that can be used as building blocks. The blocks supplied with Simulink are called built-in blocks. Users can also create their own block types and use the Simulink editor to create instances of them in a diagram. User-defined blocks are called custom blocks.

Time

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

States

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

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

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

The following is a graphical representation of a block that has states:
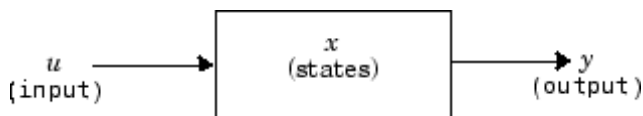
Blocks that define continuous states include the following standard Simulink blocks:

- Integrator
- State-Space
- Transfer Fcn
- Variable Transport Delay
- Zero-Pole

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

Working with States

The following facilities are provided for determining, initializing, and logging a model's states during simulation:

- The `model` command displays information about the states defined by a model, including the total number of states defined by the model, the block that defines each state, and the initial value of each state.

- The Simulink debugger displays the value of a state at each time step during a simulation, and the Simulink debugger's `states` command displays information about the model's current states (see Chapter 18, "Simulink® Debugger").

- The **Data Import/Export** pane of a model's Configuration Parameters dialog box (see "Importing and Exporting Simulation Data" on page 14-21) allows you to specify initial values for a model's states, and to record the values of the states at each time step during simulation as an array or structure variable in the MATLAB® workspace.

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

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

Continuous States

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

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

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

Discrete States

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

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

Modeling Hybrid Systems

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

Block Parameters

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

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

Tunable Parameters

Many block parameters are tunable. A *tunable parameter* is a parameter whose value can be changed without recompiling the model (see "Model Compilation" on page 2-15 for more information on compiling a model). For example, the gain parameter of the Gain block is tunable. You can alter the block's gain while a simulation is running. If a parameter is not tunable and the simulation is running, the dialog box control that sets the parameter is disabled.

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

It should be pointed out that parameter changes do not immediately occur, but are queued up and then applied at the start of the next time step during model execution. Returning to our example of the constant block, the function it defines is `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" 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 2-47 for a description of how a process called sample time propagation is used to determine the sample times of blocks that inherit their sample times.

A block diagram can be optionally color code 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

You can create libraries of custom blocks that you can then use in your models. You can create a custom block either graphically or programmatically. To create a custom block graphically, you draw a block diagram representing the block's behavior, wrap this diagram in an instance of the Simulink Subsystem

block, and provide the block with a parameter dialog, using the Simulink block mask facility. To create a block programmatically, you create an M-file or a MEX-file that contains the block's system functions (see *Writing S-Functions*). The resulting file is called an S-function. You then associate the S-function with instances of the Simulink S-Function block in your model. You can add a parameter dialog to your S-Function block by wrapping it in a Subsystem block and adding the parameter dialog to the Subsystem block. See Chapter 21, "Creating Custom Blocks" for more information.

Systems and Subsystems

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

The Simulink software differentiates between two different types of subsystems: virtual and nonvirtual. The 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, internal "systems" are generated 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. These internal systems are created 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 Chapter 4, "Creating Conditional Subsystems"). 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

The term *signal* refers to a time varying quantity that has values at all points in time. You can specify a wide range of signal attributes, including signal name, data type (e.g., 8-bit, 16-bit, or 32-bit integer), numeric type (real or complex), and dimensionality (one-dimensional, two-dimensional, or multidimensional array). Many blocks can accept or output signals of any data or numeric type and dimensionality. Others impose restrictions on the attributes of the signals they can handle.

On the block diagram, signals are represented with lines that have an arrowhead. The source of the signal corresponds to the block that writes to the signal during evaluation of its block methods (equations). The destinations of the signal are blocks that read the signal during the evaluation of the block's methods (equations).

A good way to understand the definition of a signal is to consider a classroom. The teacher is the one responsible for writing on the white board and the students read what is written on the white board when they choose to. This is also true of Simulink signals: a reader of the signal (a block method) can choose to read the signal as frequently or infrequently as so desired.

For more information about signals, see Chapter 8, "Working with Signals".

Block Methods

Blocks represent multiple equations. These equations are represented as block methods. These block methods are evaluated (executed) during the execution of a block diagram. The evaluation of these block methods is

performed within a simulation loop, where each cycle through the simulation loop represent evaluation of the block diagram at a given point in time.

Method Types

Names are assigned to the types of functions performed by block methods. Common method types include:

• Outputs

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

• Update

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

• Derivatives

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

Method Naming Convention

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

```
BlockType.MethodType
```

For example, the method that computes the outputs of a Gain block is referred to as

```
Gain.Outputs
```

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

```
g1.Outputs
```

Model Methods

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

Simulating Dynamic Systems

In this section...
"Model Compilation" on page 2-15
"Link Phase" on page 2-16
"Simulation Loop Phase" on page 2-16
"Solvers" on page 2-18
"Zero-Crossing Detection" on page 2-20
"Algebraic Loops" on page 2-31

Model Compilation

The first phase of simulation occurs when you choose **Start** from the Model Editor's **Simulation** menu, with the system's model open. This causes the Simulink® engine to invoke the model compiler. The model compiler converts the model to an executable form, a process called compilation. In particular, the compiler

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

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

- A process called attribute propagation is used to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives.

- Performs block reduction optimizations.

- Flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain (see "Solvers" on page 2-18).

- Determines the block sorted order (see "Controlling and Displaying the Sorted Order" on page 6-36 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 2-47).

Link Phase

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

Method Execution Lists

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

Block Priorities

You can assign update priorities to blocks (see "Assigning Block Priorities" on page 6-39). The output methods of higher priority blocks are executed before those of lower priority blocks. The priorities are honored only if they are consistent with its block sorting rules.

Simulation Loop Phase

Once the Link Phase completes, the simulation enters the simulation loop phase. In this phase, the Simulink engine successively computes the states and outputs of the system at intervals from the simulation start time to the finish time, using information provided by the model. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called the step size. The step size depends on the type of solver (see "Solvers" on page 2-18) used to compute the system's continuous states, the system's fundamental sample time (see "Modeling and Simulating Discrete Systems" on page 2-40), and whether the system's continuous states have discontinuities (see "Zero-Crossing Detection" on page 2-20).

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, new values for the system's inputs, states, and outputs are computed, and the model is updated to reflect the computed values. At the end of the simulation, the model reflects the final values of the system's inputs, states, and outputs. The Simulink software provides data display and logging blocks. You can display and/or log intermediate results by including these blocks in your model.

Loop Iteration

At each time step, the Simulink Engine:

1 Computes the model's outputs.

The Simulink Engine initiates this step by invoking the Simulink model Outputs method. The model Outputs method in turn invokes the model system Outputs method, which invokes the Outputs methods of the blocks that the model contains in the order specified by the Outputs method execution lists generated in the Link phase of the simulation (see "Solvers" on page 2-18).

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.

A technique called zero-crossing detection is used to detect discontinuities in continuous states. See "Zero-Crossing Detection" on page 2-20 for more information.

4 Computes the time for the next time step.

Steps 1 through 4 are repeated until the simulation stop time is reached.

Solvers

A dynamic system is simulated by computing its states at successive time steps over a specified time span, using information provided by the model. The process of computing the successive states of a system from its model is known as solving the model. No single method of solving a model suffices for all systems. Accordingly, a set of programs, known as *solvers*, are provided that each embody a particular approach to solving a model. The **Configuration Parameters** dialog box allows you to choose the solver most suitable for your model (see "Choosing a Solver Type" on page 14-11).

Fixed-Step Solvers Versus Variable-Step Solvers

The solvers provided in the Simulink software fall into two basic categories: fixed-step and variable-step.

Fixed-step solvers solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as the step size. You can specify the step size or let the solver choose the step size. Generally,

decreasing the step size increases the accuracy of the results while increasing the time required to simulate the system.

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

Continuous Versus Discrete Solvers

The Simulink product provides both continuous and discrete solvers.

Continuous solvers use numerical integration to compute a model's continuous states at the current time step 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. An extensive set of fixed-step and variable-step continuous solvers are provided, each implementing a specific ODE solution method (see "Choosing a Solver Type" on page 14-11).

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, your selection is disregarded and uses a continuous solver instead when solving the model.

Two discrete solvers are provided: A fixed-step discrete solver and a variable-step discrete solver. The fixed-step solver by default chooses a step size and hence simulation rate fast enough to track state changes in the fastest block in your model. The variable-step solver adjusts the simulation step size to keep pace with the actual rate of discrete state changes in your model. This can avoid unnecessary steps and hence shorten simulation time for multirate models (see "Determining Step Size for Discrete Systems" on page 2-46 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.

States Shape Preservation

Usually the integration step size is only related to the current step size and the current integration error. However, for signals who's derivative changes rapidly more accurate integration results can be obtained by including the derivative input information at each time step. This is done by activating the **State Shapes Preservation** option in the Solver pane of the Configuration Parameter dialog.

Zero-Crossing Detection

A variable-step solver dynamically adjusts the time step size, causing it to increase when a variable is changing slowly and to decrease when the variable changes rapidly. This behavior causes the solver to take many small steps in the vicinity of a discontinuity because the variable is rapidly changing in this region. This improves accuracy but can lead to excessive simulation times.

The Simulink software uses a technique known as *zero-crossing detection* to accurately locate a discontinuity without resorting to excessively small time steps. Usually this technique improves simulation run time, but it can cause some simulations to halt before the intended completion time.

Two algorithms are provided in the Simulink software: Non-Adaptive and Adaptive. For information about these techniques, see "Zero Crossing Algorithms" on page 2-25.

Demonstrating Effects of Excessive Zero Crossing Detection

The Simulink software comes with two demos that illustrate zero crossing behavior.

- Run the bounce demo to see how excessive zero crossings can cause a simulation to halt before the intended completion time.

- Run the doublebounce demo to see how the adaptive algorithm successfully solves a complex system with two distinct zero crossing requirements.

The Bounce Demo.

1 Load the demo by typing `bounce` at the MATLAB® command prompt.

2 Once the block diagram appears, navigate to the **Configuration Parameters** dialog. Confirm that the **Zero crossing location algorithm** is set to `Non-adaptive`.

3 Run the model for a simulation time of 20 seconds.

4 After the simulation completes, click on the scope to display the results.

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

Use the scope zoom controls to closely examine the last portion of the simulation. You can see that the velocity is hovering just above zero at the last time point.

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

6 This time the simulation halts with an error shortly after it passes the simulated 20 second time point.

Excessive chattering as the ball repeatedly approaches zero velocity has caused the simulation to exceed the default limit of 1000 for the number of consecutive zero crossings allowed. Although this limit can be increased

by adjusting the **Number of consecutive zero crossings allowed parameter** in the **Configuration Parameters** dialog, doing so in this case does not allow the simulation to simulate for 25 seconds.

7 Navigate to the **Configuration Parameters** dialog and select the **Adaptive** zero crossing location algorithm from the **Zero crossing location algorithm** pull down.

8 Change the simulation time to 25 seconds, and run the simulation again.

9 This time the simulation runs to completion because the adaptive algorithm prevented an excessive number of zero crossings from occurring.

The Doublebounce Demo.

1 Load the demo by typing `doublebounce` at the MATLAB command prompt.

2 In the demo, click the **Non-adaptive** button. This causes the demo to run with the Non-adaptive zero crossing location algorithm. This is the default setting used by the Simulink software for all models.

3 Notice that the two balls hit the ground and recoil at different times.

4 The simulation halts after 14 seconds because the ball on the left has exceeded the number of zero crossings limit. The ball on the right is left hanging in mid air.

5 Click on the error message to clear it.

6 Click on the **Adaptive** button to run the simulation with the Adaptive zero crossing location algorithm.

7 Notice that this time the simulation runs to completion, even when the ground shifts out from underneath the ball on the left after 20 seconds.

How the Simulator Can Miss Zero Crossing Events

The bounce and doublebounce demos show that high-frequency fluctuations about a discontinuity ('chattering') can cause a simulation to prematurely halt.

It is also possible for the solver to entirely miss zero crossings if the solver error tolerances are too large. This is possible because the zero crossing

detection technique checks to see if the value of a signal has changed sign after a major time step. A sign change indicates that a zero crossing has occurred, and the zero crossing algorithm will then hunt for the precise crossing time. However, if a zero crossing occurs within a time step, but the values at the beginning and end of the step do not indicate a sign change, the solver steps over the crossing without detecting it.

The following figure shows a signal that crosses zero. In the first instance, the integrator steps over the event because the sign has not changed between time steps. In the second, the solver detects change in sign and so detects the zero crossing event.
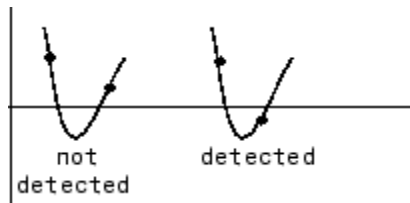
Preventing Excessive Zero Crossings

Use this table to help you prevent excessive zero crossing errors in your model.

Make this change...	How to make this change...	Rational for making this change...
Increase the number of allowed zero crossings	Increase the value of the **Number of consecutive zero crossings allowed**. option on the **Solver** pane in the **Configuration Parameters** dialog.	This may give your model enough time to resolve the zero crossing.

Make this change...	How to make this change...	Rational for making this change...
Relax the zero crossing threshold	Decrease the value of the **Consecutive zero crossings relative tolerance** option on the **Solver** pane in the **Configuration Parameters** dialog.	The solver requires less time to precisely locate the zero crossing. This can reduce simulation time and eliminate excessive number of consecutive zero crossings errors. However, relaxing the zero crossing threshold may reduce accuracy.
Use the **Adaptive** Zero crossing location algorithm	Select **Adaptive** from the **Zero crossing location algorithm** pull down on the **Solver** pane in the **Configuration Parameters** dialog.	This algorithm dynamically adjusts the zero crossing threshold, which improves accuracy and reduces the number of consecutive zero crossings detected. With this algorithm you have the option of specifying a zero crossing tolerance
Disable zero-crossing detection for a specific block	**1** Uncheck the **Enable zero crossing detection** option on the block's parameter dialog box, and **2** Select Use local settings from the **Zero crossing control** pull down on the **Solver** pane of the **Configuration Parameters** dialog box.	Locally disabling zero-crossing detection prevents a specific block from stopping the simulation because of excessive consecutive zero crossings. All other blocks continue to benefit from the increased accuracy that zero-crossing detection provides.

Make this change...	How to make this change...	Rational for making this change...
Disable zero-crossing detection for the entire model	Select `Disable all` from the **Zero crossing control** pull down on the **Solver** pane of the **Configuration Parameters** dialog box.	This prevents zero crossings from being detected anywhere in your model. A consequence is that your model no longer benefits from the increased accuracy that zero-crossing detection provides.
If using the od15s solver, consider adjusting the order of the numerical differentiation formulas	Select a value from the `Maxim order` pulldown in the **Solver** pane of the **Configuration Parameters** dialog box.	For more information, see "Maximum order".
Reduce the maximum step size	Enter a value for the `Max step size` option in the **Solver** pane of the **Configuration Parameters** dialog box.	This can insure the solver takes steps small enough to resolve the zero crossing. However, reducing the step size can increase simulation time, and is seldom necessary when using the Adaptive algorithm.

Zero Crossing Algorithms

The Simulink software includes two zero crossing detection algorithms: Non-Adaptive and Adaptive.

To choose the algorithm, either use the **Zero crossing location algorithm** option in the Solver pane of the Configuration Parameter dialog, or use the ZeroCrossAlgorithm command. The command can either be set to `'Non-adaptive'` or `'Adaptive'`.

The Non-Adaptive algorithm is provided for backwards compatibility with older versions of Simulink and is the default. It brackets the zero crossing event and uses increasingly smaller time steps to pinpoint when the zero crossing has occurred. Although adequate for many types of simulations, the Non-Adaptive algorithm can result in very long simulation times when a high degree of 'chattering' (high frequency oscillation around the zero crossing point) is present.

The Adaptive algorithm dynamically turns the bracketing on and off, and is a good choice when:

- The system contains a large amount of chattering.

- You wish to specify a guard band (tolerance) around which the zero crossing is detected.

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

- The zero crossing error is exceeded. This is determined by the value specified in the **Zero crossing location threshold** option in the Solver pane of the Configuration Parameter dialog. This can also be set with the `ZcDetectionTol` command. The default is `Auto`, but you can enter any real number greater than zero for the tolerance.

- The system has exceeded the number of consecutive zero crossings specified in the **Number of consecutive zero crossings allowed** option in the Solver pane of the Configuration Parameter dialog. Alternatively, this can be set with the `MaxConsecutiveZCs` command.

Understanding Zero Crossing Threshold

The Adaptive algorithm automatically sets a tolerance for zero crossing detection. Alternatively, you can set the tolerance by entering a real number greater than or equal to zero in the Configuration Parameters Solver pane, `Zero crossing location threshold` pull down. This option only becomes active when the Zero crossing algorithm is set to `Adaptive`.

This graphic shows how the Zero crossing threshold sets a window region around the zero crossing point. Signals falling within this window are considered as being at zero.
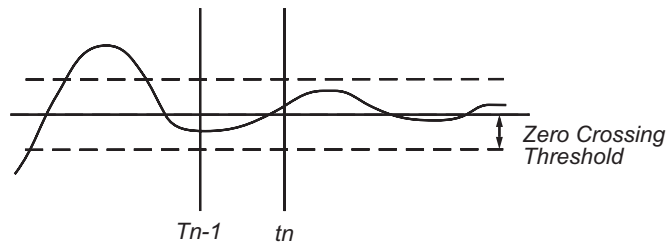
The zero crossing event is bracketed by time steps T_{n-1} and T_n. The solver iteratively reduces the time steps until the state variable lies within the band defined by the zero crossing threshold, or until the number of consecutive zero crossings equals or exceeds the value in the Configuration Parameters Solver pane, `Number of consecuitive zero crossings allowed` pull down.

It is evident from the figure that increasing the zero crossing threshold increases the distance between the time steps which will be executed. This often results in faster simulation times, but could reduce accuracy.

How Blocks Work with Zero-Crossing Detection

A block can register a set of zero-crossing variables, each of which is a function of a state variable that can have a discontinuity. The zero-crossing function passes through zero from a positive or negative value when the corresponding discontinuity occurs. The registered zero-crossing variables are updated at the end of each simulation step, and any variable that has changed sign is identified as having had a zero crossing event.

If any zero crossings are detected, the Simulink software interpolates between the previous and current values of each variable that changed sign to estimate the times of the zero crossings (that is, the discontinuities).

Blocks that Register Zero Crossings. The following table lists blocks that register zero crossings and explains how the blocks use the zero crossings:

Block	Description of Zero Crossing
Abs	One: to detect when the input signal crosses zero in either the rising or falling direction.
Backlash	Two: one to detect when the upper threshold is engaged, and one to detect when the lower threshold is engaged.
Compare To Zero	One: to detect when the signals equals zero.
Dead Zone	Two: one to detect when the dead zone is entered (the input signal minus the lower limit), and one to detect when the dead zone is exited (the input signal minus the upper limit).
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
Stateflow®	One: to detect if there is a valid state transition

Block	Description of Zero Crossing
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.

Implementation Example: Saturation Block. An example of a Simulink block that registers zero crossings is the Saturation block. Zero crossing detection identifies these state events in the Saturation block:

- The input signal reaches the upper limit.

- The input signal leaves the upper limit.

- The input signal reaches the lower limit.

- The input signal leaves the lower limit.

Simulink blocks that define their own state events are considered to have *intrinsic zero crossings*. Use the Hit Crossing block to receive explicit notification of a zero-crossing event. See "Blocks that Register Zero Crossings" on page 2-27 for a list of blocks that incorporate zero crossings.

The detection of a state event depends on the construction of an internal zero-crossing signal. This signal is not accessible by the block diagram. For the Saturation block, the signal that is used to detect zero crossings for the upper limit is zcSignal = UpperLimit - u, where u is the input signal.

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

- *rising* — A zero crossing occurs when a signal rises to or through zero, or when a signal leaves zero and becomes positive.

- *falling* — A zero crossing occurs when a signal falls to or through zero, or when a signal leaves zero and becomes negative.

- *either* — A zero crossing occurs if either a rising or falling condition occurs.

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

Algebraic Loops

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

- Math Function block

- Gain block

- Integrator block's initial condition ports

- Product block

- State-Space block when there is a nonzero D matrix

- Sum block

- Transfer Fcn block when the numerator and denominator are of the same order

- Zero-Pole block when there are as many zeros as poles

An *algebraic loop* generally occurs when an input port with direct feedthrough is driven by the output of the same block, either directly, or by a feedback path through other blocks with direct feedthrough. An example of an algebraic loop is this simple scalar loop.
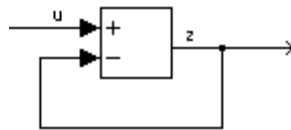
Mathematically, this loop implies that the output of the Sum block is an algebraic state z constrained to equal the first input u minus z (i.e., $z = u - z$). The solution of this simple loop is $z = u/2$, but most algebraic loops cannot be solved by inspection.

It is easy to create vector algebraic loops with multiple algebraic state variables *z1*, *z2*, etc., as shown in this model.
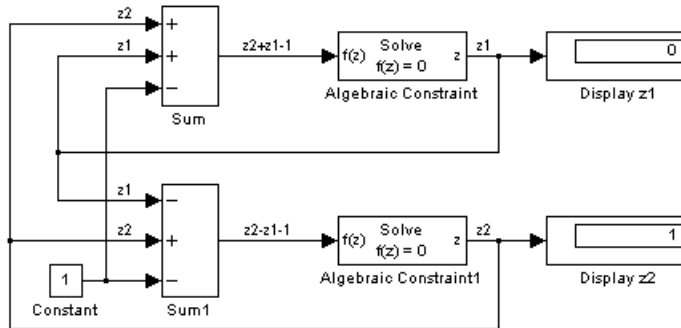
The Algebraic Constraint block is a convenient way to model algebraic equations and specify initial guesses. The Algebraic Constraint block constrains its input signal *F(z)* to zero and outputs an algebraic state *z*. This block outputs the value necessary to produce a zero at the input. The output must affect the input through some direct feedback path, i.e., the feedback path solely contains blocks with direct feedthrough. You can provide an initial guess of the algebraic state value in the block's dialog box to improve algebraic loop solver efficiency.

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

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

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

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

Highlighting Algebraic Loops

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

For example, the following figure shows the block diagram of the 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

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

- Atomic Subsystem
- Enabled Subsystem
- Model

2-33

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

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

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

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

As an example of the ability of the Simulink software to eliminate algebraic loops, consider the following model.
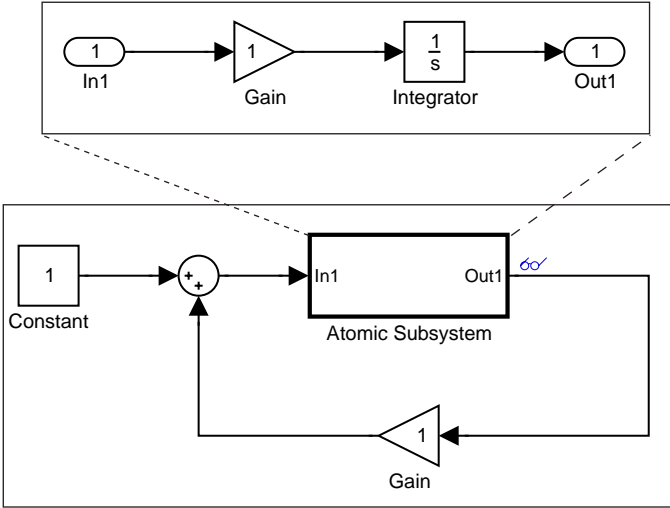
Simulating this model with the solver's Algebraic Loop diagnostic set to error (see "Diagnostics Pane: Solver" for more information) reveals that this model contains an algebraic loop involving its atomic subsystem.
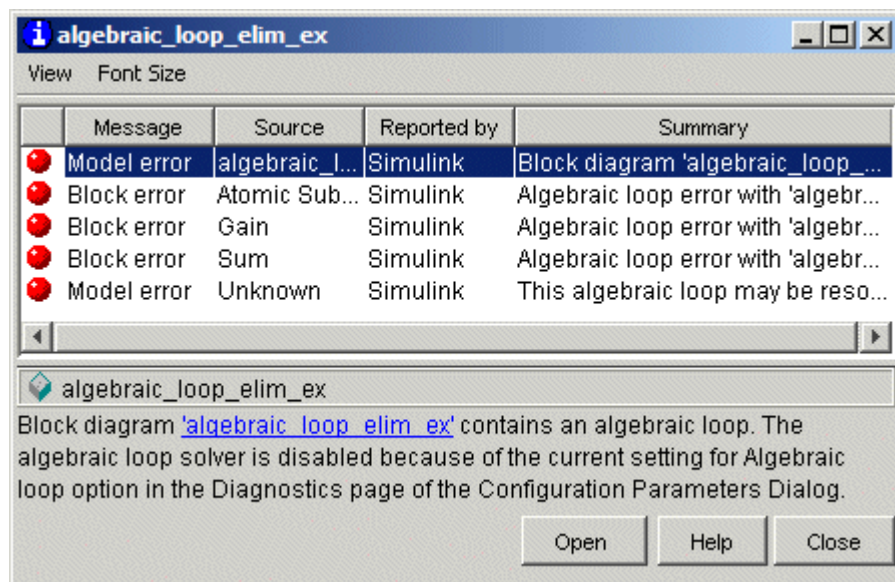
Checking the atomic subsystem's **Minimize algebraic loop occurrences** parameter eliminates the algebraic loop from the compiled version of the model.
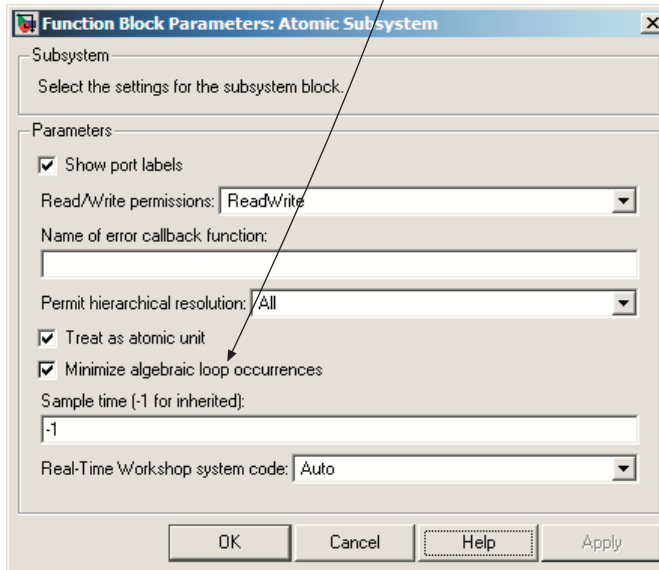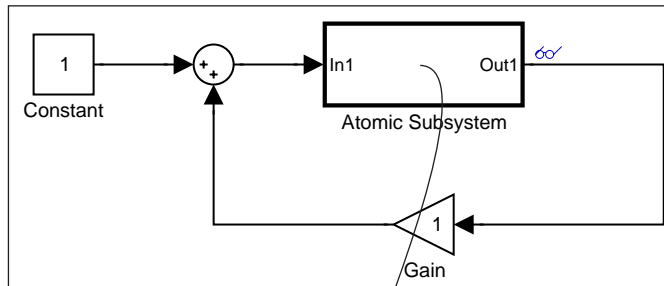
As a result, the model now simulates without error.
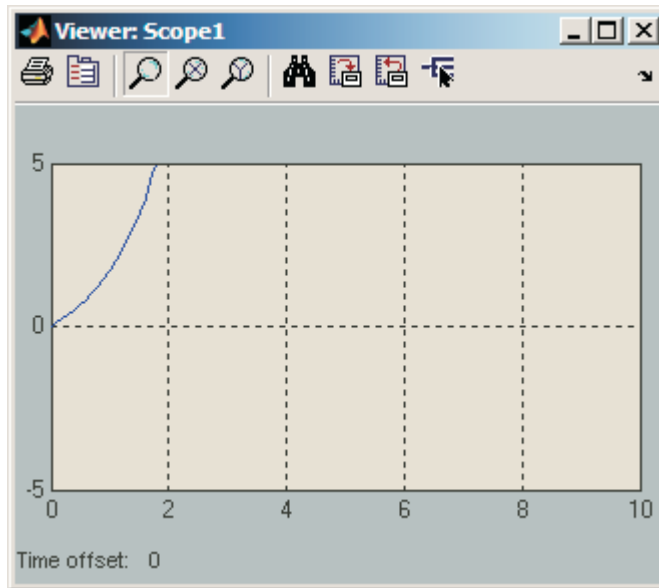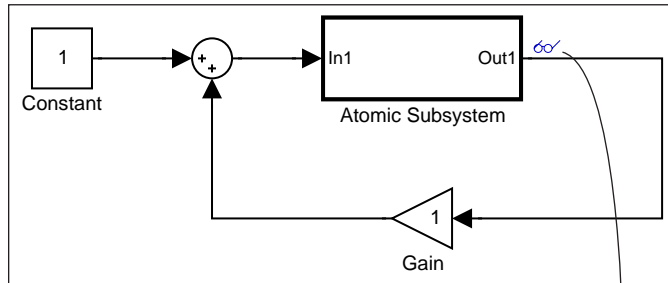
Note that the Simulink software is able to eliminate the algebraic loop involving this model's atomic subsystem because the atomic subsystem contains a block with a port that does not have direct feedthrough, i.e., the Integrator block.

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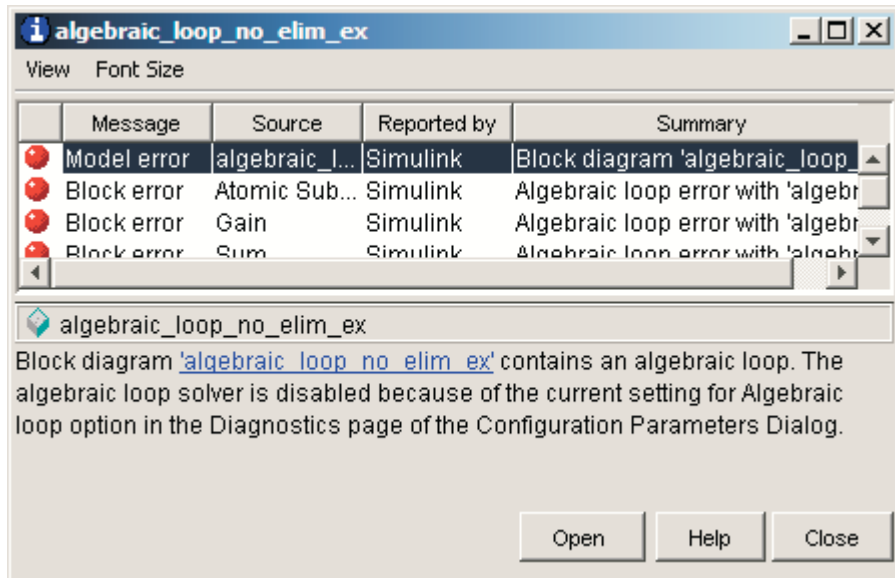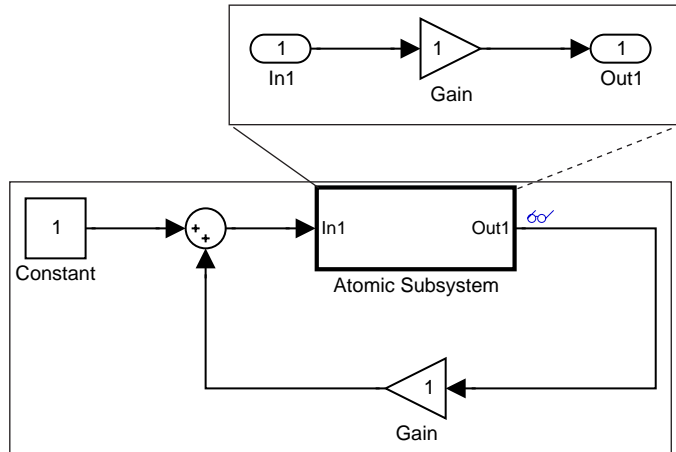
Modeling and Simulating Discrete Systems

Multirate and Hybrid Systems

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

- SampleTime block parameter

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

You can 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 the Simulink software interprets them to determine a block's sample time.

Sample Time	Usage
$[T_s, T_o]$ $0 < T_s < T_{sim}$ $\|T_o\| < T_s$	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 the block be updated 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*.

Sample Time	Usage
[0, 1]	Specifies that updates occur only at major time steps, skipping minor time steps (see "Minor Time Steps" on page 2-20). 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*.
[-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 the Simulink software 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, a warning message is displayed when you update or simulate the model.

Sample Time	Usage
[-2, T_{vo}]	Specifies that a block has a *variable sample time*, that is, computes its output only at times specified by the block. Every block with variable sample time has a unique T_{vo} determined by the Simulink software. The only built-in Simulink block that can have variable sample time is the Pulse Generator block.
inf	The meaning of this sample time depends on whether the active model configuration's inline parameters optimization (see "Inline parameters") 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 2-49). 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, inf is treated 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, the sample time of the block is determined 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.
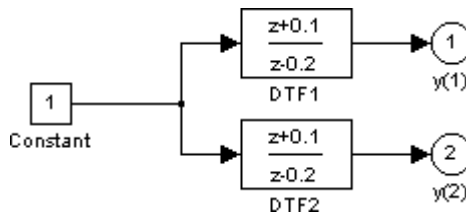
For this example the DTF1 Discrete Transfer Fcn block's **Sample time** is set to [1 0.1], which gives it an offset of 0.1. The DTF2 Discrete Transfer Fcn block's **Sample time** is set to 0.7, with no offset.

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

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

produces this plot
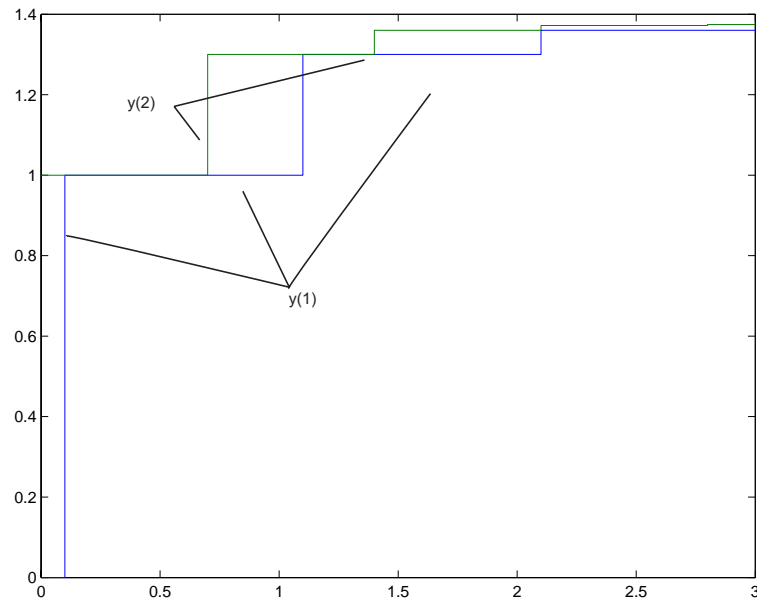
See "Running a Simulation Programmatically" on page 14-74 for information on the `sim` command.

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. This is avoided by choosing a simulation step size to ensure that steps coincide with sample time hits. The step size that the Simulink software 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 the Simulink software 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.
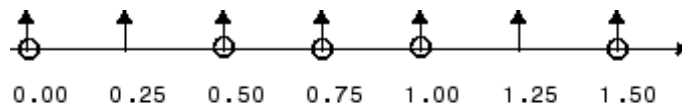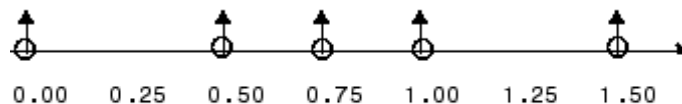
In the diagram, arrows indicate simulation steps and circles represent sample time hits. As the diagram illustrates, a variable-step solver requires fewer simulation steps to simulate a system, if the fundamental sample time is less than any of the actual sample times of the system being simulated. On 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, a process called sample time propagation is used 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.
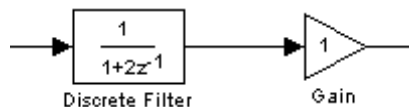
Because the Gain block's output is simply the input multiplied by a constant, its output changes at the same rate as the filter. In other words, the Gain block has an effective sample rate equal to that of the filter's sample rate. This is the fundamental mechanism behind sample time propagation in the Simulink product.

An inherited sample time is assigned 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, that sample time is assigned 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

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), the source block's sample time may be back propagated, 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. This only happens if it can be done without changing the results of simulating the model. For example, in the model below, the Simulink software recognizes that the Sine Wave block is driving a Discrete-Time Integrator block whose sample time is 1, so it assigns the Sine Wave block a sample time of 1.
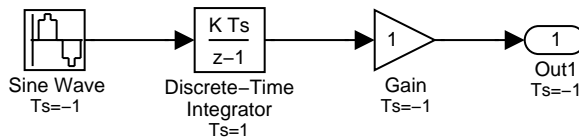
You can verify this by selecting **Sample Time Colors** from the Simulink **Format** menu and noting that all blocks are colored red. Because the Discrete-Time Integrator block only looks at its input at its sample times, this change does not affect the outcome of the simulation but does result in a 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.
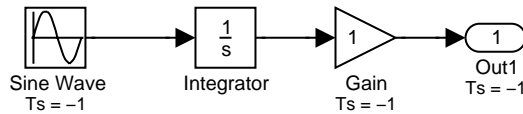
Note Back propagation makes the sample times of a model's sources dependent on block connectivity. If you change the connectivity of a model whose sources inherit sample times, you can inadvertently change the source sample times. For this reason, when you update or simulate a model, by default the Simulink software displays warnings at the command line if the model contains sources that inherit their sample times. See "Source block specifies -1 sample time" 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").

- 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 the Simulink software updates a model, for example, at the beginning of a simulation, it determines which blocks, if any, have constant sample time, and computes the initial values of the output ports. During the simulation, the initial values are used 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.

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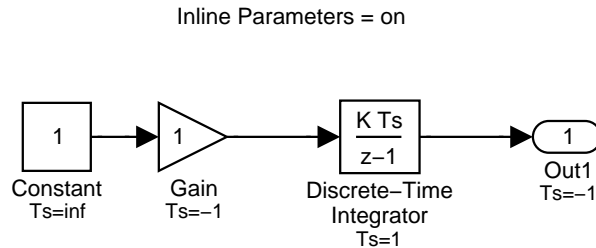
The Gain block has constant sample time because it inherits its sample time from the Constant block and all of the model's parameters are inlined, i.e., nontunable.

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 Blocks with Infinite Sample Times and Tunable Parameters are Treated

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, sample time propagation (see "Sample Time Propagation" on page 2-47) is used to determine the block's actual sample time.

For example, consider the following model.
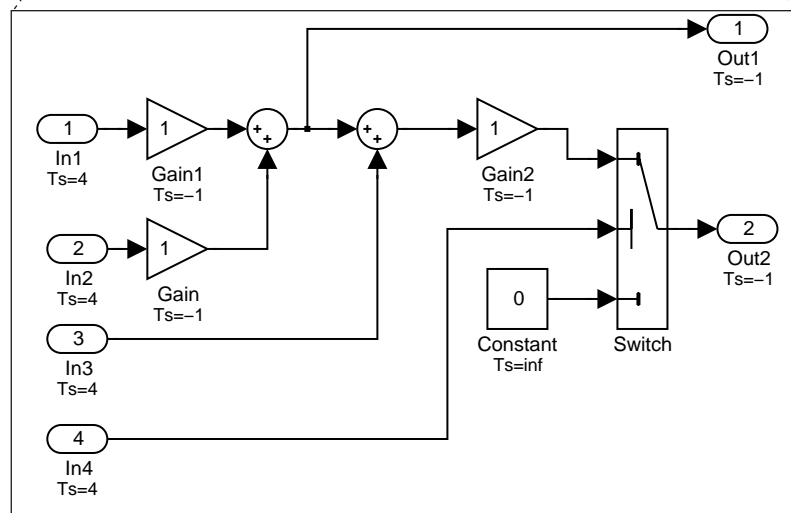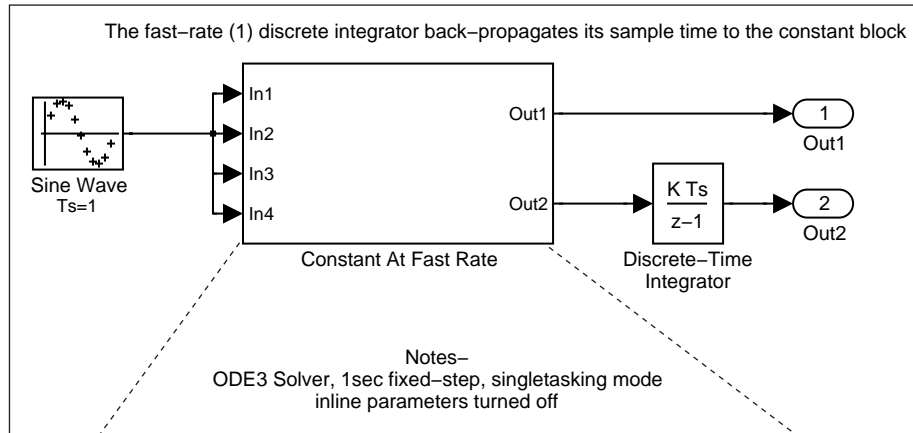
In this example, although the Constant block's sample time is specified to be infinite, it cannot have constant sample time because the inlined parameters option is off for this model and therefore the block's **Constant value** parameter is tunable. Since the Constant block's output can change during the simulation, a sample time for the block has to be determined 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.

3

Creating a Model

Creating an Empty Model

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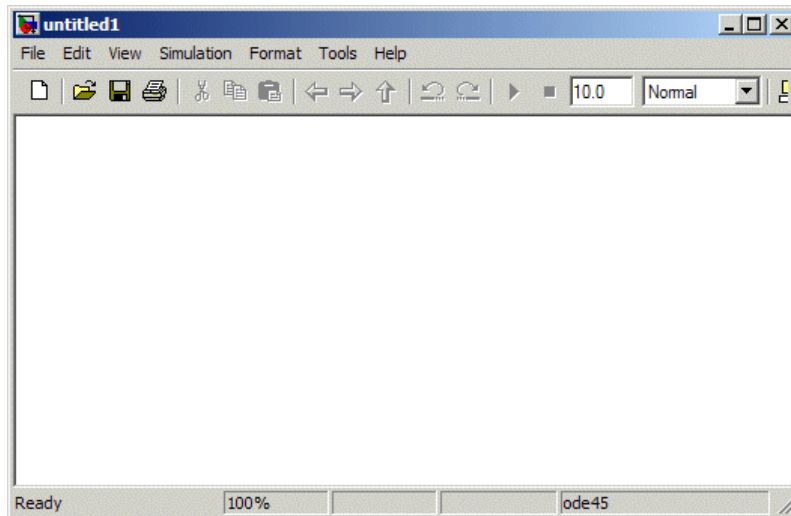
Creating a Model Template

When creating a model, defaults are used for many of 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 the Simulink® software model construction commands (see "Model Construction") to write a function that creates a model with the defaults you prefer.

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

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

```
%      argument, the new model is named 'my_untitled'.

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

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

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

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

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

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

Selecting Objects

In this section...
"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.
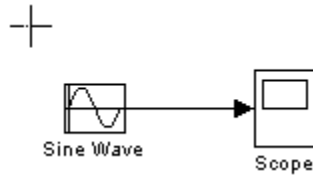
2 Drag the pointer to the opposite corner of the box. A dotted rectangle encloses the selected blocks and lines.
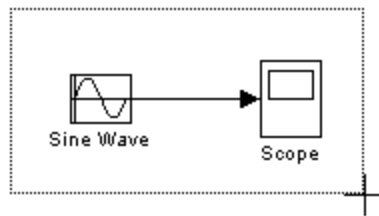
3 Release the mouse button. All blocks and lines at least partially enclosed by the bounding box are selected.
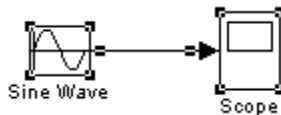
Selecting All Objects

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 3-35.

Specifying Block Diagram Colors

How to Specify Block Diagram Colors

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

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

Choosing a Custom Color

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

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

Defining a Custom Color

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

The dialog box expands to display a custom color definer.
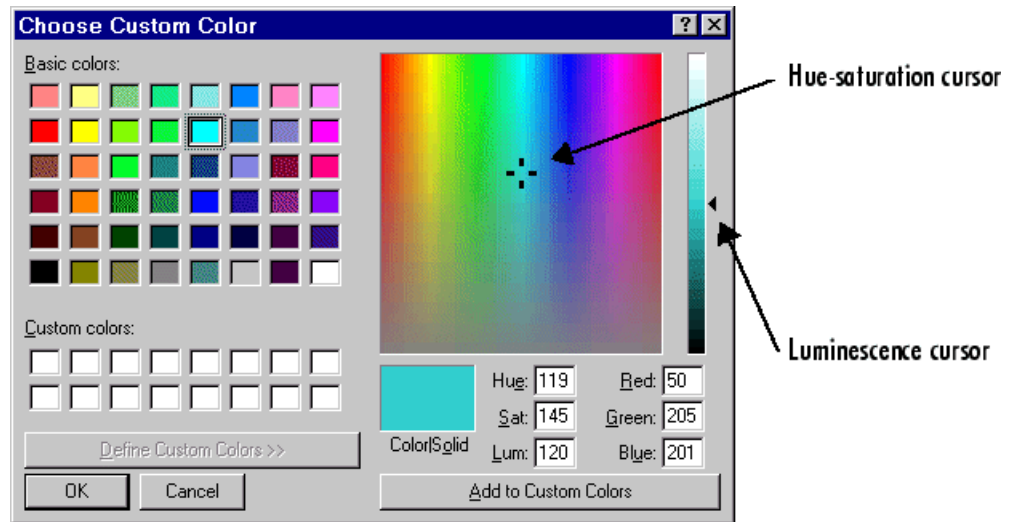
The color definer allows you to specify a custom color by

- Entering the red, green, and blue components of the color as values between 0 (darkest) and 255 (brightest)

- Entering hue, saturation, and luminescence components of the color as values in the range 0 to 255

- Moving the hue-saturation cursor to select the hue and saturation of the desired color and the luminescence cursor to select the luminescence of the desired color

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

Specifying Colors Programmatically

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

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

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

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

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

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

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

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

Displaying Sample Time Colors

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

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

Color	Use
Dark Green	Fifth fastest discrete sample time
Orange	Sixth, seventh, eighth, etc., fastest discrete sample time
Yellow	Indicates a block with hybrid sample time, e.g., subsystems grouping blocks and Mux or Demux blocks grouping signals with different sample times, Data Store Memory blocks updated and read by different tasks.
Cyan	Blocks in triggered subsystems
Brown	Variable sample time. See the Pulse Generator block and "Specifying Sample Time" on page 2-41 for more information
Gray	Fixed in minor step

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

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

The color that is assigned to each block depends on its sample time relative to other sample times in the model. This means that the same sample time may be assigned different colors in a top-level model and in models that it references. (See Chapter 5, "Referencing a Model".)

For example, suppose that a model defines three sample times: 1, 2, and 3. Further, suppose that it references a model that defines two sample times: 2 and 3. In this case, blocks operating at the 2 sample rate appear as green in the 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

Automatically Connecting Blocks

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

Connecting Two Blocks

To autoconnect two blocks:

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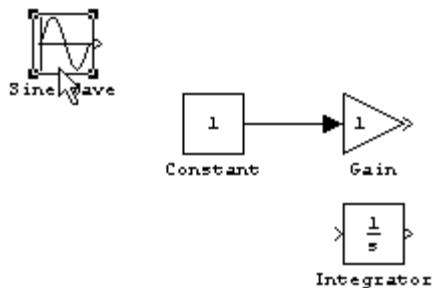
2 Hold down **Ctrl** and left-click the destination block.

The source block is connected to the destination block, and the lines are routed around intervening blocks if necessary.
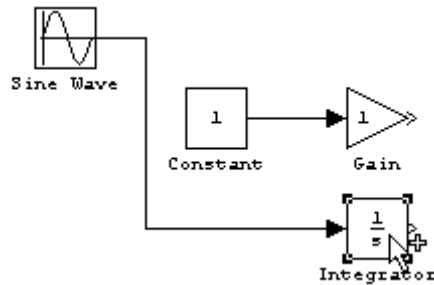
When connecting two blocks, the Simulink software draws as many connections as possible between the two blocks as illustrated in the following example.

Before autoconnect   After autoconnect
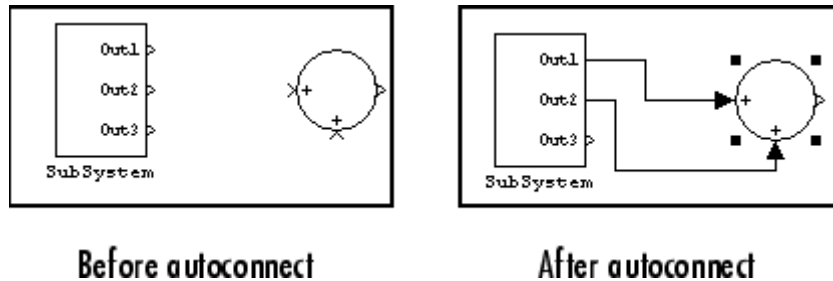
Connecting Groups of Blocks

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

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

1 Select the source blocks.
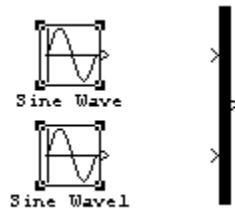
2 Hold down **Ctrl** and left-click the destination block.
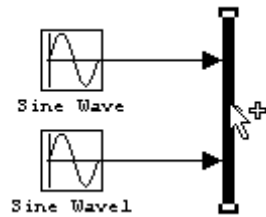
To connect a source block to a group of destination blocks:

1 Select the *destination* blocks.
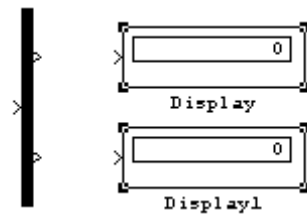
2 Hold down **Ctrl** and left-click the *source* block.
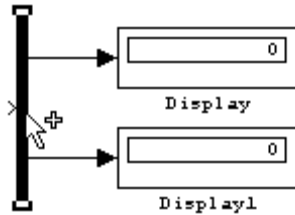
Manually Connecting Blocks

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

Drawing a Line Between Blocks

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

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

The cursor shape changes to crosshairs.

2 Press and hold down the mouse button.

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

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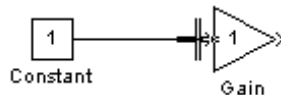
4 Release the mouse button. The port symbols are replaced by a connecting line with an arrow showing the direction of the signal flow. You can create lines either from output to input, or from input to output.

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

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

Drawing a Branch Line

A *branch line* is a line that starts from an existing line and carries its signal to the input port of a block. Both the existing line and the branch line represent the same signal. Using branch lines enables you to connect a signal to more than one block.

This example connect the output of the Product block to both the Scope block and the To Workspace block.
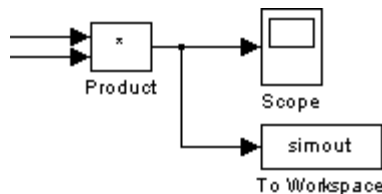
To add a branch line:

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

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

Moving a Line Segment

To move a line segment:

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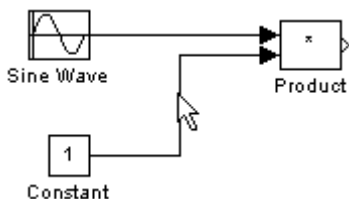
2 Press and hold down the left mouse button.
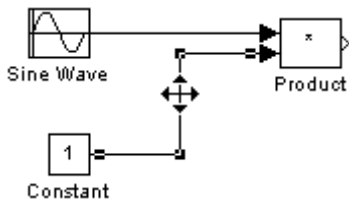
3 Drag the pointer to the desired location.
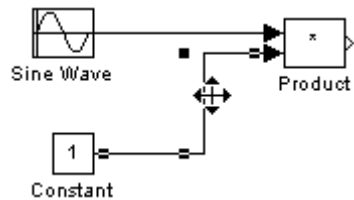
4 Release the mouse button.
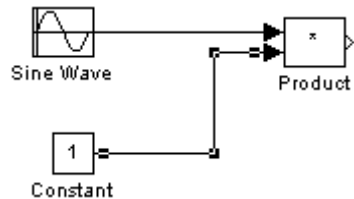
To move the segment connected to an input port, position the pointer over the port and drag the end of the segment to the new location. You cannot move the segment connected to an output port.

Moving a Line Vertex

To move a vertex of a line:

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

The cursor changes to a circle that encloses the vertex.
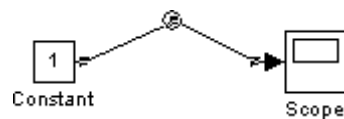
2 Drag the pointer to the desired location.
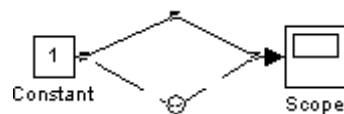
3 Release the mouse button.

Inserting Blocks in a Line

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

To insert a block in a line:

1 Position the pointer over the block and press the left mouse button.
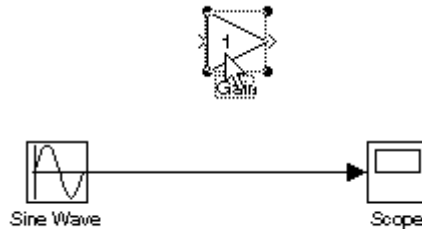
2 Drag the block over the line in which you want to insert the block.
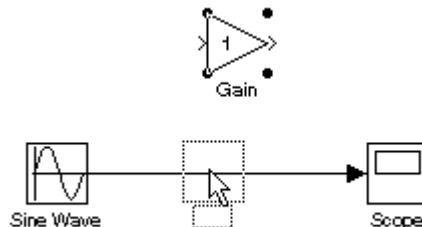
3 Release the mouse button to drop the block on the line.

The block is inserted where you dropped it.
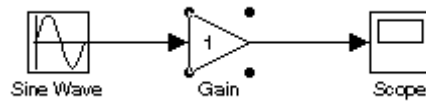
Disconnecting Blocks

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

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

Aligning, Distributing, and Resizing Groups of Blocks

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

1 Select the blocks that you want to align.
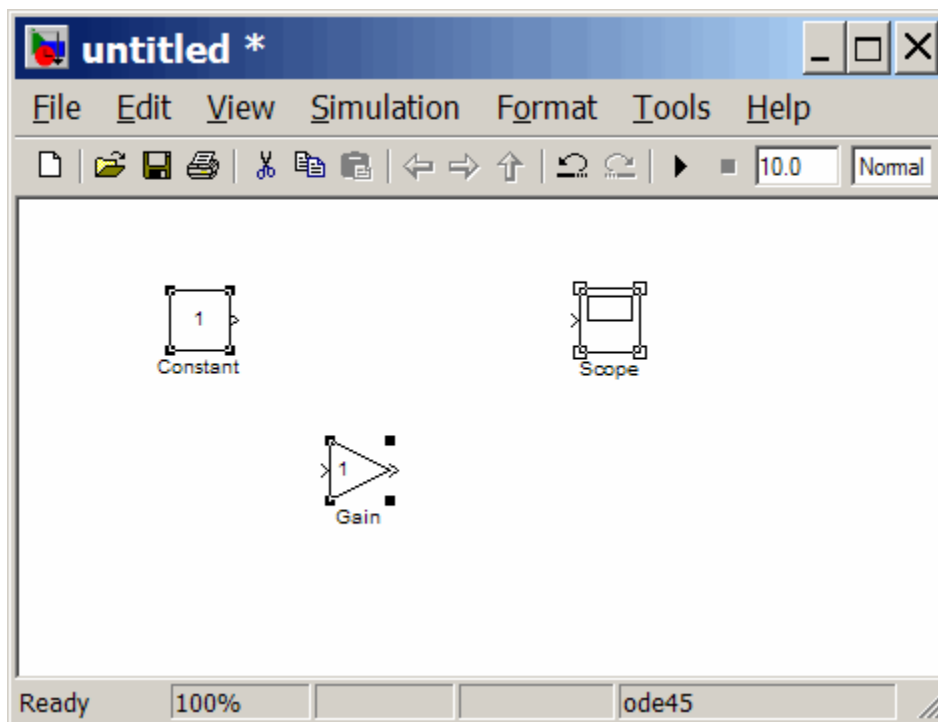
One of the selected blocks displays empty selection handles. The model editor uses this block as the reference for aligning the other selected blocks. If you want another block to serve as the alignment reference, click that block.

2 Select one of the alignment options from the editor's **Format > Align Blocks** menu (or distribution options from the **Format > Distribute Blocks** or resize options from the **Format > Resize Blocks** menu). For

example, selecting **Align Top Edges** aligns the top edges of the selected blocks with the top edge of the reference block.
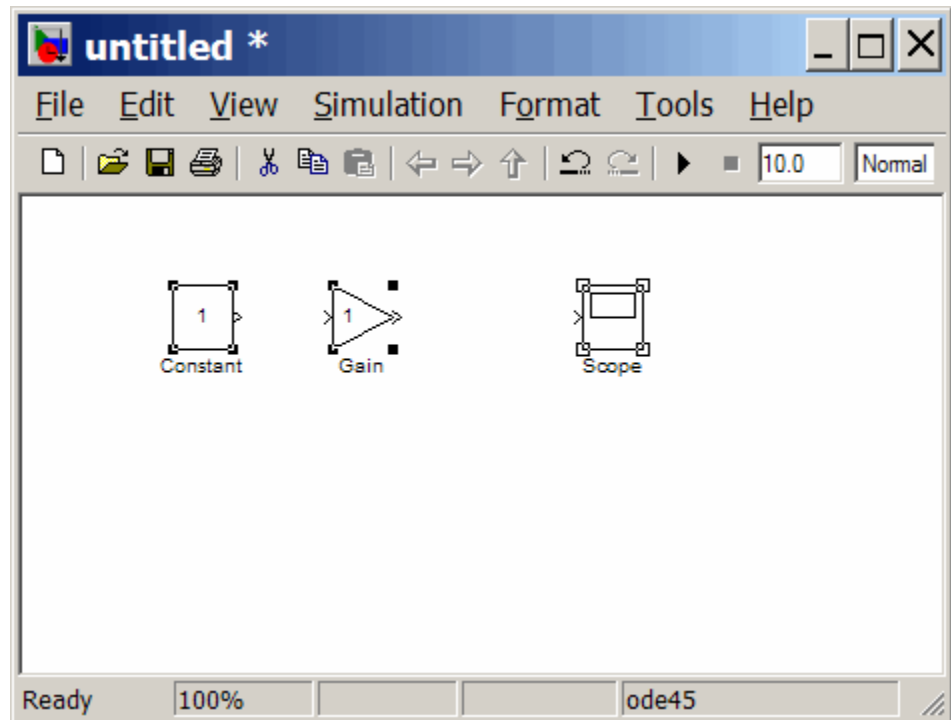
Annotating Diagrams

How to Annotate Diagrams

Annotations provide textual information about a model. You can add an annotation to any unoccupied area of your block diagram.
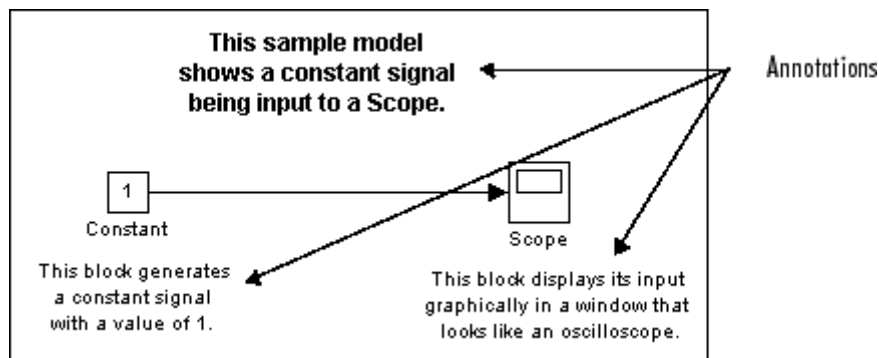
To create a model annotation, double-click an unoccupied area of the block diagram. A small rectangle appears and the cursor changes to an insertion point. Start typing the annotation contents. Each line is centered in the rectangle that surrounds the annotation.

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

To edit an annotation, select it:

- To replace the annotation, click the annotation, then double-click or drag the cursor to select it. Then, enter the new annotation.

- To insert characters, click between two characters to position the insertion point, then insert text.

- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

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

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

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

Annotations Properties Dialog Box

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

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

The dialog box appears.
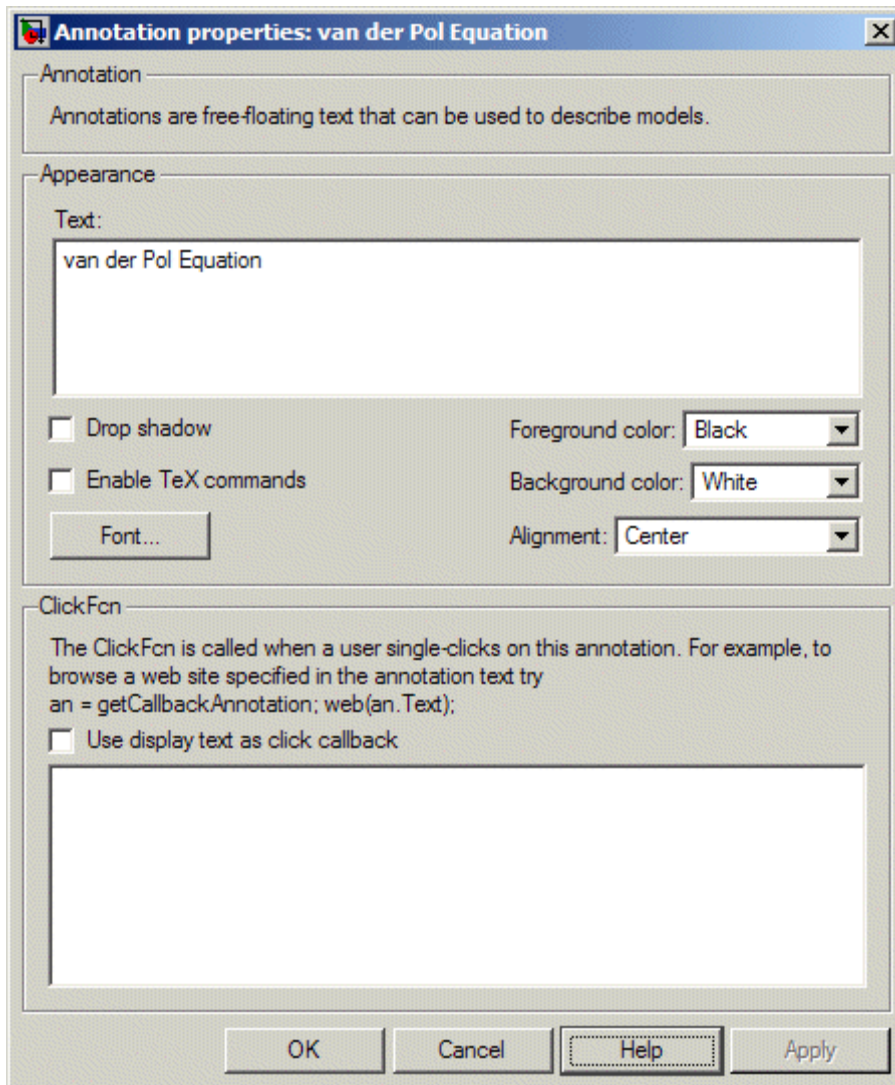
The dialog box includes the following controls.

Text

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

Drop shadow

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

Enable TeX commands

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

Font

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

Foreground color

Specifies the color of the annotation text.

Background color

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

Alignment

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

ClickFcn

Specifies MATLAB® code to be executed when a user single-clicks this annotation. The Simulink software stores the code entered in this field with the model. See "Associating Click Functions with Annotations" on page 3-29 for more information.

Use display text as click callback

Checking this option causes the text in the **Text** field to be treated as the annotation's click function. The specified text must be a valid MATLAB expression comprising symbols that are defined in the MATLAB workspace when the user clicks this annotation. See "Associating Click Functions with Annotations" on page 3-29 for more information. Note that selecting this option disables the **ClickFcn** edit field.

Annotation Callback Functions

You can associate the following callback functions with annotations.

Click Function

A click function is an M function that the Simulink software invokes when a user single-clicks an annotation. You can associate a click function with any of a model's annotations (see "Associating Click Functions with Annotations" on page 3-29). the Simulink software uses the color blue to display the text of annotations associated with click functions. This allows the user to see at a glance which annotations are associated with click functions. Click functions allow you to add hyperlinks and custom command "buttons" to your model's block diagram. For example, you can use click functions to allow a user to display the values of workspace variables referenced by the model or to open related models simply by clicking on annotations displayed on the block diagram. (See Chapter 5, "Referencing a Model".)

Load Function

This function is invoked when it loads the model containing the associated annotation. To associate a load function with an annotation, set the LoadFcn property of the annotation to the desired function (see "Annotations API" on page 3-31).

Delete function

This function is invoked before deleting the associated annotation. To associate a delete function with an annotation, set the DeleteFcn property of the annotation to the desired function (see "Annotations API" on page 3-31).

Associating Click Functions with Annotations

Two ways are provided to associate a click function with an annotation via the annotation's properties dialog box (see "Annotations Properties Dialog Box" on page 3-25). You can specify either the annotation itself as the click function or a separately defined function. To specify the annotation itself as the click function, enter a valid MATLAB expression in the dialog box's **Text** field and check the **Use display text** as callback option. To specify a separately defined click function, enter the M-code that defines the click function in the dialog box's **ClickFcn** edit field.

The following model illustrates the two ways to associate click functions with an annotation.
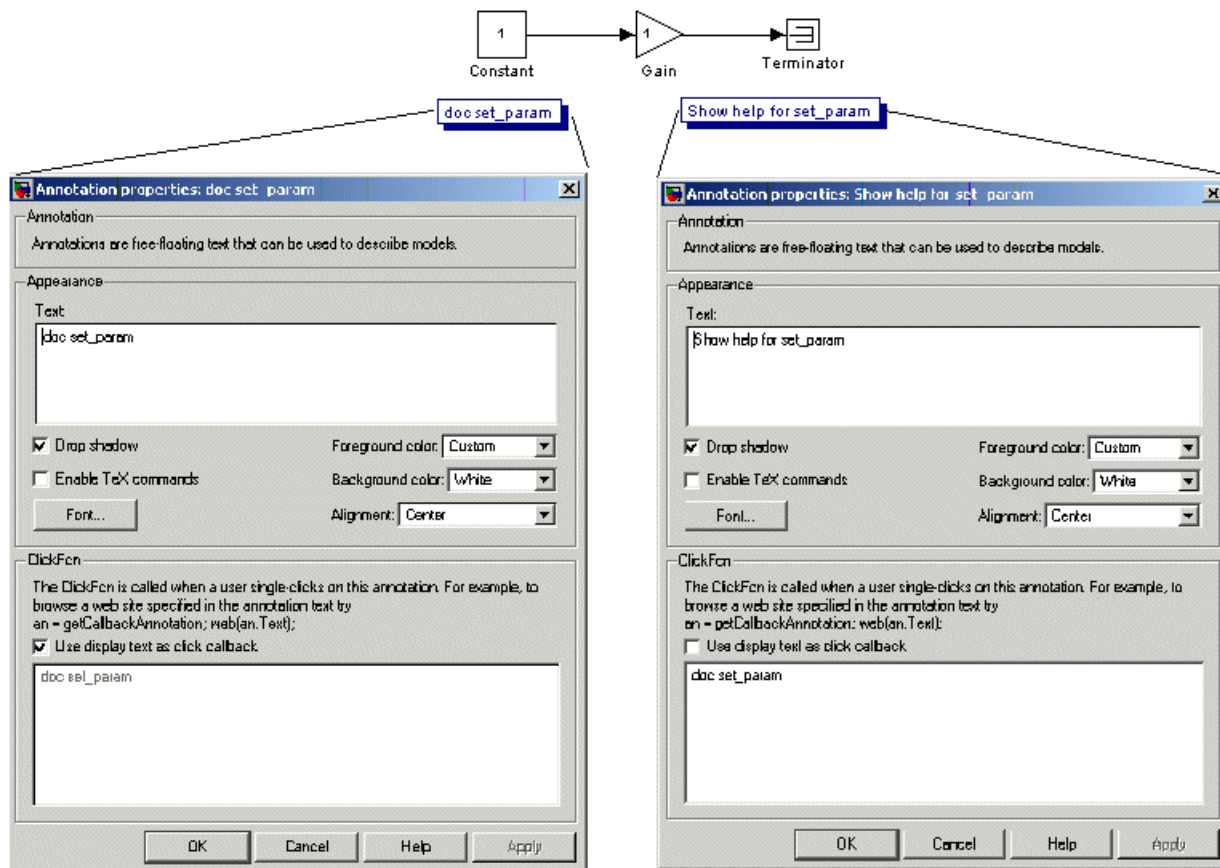
Annotation text as the click function   Separately defined click function

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

Note You can also use M-code to associate a click function with an annotation. See "Annotations API" on page 3-31 for more information.

Selecting and Editing Annotations Associated with Click Functions

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

Annotations API

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

- `Simulink.Annotation` class

  Allows M-code, e.g., annotation load functions (see "Load Function" on page 3-28), 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
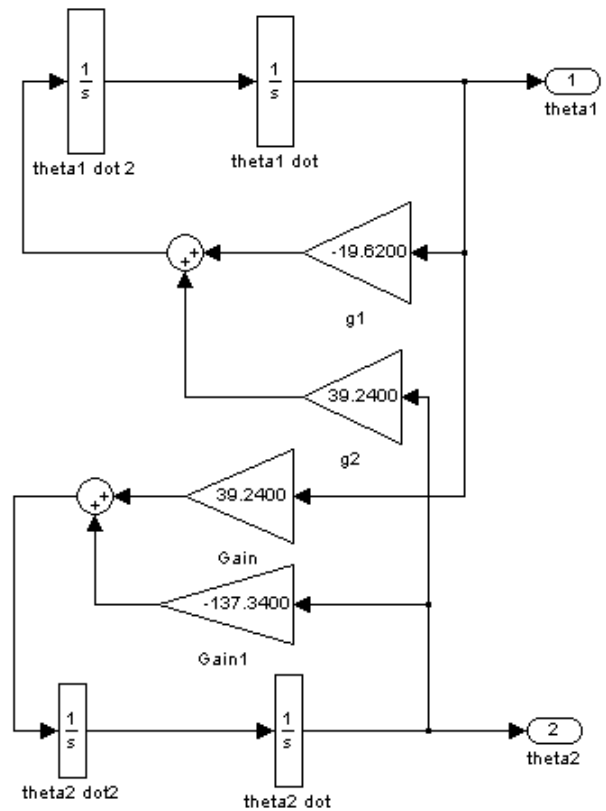
To use TeX commands in an annotation:

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.

See "Mathematical Symbols, Greek Letters, and TeX Characters" in the MATLAB documentation for information on the TeX formatting commands which are supported.

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

The formatted text is displayed.

Creating Annotations Programmatically

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

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

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

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

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

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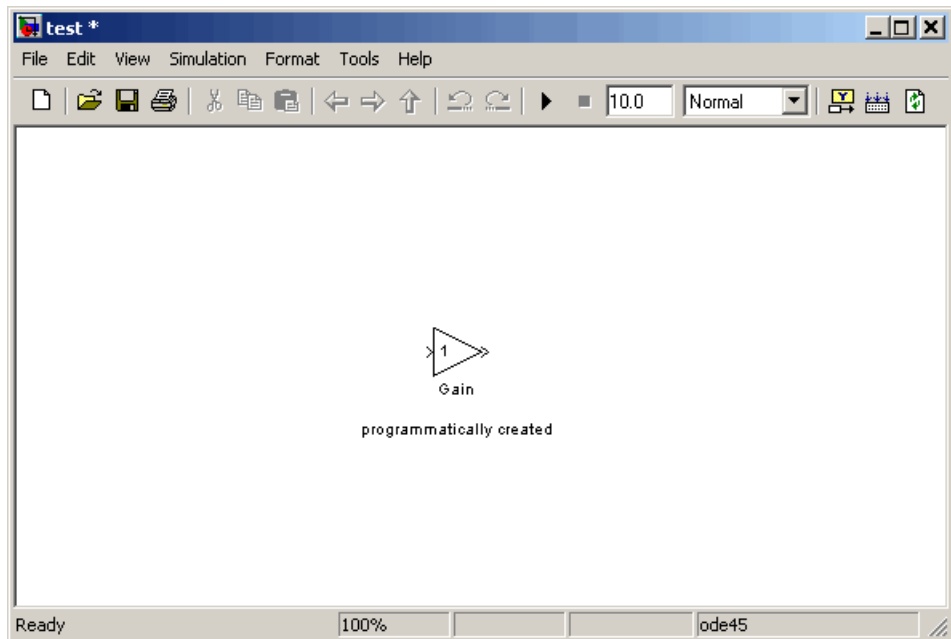
To delete an annotation, use the `find_system` command to get the annotation's handle. Then use the `delete` function to delete the annotation, e.g.,

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

Creating Subsystems

Why Subsystems are Advantageous

A subsystem is a set of blocks that have been replaced by a single block called a Subsystem block. As your model increases in size and complexity, you can simplify it by grouping blocks into subsystems. Using subsystems has these advantages:

- It helps reduce the number of blocks displayed in your model window.

- It allows you to keep functionally related blocks together.

- It enables you to establish a hierarchical block diagram, where a Subsystem block is on one layer and the blocks that make up the subsystem are on another.

You can create a subsystem in two ways:

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

- Add the blocks that make up the subsystem, then group those blocks into a subsystem.

A subsystem can be execute conditionally or unconditionally. An unconditionally executed subsystem always executes. A conditionally executed subsystem may or may not execute, depending on the value of an input signal. For information about conditionally executed subsystems, see Chapter 4, "Creating Conditional Subsystems".

Creating a Subsystem by Adding the Subsystem Block

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

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

2 Open the Subsystem block by double-clicking it.

The subsystem is opened in the current or a new model window, depending on the model window reuse mode that you selected (see "Window Reuse" on page 3-38).

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.
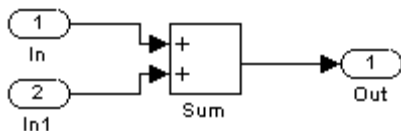
Creating a Subsystem by Grouping Existing Blocks

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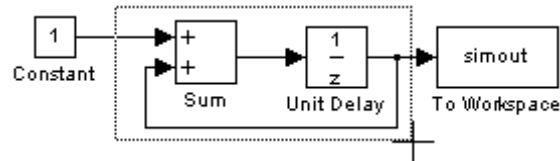
When you release the mouse button, the two blocks and all the connecting lines are selected.

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

This figure shows the model after you choose the **Create Subsystem** command (and resize the Subsystem block so the port labels are readable).
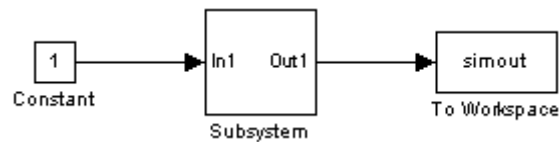
If you open the Subsystem block, the underlying system is displayed, as shown below.
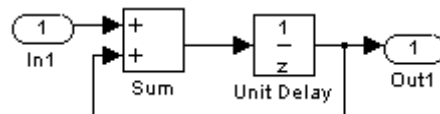
Notice that the Simulink® software adds Inport and Outport blocks to represent input from and output to blocks outside the subsystem.

As with all blocks, you can change the name of the Subsystem block. You can also use the masking feature to customize the block's appearance and dialog box. See Chapter 17, "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. The Simulink software alerts you to this limitation by displaying a warning dialog box before undoing creation of a modified subsystem.

Model Navigation Commands

Subsystems allow you to create a hierarchical model comprising many layers. You can navigate this hierarchy using the Model Browser (see "The Model Browser" on page 13-28) and/or the following model navigation commands:

- **Open Block**

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

- **Open Block In New Window**

  Opens the currently selected subsystem regardless of the window reuse settings (see "Window Reuse" on page 3-38). 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 the Simulink software **View** menu.

Window Reuse

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

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

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

Labeling Subsystem Ports

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

You can hide (or show) the port labels by

- Selecting the Subsystem block, then choosing **Hide Port Labels** (or **Show Port Labels**) from the **Format** menu

- Selecting an Inport or Outport block in the subsystem and choosing **Hide Name** (or **Show Name**) from the **Format** menu

- Selecting the **Show port labels** option in the Subsystem block's parameter dialog

This figure shows two models.

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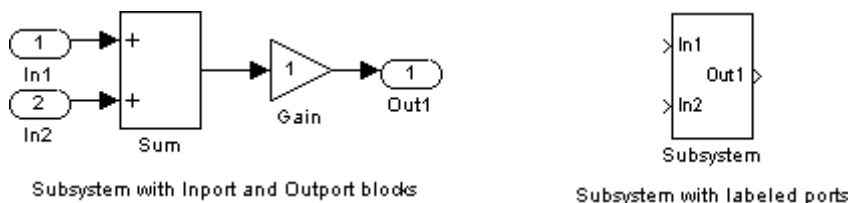
The subsystem on the left contains two Inport blocks and one Outport block.
The Subsystem block on the right shows the labeled ports.

Controlling Access to Subsystems

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

To restrict access to a library subsystem, open the subsystem's parameter
dialog box and set its **Read/Write permissions** parameter to either `ReadOnly`
or `NoReadOrWrite`. The first option allows a user to view the contents of
the library subsystem but prevents the user from modifying the reference
subsystem without first disabling its library link or changing its **Read/Write
permissions** parameter to `ReadWrite`. The second option prevents the user
from viewing the contents of the library subsystem, modifying the reference
subsystem, and changing the reference subsystem's permissions. Note that
both options allow a user to use the library subsystem in models by creating
links (see Chapter 7, "Working with Block Libraries"). See the Subsystem
block in the *Simulink Reference* guide for more information on subsystem
access options.

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

Interconverting Subsystems and Block Diagrams

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

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

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

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

Emptying Subsystems and Block Diagrams

These functions are provided to empty subsystems and block diagrams:

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

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

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

Modeling Control Flow Logic

In this section...
"Equivalent C Language Statements" on page 3-42
"Modeling Conditional Control Flow Logic" on page 3-42
"Modeling While and For Loops" on page 3-45

Equivalent C Language Statements

You can use block diagrams to model control flow logic equivalent to the following C programming language statements:

- for
- if-else
- switch
- while

Modeling Conditional Control Flow Logic

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

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

Modeling If-Else Control Flow

The following diagram models if-else control flow.
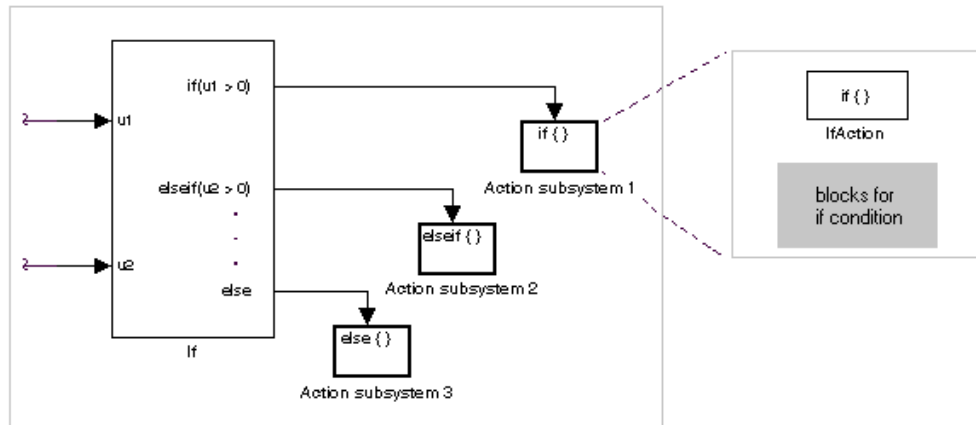
Construct an `if-else` control flow diagram as follows:

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

Inputs to the If block are set in the If block properties dialog box. Internally, they are designated as `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.
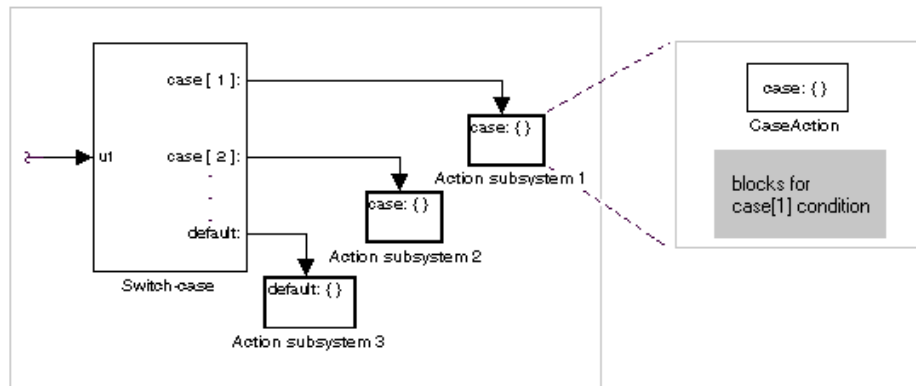
Construct a switch control flow statement as follows:

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

  The input to the Switch Case block is the argument to the switch control flow statement. This value determines the appropriate case to execute. Noninteger inputs to this port are truncated.

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

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

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

  Each case output of the Switch Case block is connected to a subsystem to be executed if the port's case is true. You create these subsystems by placing

an Action Port block in a subsystem. This creates an atomic subsystem with a port named Action, which you then connect to a condition on the Switch Case block. Once connected, the subsystem takes on the identity of the condition and behaves like an enabled subsystem. Place all the block programming executed for that case in this subsystem.

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

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

Modeling While and For Loops

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

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

Modeling While Loops

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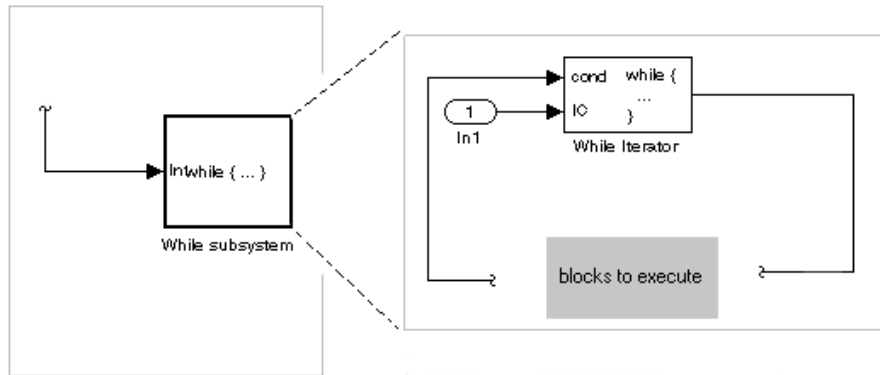
In this example, the Simulink software repeatedly executes the contents of the While subsystem at each time step until a condition specified by the While Iterator block is satisfied. In particular, for each iteration of the loop specified by the While Iterator block, the Simulink software invokes the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

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

Construct a while loop as follows:

• Place a While Iterator block in a subsystem.

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

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

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

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

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

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

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

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

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

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

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

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

For more information, see the While Iterator block.

Modeling For Loops

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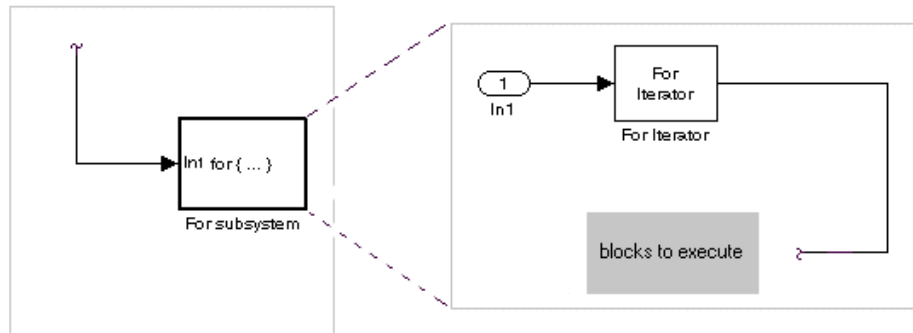
In this example, the Simulink software executes the contents of the For subsystem multiples times at each time step with the number of iterations being specified by the input to the For Iterator block. In particular, for each iteration of the `for` loop, the Simulink software invokes the update and output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

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

Construct a `for` loop as follows:

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

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

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

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

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

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

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

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

The For Iterator block works well with the Assignment block to reassign values in a vector or matrix. This is demonstrated in the following example. Note the matrix dimensions in the data being passed.
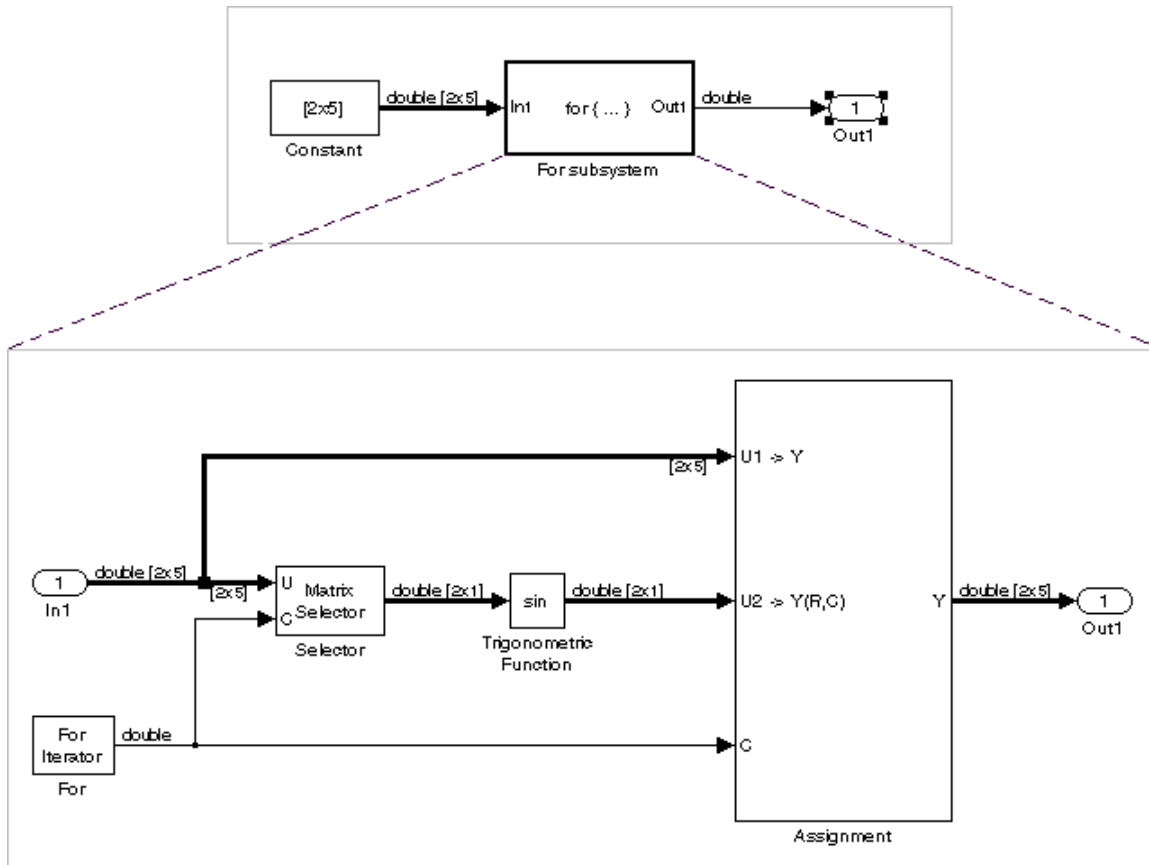
The above example outputs the sine value of an input 2-by-5 matrix (2 rows, 5 columns) using a For subsystem containing an Assignment block. The process is as follows:

1 A 2-by-5 matrix is input to the Selector block and the Assignment block.

2 The Selector block strips off a 2-by-1 matrix from the input matrix at the column value indicated by the current iteration value of the For Iterator block.

3 The sine of the 2-by-1 matrix is taken.

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

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

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

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

Using Callback Functions

In this section...
"About Callback Functions" on page 3-52
"Tracing Callbacks" on page 3-52
"Creating Model Callback Functions" on page 3-53
"Creating Block Callback Functions" on page 3-55
"Port Callback Parameters" on page 3-59

About 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.

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

Tracing Callbacks

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

Creating Model Callback Functions

You can create model callback functions interactively or programmatically. Use the **Callbacks** pane of the model's Model Properties dialog box (see "Callbacks Pane" on page 3-100) 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 3-53).

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.

Parameter	When Executed
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. **Note** In a PreLoadFcn callback routine, the get_param command does not return the model's parameter values because the model is not yet loaded. In a PreLoadFcn routine, get_param returns: • The default value for a standard model parameter such as solver • An error message for a model parameter added with add_param In a PostLoadFcn callback routine, however, get_param returns the model's parameter values because the model is loaded.
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. (See Chapter 5, "Referencing a Model".) For example, suppose that model A references model B and that model A's `OpenFcn` creates variables in the MATLAB workspace and model B's `CloseFcn` clears the MATLAB workspace. Now suppose that simulating model A requires rebuilding model B. Rebuilding B entails opening and closing model B and hence invoking model B's `CloseFcn`, which clears the MATLAB workspace, including the variables created by A's `OpenFcn`.

Creating Block Callback Functions

You can create block callback functions interactively or programmatically. Use the **Callbacks** pane of the block's Block Properties dialog box (see "Callbacks Pane" on page 6-25) 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 3-55).

Note A callback for a masked subsystem cannot directly reference the parameters of the masked subsystem (see "About Masks" on page 17-2). The Simulink software evaluates block callbacks in the MATLAB base workspace whereas the mask parameters reside in the masked subsystem's private workspace. A block callback, however, can use get_param to obtain the value of a mask parameter, e.g., get_param(gcb, 'gain'), where gain is the name of a mask parameter of the current block.

Block Callback Parameters

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

Parameter	When Executed
ClipboardFcn	When the block is copied or cut to the system clipboard.

Parameter	When Executed
CloseFcn	When the block is closed using the close_system command. The CloseFcn is not called when you interactively close the block, when you interactively close the subsystem or model containing the block, or when you close the subsystem or model containing the block using close_system.
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 or line is deleted in a subsystem. If the block has a DeleteFcn or DestroyFcn, those functions are executed prior to the DeleteChildFcn. Only Subsystem blocks have a DeleteChildFcn callback.
DeleteFcn	After a block is graphically deleted, e.g., when you graphically delete the block, invoke delete_block on the block, or close the model containing the block. When the DeleteFcn is called, the block handle is still valid and can be accessed using get_param. The DeleteFcn callback is recursive for Subsystem blocks. If the block is graphically deleted by invoking delete_block or by closing the model, after deletion the block is destroyed from memory and the block's DestroyFcn is called.
DestroyFcn	When the block has been destroyed from memory, e.g., when you invoke delete_block on either the block or a subsystem containing the block or close the model containing the block. If the block was not previously graphically deleted, the block's DeleteFcn is called prior to the DestroyFcn. When the DestroyFcn is called, the block handle is no longer valid.

Parameter	When Executed
InitFcn	Before the block diagram is compiled and before block parameters are evaluated.

ErrorFcn	When an error has occurred in a subsystem. Only Subsystem blocks have a ErrorFcn callback. 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 Simulink string indicating the type of error that occurred, and errorMsg is a string specifying the error message to be displayed to the user. The following command sets the ErrorFcn of the subsystem subsys to call the errorHandler callback function
	
	```set_param(subsys,'ErrorFcn','errorHandler')```
	
	Do not include the callback function's input arguments in the call to set_param. The Simulink software displays the error message errorMsg returned by the callback function.

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.

Parameter	When Executed
OpenFcn	When the block is opened. This parameter is generally used with Subsystem blocks. The routine is executed when you double-click the block or when an open_system command is called with the block as an argument. The OpenFcn parameter overrides the normal behavior associated with opening a block, which is to display the block's dialog box or to open the subsystem.
ParentCloseFcn	Before closing a subsystem containing the block or when the block is made part of a new subsystem using the new_system command (see new_system in the online Simulink software 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.

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

Port Callback Parameters

Block input and output ports have a single callback function parameter, ConnectionCallback. This parameter allows you to set callbacks on ports that are triggered every time the connectivity of these ports changes. Examples of connectivity changes include adding a connection from the port to a block, deleting a block connected to the port, and deleting, disconnecting, or connecting branches or lines to the port.

Use get_param to get the port handle of a port and set_param to set the callback on the port. The callback function must have one input argument that represents the port handle, but the input argument is not included in the call to set_param. For example, suppose the currently selected block has a single input port. The following code fragment sets foo as the connection callback on the input port.

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

where, foo is defined as:

```
function foo(portHandle)
```

Using Model Workspaces

In this section...
"About Model Workspaces" on page 3-61
"Changing Model Workspace Data" on page 3-62
"Model Workspace Dialog Box" on page 3-64

About Model Workspaces

Each model is provided with its own workspace for storing variable values. The model workspace is similar to the base MATLAB® workspace except that

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

  If both the MATLAB workspace and a model workspace define a variable of the same name, and the variable does not appear in any intervening masked subsystem or model workspaces, the Simulink® software uses the value of the variable in the model workspace. A model's workspace effectively provides it with its own name space, allowing you to create variables for the model without risk of conflict with other models.

- When the model is loaded, the workspace is initialized from a data source.

  The data source can be the model's MDL-file, a MAT-file, or M-code stored in the model file (see "Data source" on page 3-65 for more information).

- You can interactively reload and save MAT-file and M-code data sources.

- Only `Simulink.Parameter` and `Simulink.Signal` objects for which the storage class is set to `Auto` can reside in a model workspace. You must create all other Simulink software data objects in the base MATLAB workspace to ensure the objects are unique within the global Simulink context and accessible to all models.

Note Subclasses of `Simulink.Parameter` and `Simulink.Signal` classes, including `mpt.Parameter` and `mpt.Signal` objects (Real-Time Workshop® Embedded Coder license required), can reside in a model workspace only if their storage class is set to `Auto`.

- In general, parameter variables in a model workspace are not tunable.

  However, you can tune model workspace variables declared as model arguments for referenced models (see "Using Model Arguments" on page 5-28 for more information).

Note When resolving references to variables used in a referenced model, the referenced model's variables are resolved as if the parent model did not exist. For example, suppose a referenced model references a variable that is defined in both the parent model's workspace and in the MATLAB workspace but not in the referenced model's workspace. In this case, the MATLAB workspace is used. (See Chapter 5, "Referencing a Model".)

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

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

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

Changing Model Workspace Data

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

Changing Workspace Data Whose Source Is the Model File

If a model workspace's data source is data stored in the model, you can use Model Explorer (see "The Model Explorer" on page 13-2) or MATLAB commands to change the model's workspace (see "Using MATLAB® Commands to Change Workspace Data" on page 3-63).

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 3-64). To discard changes to the workspace, use the **Reinitialize From Source** button on the Model Workspace dialog box.

Changing Workspace Data Whose Source Is M-Code

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

Using MATLAB® Commands to Change Workspace Data

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

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

This command returns a handle to a `Simulink.ModelWorkspace` object whose properties specify the source of the data used to initialize the model workspace. Edit the properties to change the data source. Use the workspace's

methods to list, set, and clear variables, evaluate expressions in, and save and reload the workspace.

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

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

Model Workspace Dialog Box

The Model Workspace dialog box enables you to specify a model workspace's source and model reference arguments (See Chapter 5, "Referencing a Model".) To display the dialog box, select the model workspace in Model Explorer's Model Hierarchy pane. To use MATLAB commands to change data in a model workspace, see "Using MATLAB® Commands to Change Workspace Data" on page 3-63.
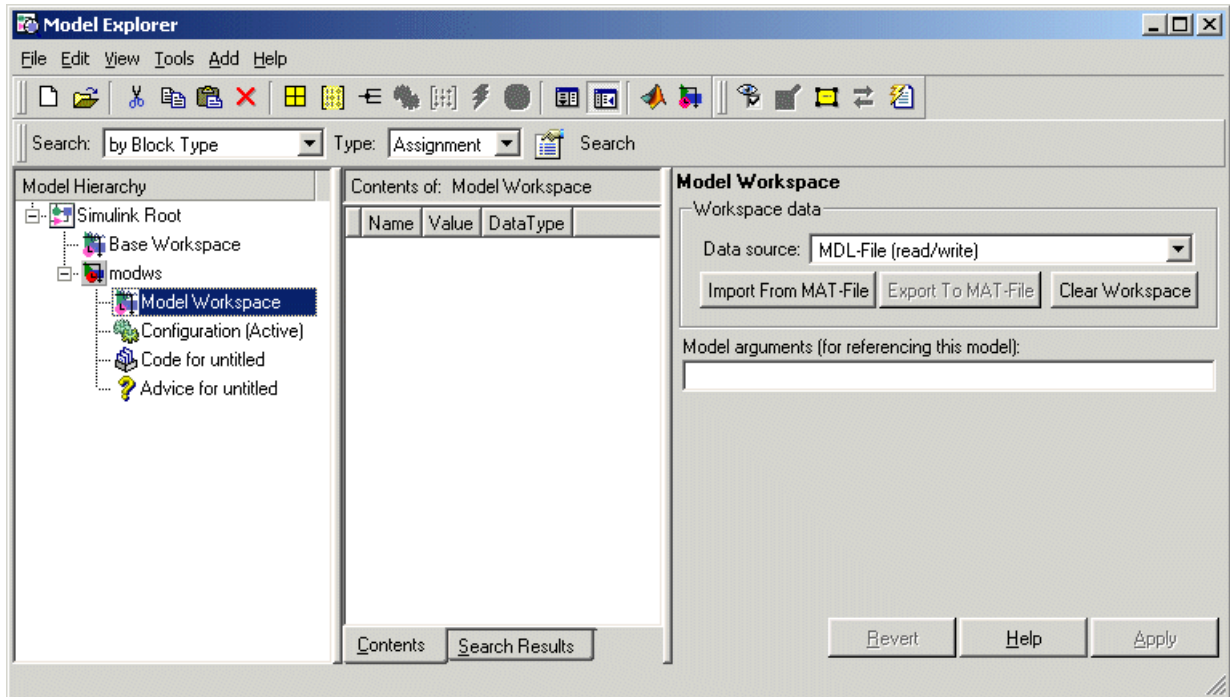
The dialog box contains the following controls.

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 3-66).

- `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 3-66).

- `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 3-67).

MDL-File Source Controls

Selecting Mdl-File as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.
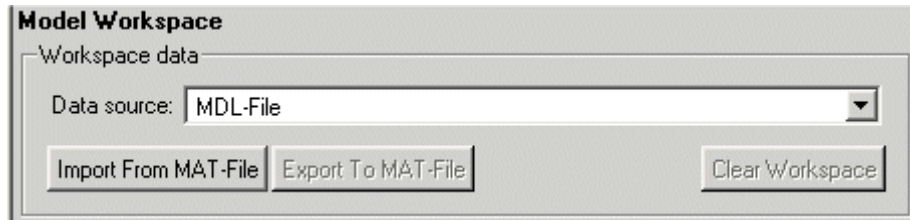
Import From MAT-File. This button lets you import data from a MAT-file. Selecting the button causes a file selection dialog box to be displayed. Use the dialog box to select the MAT file that contains the data you want to import.

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

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

MAT-File Source Controls

Selecting MAT-File as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.
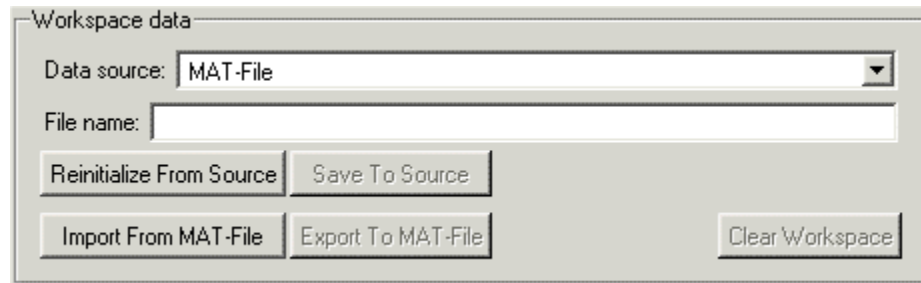
File name. File name or path name of the MAT file that is the data source for the selected workspace. If a file name, the name must reside on the MATLAB path.

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

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

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

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

Clear Workspace. Clears the selected workspace.

M-Code Source Controls
Selecting M-Code as the **Data source** for a workspace causes the Model Workspace dialog box to display additional controls.
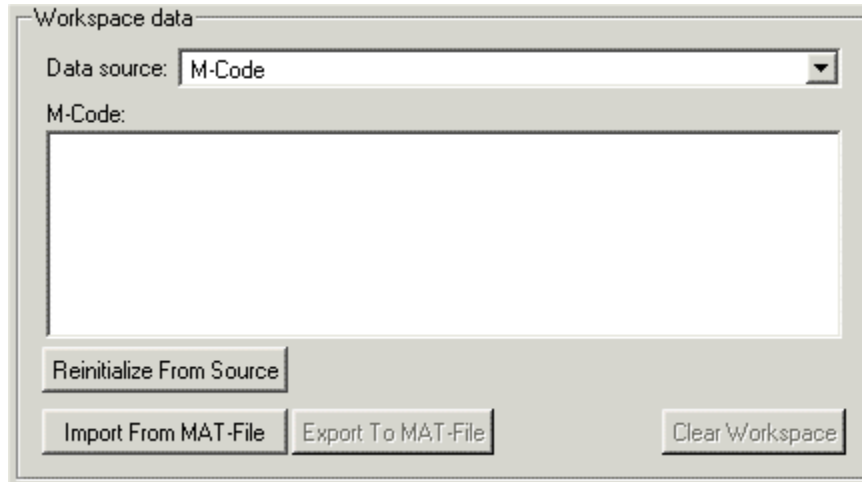
M-Code. Specifies M-code that initializes the selected workspace. To change the initialization code, edit this field, then select the **Reinitialize from source** button on the dialog box to clear the workspace and execute the modified code.

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

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

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

Clear Workspace. Clears the selected workspace.

Model Arguments
This field allows you to specify arguments that can be passed to instances of this model referenced by another model. See Chapter 5, "Referencing a Model" and "Using Model Arguments" on page 5-28 for more information.

Resolving Symbols

About Symbol Resolution

When you create a Simulink® model, you can use symbols to provide values and definitions for many types of entities in the model. Model entities that can be defined with symbols include block parameters, configuration set parameters, data types, signals, signal properties, and bus architecture.

A symbol that provides a value or definition must be a legal MATLAB® identifier. Such an identifier starts with an alphabetic character, followed by up to 63 alphanumeric or underscore characters. A symbol provides a value or definition in a Simulink model by corresponding to some item that:

- Exists in an accessible workspace
- Has a name that matches the symbol
- Provides the required information

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

When the Simulink software compiles a model, it tries to resolve every symbol in the model, except symbols in M-code that runs in a callback or as part of

mask initialization. Depending on the particular case, the item to which a symbol resolves can be a variable, object, or function.

Hierarchical Symbol Resolution

The Simulink software attempts to resolve a symbol by searching through the accessible workspaces in hierarchical order for a MATLAB variable or Simulink object whose name is the same as the symbol. The search path is identical for every symbol. The search begins with the block that uses the symbol, or creates a signal that is named by the symbol, and proceeds upward. Except when simulation occurs via the `sim` command, the search order is the following:

1 Any mask workspaces, in order from the block upwards (see "Mask Workspace" on page 17-4)

2 The model workspace of the model that contains the block (see "Using Model Workspaces" on page 3-61)

3 The MATLAB base workspace (See "MATLAB Workspace")

If the Simulink software finds a matching item in the course of this search, the search terminates successfully at that point, and the symbol resolves to the matching item. The result is the same as if the value of that item had appeared literally instead of the symbol that resolved to the item. An object defined at a lower level shadows any object defined at a higher level.

If no matching item exists on the search path, the Simulink software attempts to evaluate the symbol as a function. If the function is defined and returns an appropriate value, the symbol resolves to whatever the function returned. Otherwise, the symbol remains unresolved, and an error occurs. Evaluation as a function occurs as the final step whenever a hierarchical search terminates without having found a matching workspace item.

If the model that contains the symbol is a referenced model, and the search reaches the model workspace but does not succeed there, the search jumps directly to the base workspace *without* trying to resolve the symbol in the workspace of any parent model. Thus a given symbol will resolve to the same item irrespective of whether the model that contains the symbol is a

referenced model. See Chapter 5, "Referencing a Model" for information about model referencing.

Specifying Numeric Values with Symbols

Any block parameter that requires a numeric value can be specified by providing a literal value, a symbol, or an expression, which can contain symbols and literal values. Each is symbol evaluated separately, as if none of the others existed. Different symbols in an expression can thus resolve to items on different workspaces, and to different types of item.

When a single symbol appears and resolves successfully, its value provides the value of the parameter. When an expression appears, and all symbols resolve successfully, the value of the expression provides the value of the parameter. If any symbol cannot be resolved, or resolves to a value of inappropriate type, an error occurs.

For example, suppose that the **Gain** parameter of a Gain block is given as `cos(j*(k+2))`. The symbol `cos` will resolve to the MATLAB cosine function, and `j` and `k` must resolve to numeric values, which could be obtained from the same or different types of items in the same or different workspaces. If the symbols resolve to numeric values, the value returned by the cosine function becomes the value of the **Gain** parameter.

Specifying Other Values with Symbols

Most symbols and expressions that use them provide numeric values, but the same techniques that provide numeric values can provide any type of value that is appropriate for its context. Another common use of symbols is to name objects that provide definitions of some kind. For example, a signal name can resolve to a signal object (`Simulink.Signal`) that defines the properties of the signal, and a Bus Creator block's **Bus object** parameter can name a bus object (`Simulink.Bus`) that defines the properties of the bus. Symbols can be used when defining data types, can specify input data sources and logged data destinations, and can serve many other purposes.

From the standpoint of hierarchical symbol resolution, all of these different uses of symbols, whether singly or in expressions, are the same: each symbol is resolved, if possible, independently of any others, and the result becomes available where the symbol appeared. The only difference between one symbol

and another is the specific item to which the symbol resolves and the use made of that item. The only requirement is that every symbol must resolve to something that can legally appear at the location of the symbol.

Limiting Signal Resolution

Hierarchical symbol resolution traverses the complete search path by default. You can truncate the search path by using the **Permit Hierarchical Resolution** option of any subsystem. This option controls what happens if the search reaches that subsystem without resolving to a workspace variable. The **Permit Hierarchical Resolution** values are:

- `All`

  Continue searching up the workspace hierarchy trying to resolve the symbol. This is the default value.

- `None`

  Do not continue searching up the hierarchy. Attempt to resolve the symbol as a function.

- `ExplicitOnly`

  Continue searching up the hierarchy only if the symbol specifies a block parameter value, data store memory (where no block exists), or a signal or state that explicitly requires resolution. Do not continue searching for an implicit resolution. See "Explicit and Implicit Resolution" on page 3-72 for more information.

If the search does not find a match in the workspace, and terminates because the value is `ExplicitOnly` or `None`, the Simulink software evaluates the symbol as a function. The search succeeds or fails depending on the result of the evaluation, as previously described.

Explicit and Implicit Resolution

Models and some types of model entities have associated parameters that can affect symbol resolution. For example, suppose that a model includes a signal named Amplitude, and that a `Simulink.Signal` object named Amplitude exists in an accessible workspace. If the Amplitude signal's **Signal name**

must resolve to Simulink signal object option is checked, the signal will resolve to the object. See "Signal Properties Dialog Box" for more information.

If the option is not checked, the signal may or may not resolve to the object, depending on the value of **Configuration Parameters > Data Validity > Signal resolution**. This parameter can suppress resolution to the object even though it is available, or it can specify that resolution occurs on the basis of the name match alone. See "Diagnostics Pane: Data Validity" > "Signal resolution" for more information.

Resolution that occurs because an option like **Signal name must resolve to Simulink signal object** requires it is called *explicit symbol resolution*. Resolution that occurs on the basis of name match alone, without an explicit specification, is called *implicit symbol resolution*.

Programmatic Symbol Resolution

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

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

Working with Data Stores

In this section...
"About Data Stores" on page 3-74
"Defining Data Stores" on page 3-74
"Accessing Data Stores" on page 3-76
"Data Store Examples" on page 3-77

About 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 referenced models to share data without having to use I/O ports to pass the data from level to level.

See Chapter 5, "Referencing a Model" for information about referenced models, and "Data Store Examples" on page 3-77 for examples of using data stores to share data among subsystems and model references.

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 3-62). 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.,
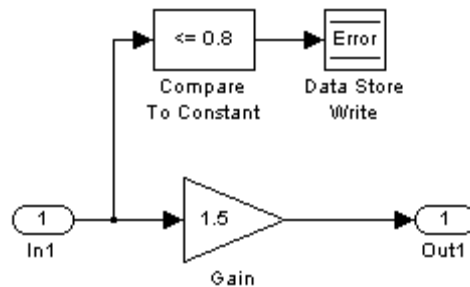
To get the value of a data store at each time step, create an instance of a
Data Store Read block at the level of your model that needs the value, set the
block's **Data store name** parameter to the name of the data store to be read,
and connect the output of the data store read block to the input of the block
that need's the data store's value, e.g.,
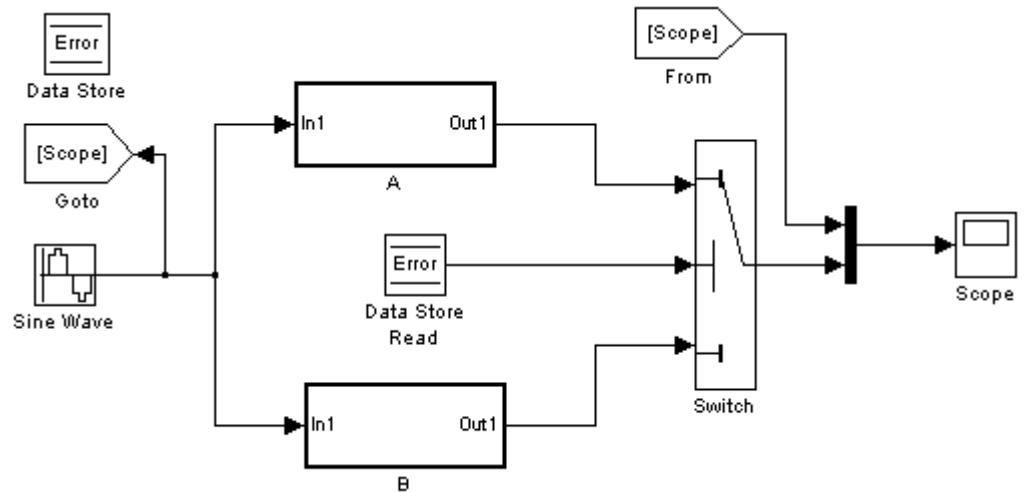
When connected to a global data store, a data store access block displays the word Global above the data store's name.

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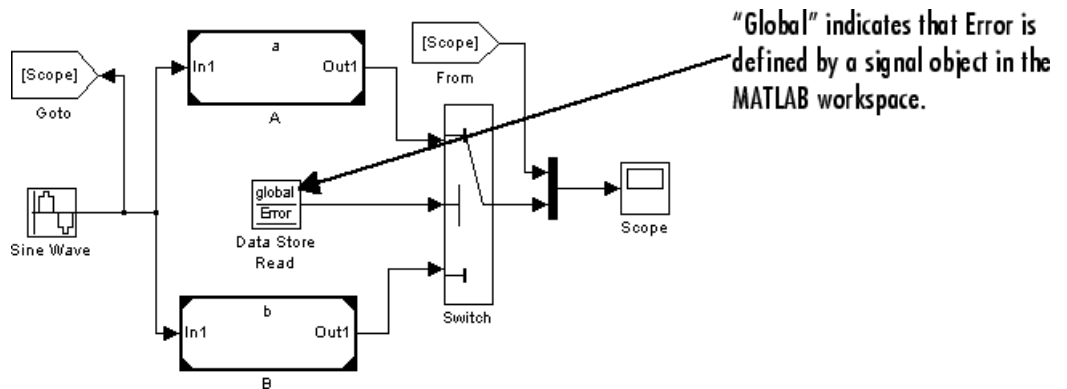
This is done to remind you that the data store is defined by a signal object in the MATLAB workspace rather than by a Data Store Memory block.

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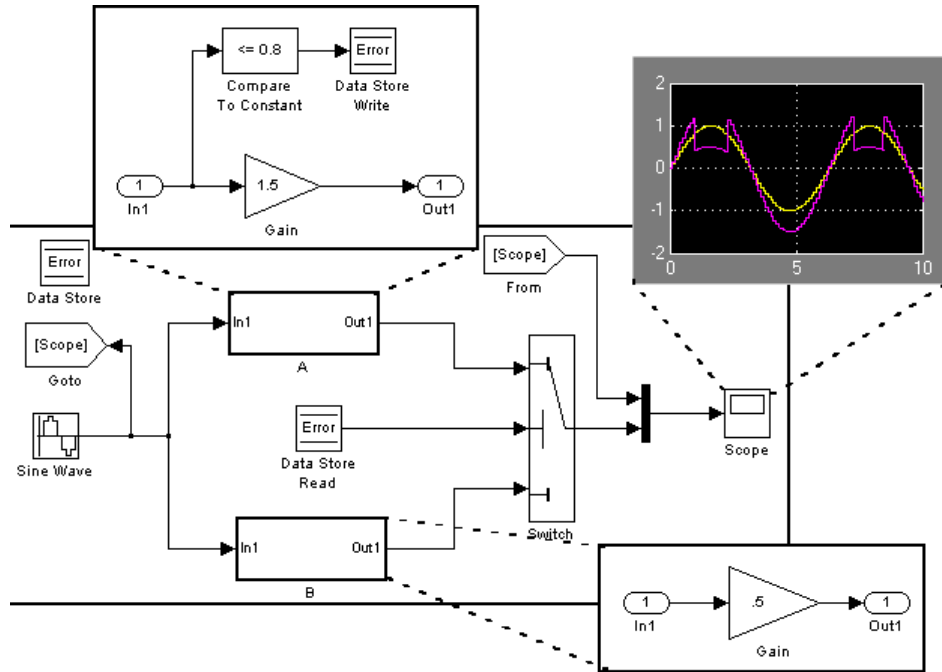
This model uses a data store to permit subsystem A to signal that its output is invalid. If subsystem A's output is invalid, the model uses the output of subsystem B.

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.

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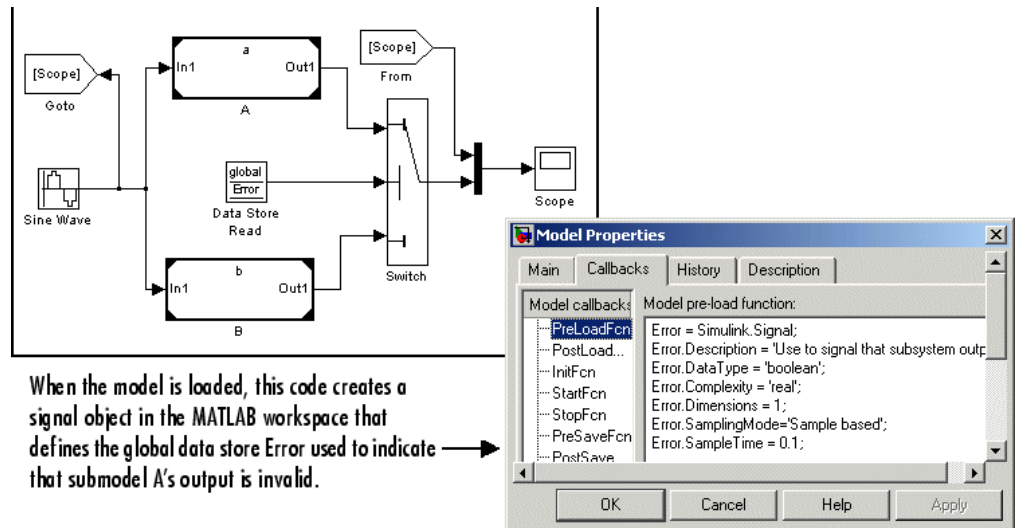
In this example, the top model uses a signal object in the MATLAB workspace to define the error data store. This is necessary because data stores are visible across model boundaries only if they are defined by signal objects in the MATLAB workspace.

Consulting the Model Advisor

About the Model Advisor

The Model Advisor checks a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation of the system represented by the model or in 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 "Model Advisor Checks" in the Simulink® Reference documentation for more information on individual checks.

Starting Model Advisor

You can use any of the following methods to start the 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 13-2), select **Advice for *model1***, where *model1* is the name of the model that you want to check.

- At the MATLAB® prompt, enter modeladvisor(*model*), where *model* is a handle or name of the model or subsystem you want to check (see the modeladvisor function reference page for more information).

- Select **Model Advisor** from the context (right-click) menu of a subsystem that you want to check.

Note   The Model Advisor uses the Simulink project (`slprj`) directory (see "Simulation Targets" on page 5-16 for more information) in the current directory to store reports and other information. If such a directory does not exist in the current directory, the Model Advisor creates it. For this reason, you should, before starting the Model Advisor, ensure that the current directory is writable. If the directory is not writable, an error message is displayed when you start the Model Advisor.

Model Advisor Window

When you start the Model Advisor, the Model Advisor window is displayed. Expanding the folders displays the available checks.
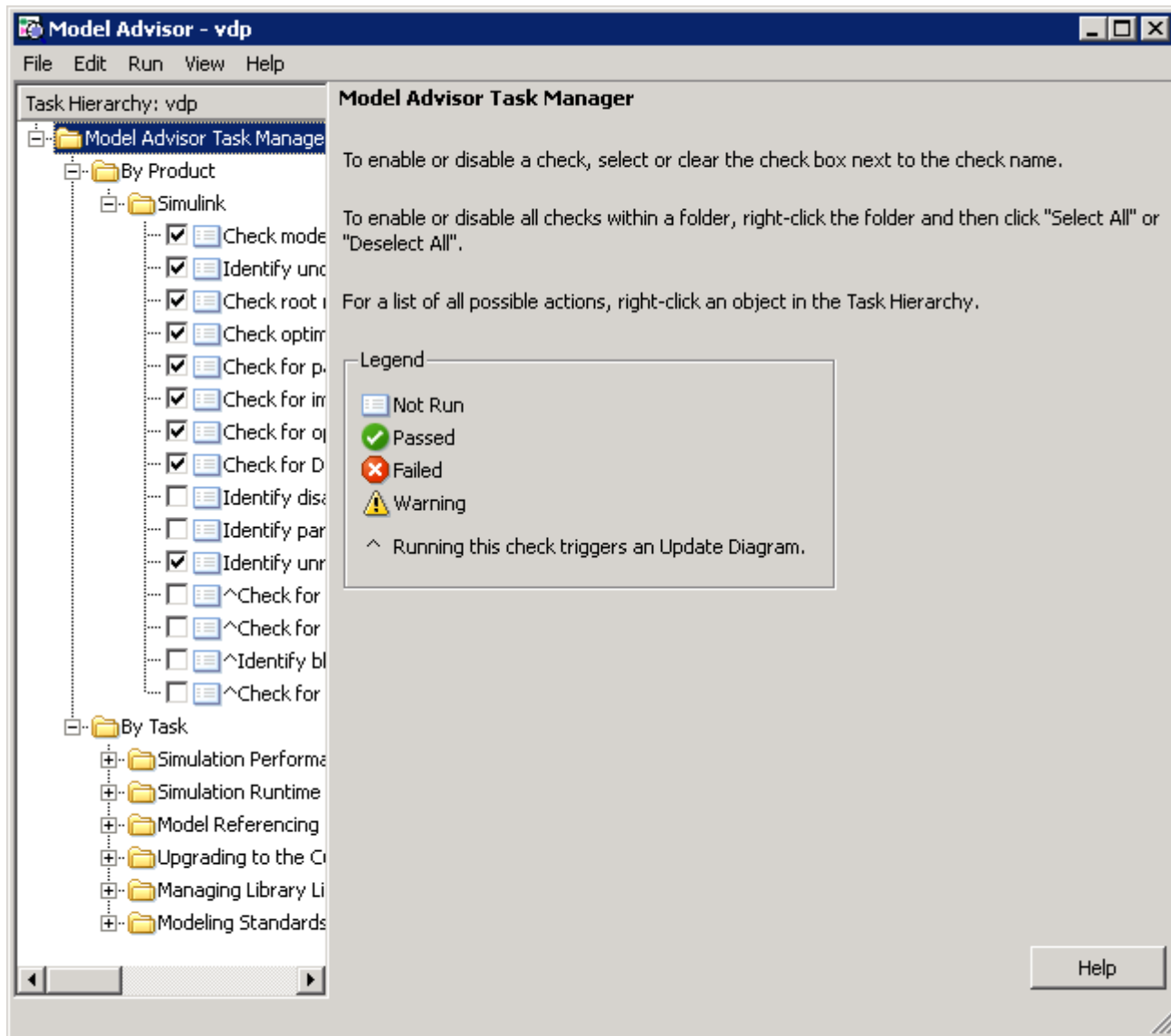
The left pane lists the checks that the Model Advisor performs. By default, the Model Advisor groups the checks by product. Select **By Task** to display

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 run an individual check or all selected checks. Some checks have input parameters where you can specify information provided for the check to run (see "Check for proper Merge block usage" for an example). The results of the checks are displayed in the Model Advisor window. Additionally, the Model Advisor generates an HTML report of the check results, which you can opt to view in a separate browser window.

Note When you open the Model Advisor on a model that you have previously checked, the Model Advisor tool 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. You can also reset the status of the checks to not run by right-clicking the **Model Advisor Task Manager** folder and selecting `Reset` from the folder context menu.

Navigating Model Advisor Checks

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

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

2 Open the Model Advisor by selecting **Model Advisor** from the Model Editor's **Tools** menu. The Model Advisor window starts and displays checks for the `vdp` demo model, as shown in "Model Advisor Window" on page 3-81. By default, most of the checks are selected.

3 Select **By Product** in the left pane. This changes the right pane to a **By Product** view.
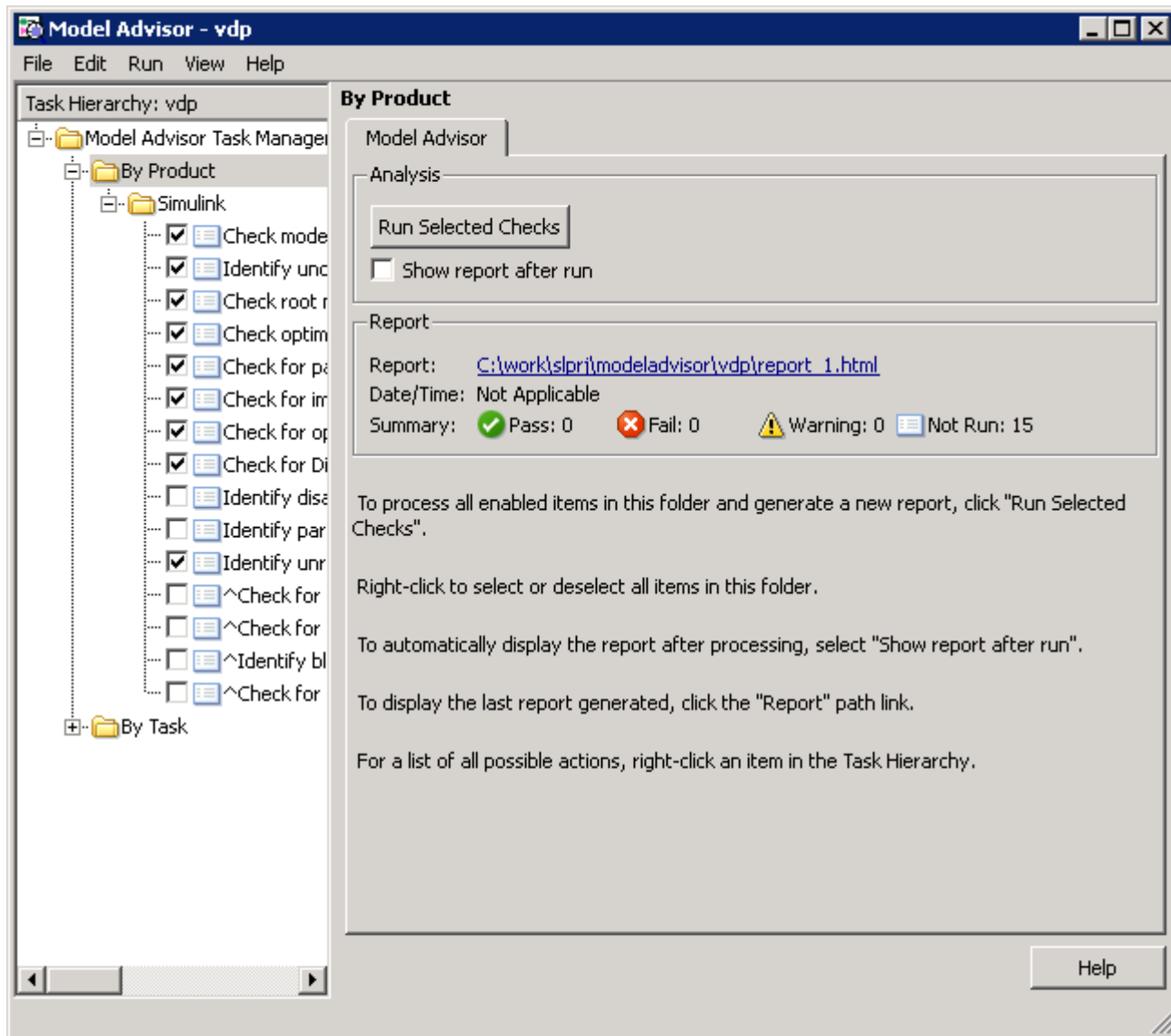
4 Select the **Show report after run** check box. This option causes an HTML report of check results to be displayed after the checks run.

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

Tip While you can use Model Advisor reports to fix checks, you should use the Model Advisor window for interactive check fixing. Use Model Advisor reports to view a summary of checks rather than to fix checks.

---

6 Return to the Model Advisor window, which has been updated with the check results.

7 Select an individual check to open a detailed view of the results in the right pane. For example, selecting **Check optimization settings** changes the right pane to the following view. Unose this view to examine and exercise a check individually.
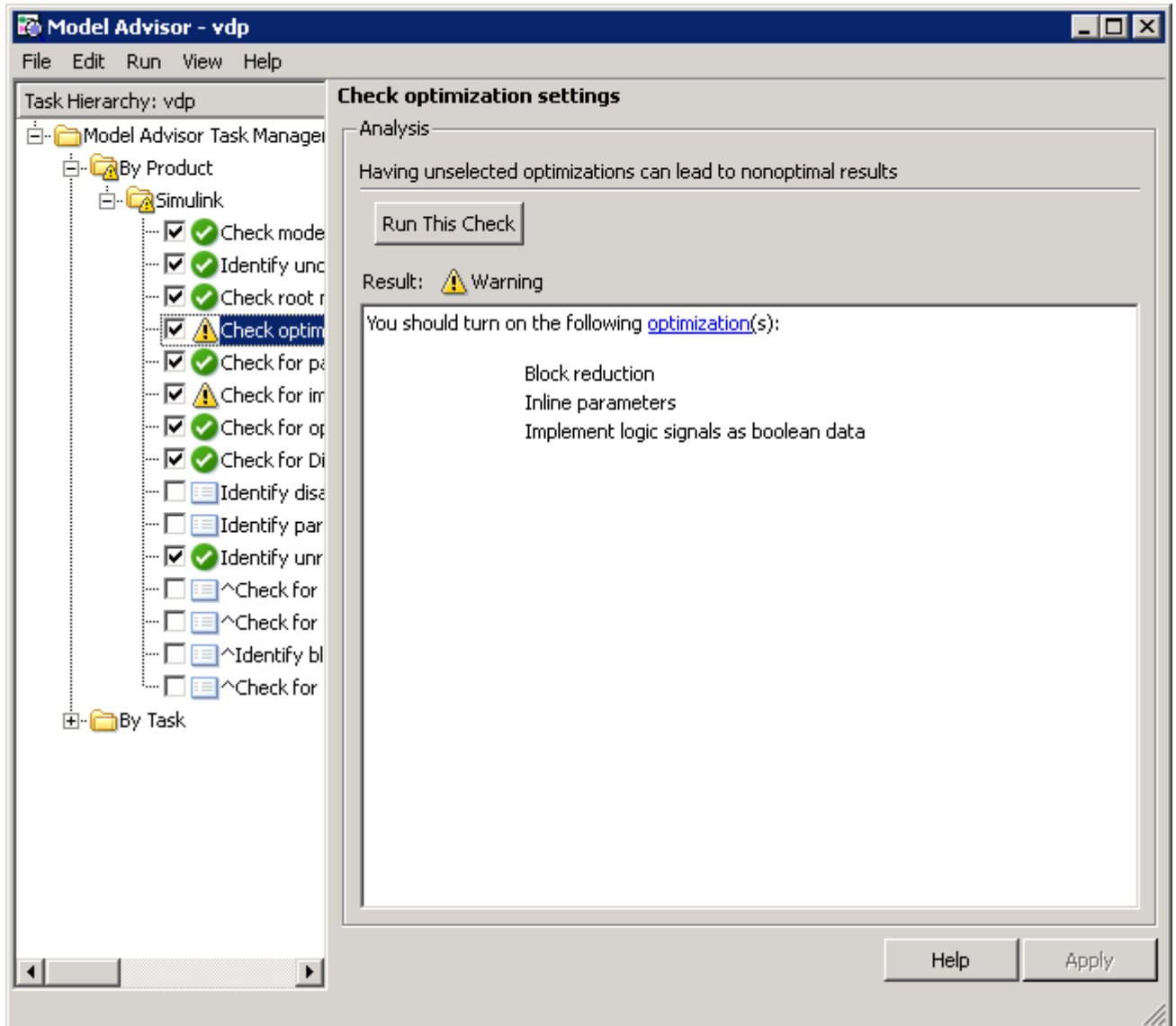
Model Advisor Result Explorer

Some checks in the Model Advisor have an **Explore Result** button that starts the Model Advisor Result Explorer. The Model Advisor Result Explorer allows you to quickly locate, view, and change elements of a model.

In the example below, a default Inport block is added to the vdp model. If you run **Check root model Inport block specifications**, the result is a warning, which enables the **Explore Result** button. Clicking the button opens the Model Advisor Result Explorer window.
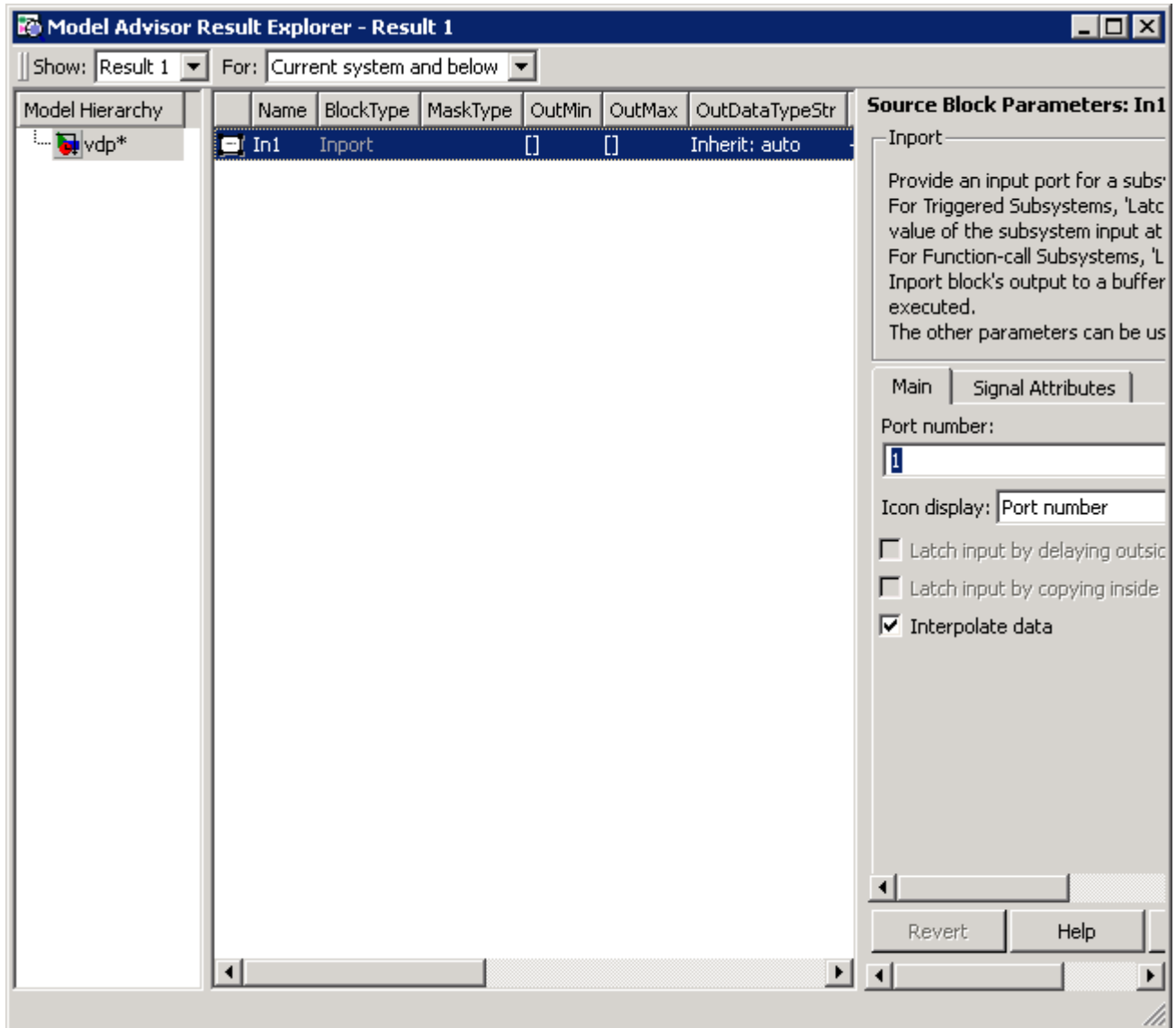
The Model Advisor Result Explorer is a reduced version of the Model Explorer to assist you in modifying only the items that are necessary for the Model

Advisor check. See "The Model Explorer" on page 13-2 for more information about using this window.

Model Advisor Reports

When the Model Advisor runs checks, it generates an HTML report of check results. Each folder in the Model Advisor Task Hierarchy contains a report for all of the items in that folder. You can access any report by selecting a folder and clicking the link in the **Report** box.

Tip While you can use Model Advisor reports to fix checks, you should use the Model Advisor window for interactive check fixing. Use Model Advisor reports to view a summary of checks rather than to fix checks.

As you run checks, the reports are updated with the latest information for each check within the folder. A message appears in the report when the checks are run at different times. The timestamps in the report indicate when checks have been run. All checks have either run during the current run timestamp, indicated in the top right of the report, or during previous runs, indicated by a timestamp following the check name.

Model Advisor -- C:\work\slprj\modeladvisor\vdp\report_2.html

Report name: Model Advisor - **Simulink**

Simulink version: 7.1   **Model version: 1.5**

System: vdp   **Current run: 19-Dec-2007 17:57:00**

ⓘ 3 items with a timestamp different than 19-Dec-2007 17:57:00

Run Summary

Pass	Fail	Warning	Not Run	Total
✅ 1	❌ 0	⚠ 3	📋 11	15

Simulink

✅ **Check model, local libraries, and referenced models for known upgrade issues** (19-Dec-2007 17:56:55)

Passed

⚠ **Identify unconnected lines, input ports, and output ports**

The following unconnected lines and/or ports are likely to cause problems propagating signal attributes (e.g., data type, sample time, dimensions). Note that ports connected to ground/terminator blocks pass this check.

vdp/In1

⚠ **Check root model Inport block specifications** (19-Dec-2007 14:42:49)

Done

Checking Code-Generation Targets

If Real-Time Workshop® software is installed on your system, before running the 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"). The Model Advisor works most effectively with ERT and ERT-based targets (targets based on the Real-Time Workshop® Embedded Coder™ software).

Model Advisor Demo

Enter sldemo_mdladv at the MATLAB command line to run a demo that illustrates usage of the Model Advisor:

If Real-Time Workshop software is installed on your system, the following models illustrate usage of the 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')
```

starts the rtwdemo_advisor1 model. Note that demo models rtwdemo_advisor2 and rtwdemo_advisor3 require Stateflow® and Fixed-Point Toolbox™ software.

Running the Model Advisor Programmatically

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

Managing Model Versions

In this section...
"How Simulink® Helps You Manage Model Versions" on page 3-95
"Model File Change Notification" on page 3-96
"Specifying the Current User" on page 3-97
"Model Properties Dialog Box" on page 3-99
"Creating a Model Change History" on page 3-107
"Version Control Properties" on page 3-108

How Simulink® Helps You Manage Model Versions

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

- Model File Change Notification helps you manage work with source control operations and multiple users

- As you edit a model, the Simulink software generates version control information about the model, including a version number, who created and last updated the model, and an optional change history. The Simulink software automatically saves these **Version Control Properties** with the model

- The Model Properties dialog box lets you edit some of the version control information stored in the model and select various version control options

- The Model Info block lets you display version control information, including information maintained by an external version control system, as an annotation block in a model diagram

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

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

Model File Change Notification

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

Note To programmatically check whether the model has changed on disk since it was loaded, use the function slIsFileChangedOnDisk.

To access the Simulink Preferences window,

- Select **File > Preferences** in the Simulink product.

- Select **File > Preferences** in MATLAB to open the MATLAB Preferences, then select **Simulink** in the left pane, and click the button **Launch Simulink Preferences**.
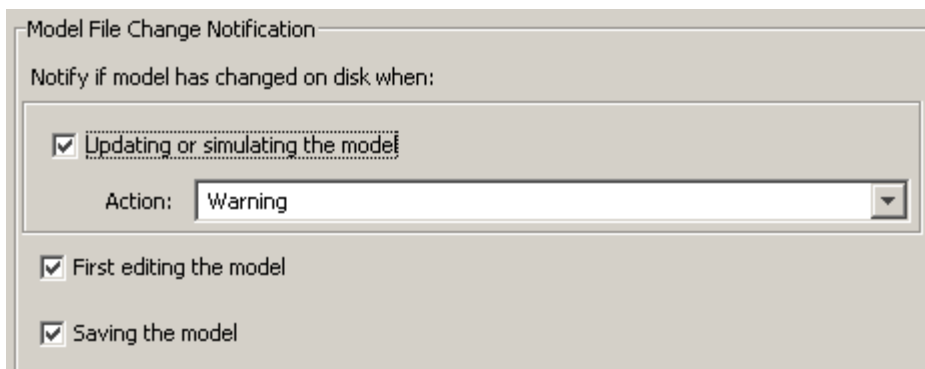
The Model File Change Notification options are in the right pane. You can use the three independent options as follows:

- If you select the **Updating or simulating the model** check box, you can choose what form of notification you want from the **Action** list:

  - Warning — in the MATLAB command window.

- **Error** — in the MATLAB command window if simulating from the command line, or if simulating from a menu item, in the Simulation Diagnostics window.

- **Reload model (if unmodified)** — if the model is modified, you see the prompt dialog. If unmodified, the model is reloaded.

- **Show prompt dialog** — in the dialog, you can choose to close and reload, or ignore the changes.

- If you select the **First editing the model** check box, and the file has changed on disk, and the block diagram is unmodified in Simulink:

  - Any command-line operation that causes the block diagram to be modified (e.g., a call to set_param) will result in a warning:

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

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

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

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

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

Specifying the Current User

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

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

Windows® systems, on the other hand, might define some or none of the "user name" environment variables that the Simulink software expects, depending on the version of Windows installed on your system and whether it is connected to a network. Use the MATLAB command getenv to determine which of the environment variables is defined. For example, enter

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

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

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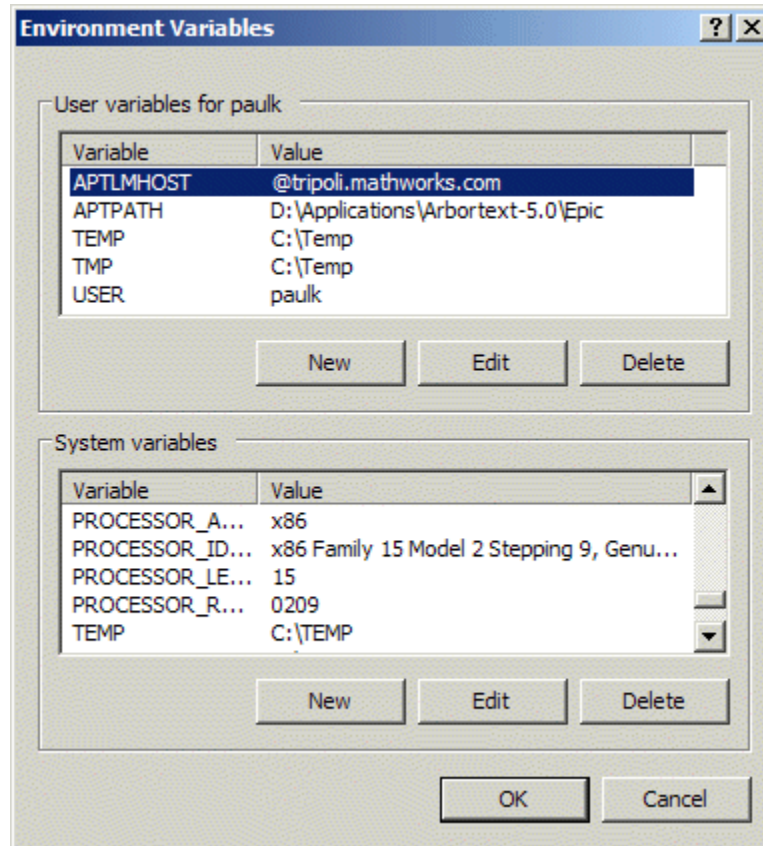
To display the System Properties dialog box, select
Start > Settings > Control Panel to open the Control Panel. Double-click
the **System** icon. To set the USER variable, enter USER in the **Variable** field
and enter your login name in the **Value** field. Click **Set** to save the new
environment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

Model Properties Dialog Box

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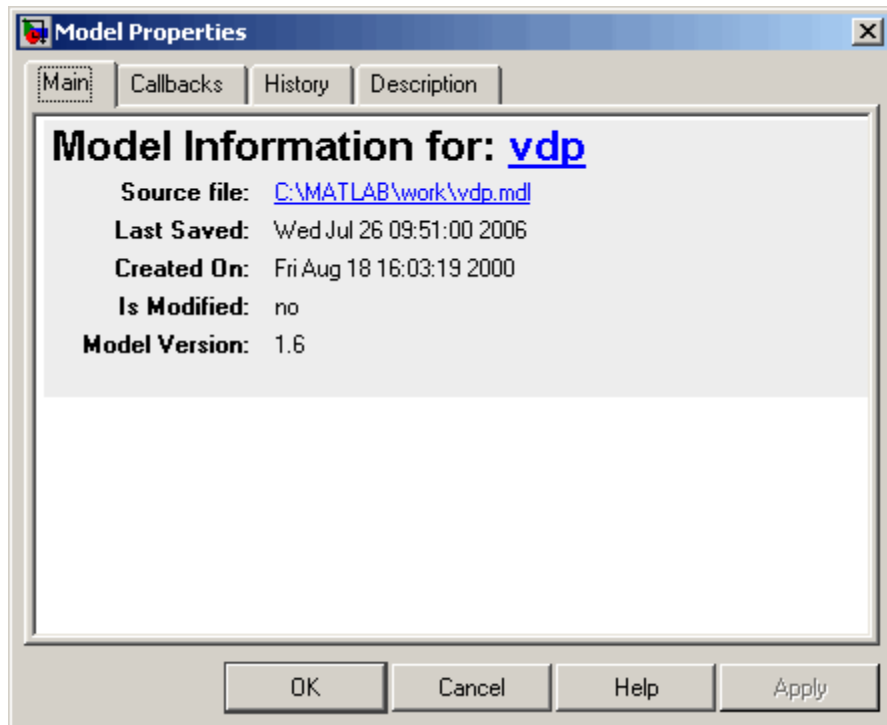
The dialog box includes the following panes.

Main Pane

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

Callbacks Pane

The **Callbacks** pane lets you specify functions to be invoked at specific points in the simulation of the model.
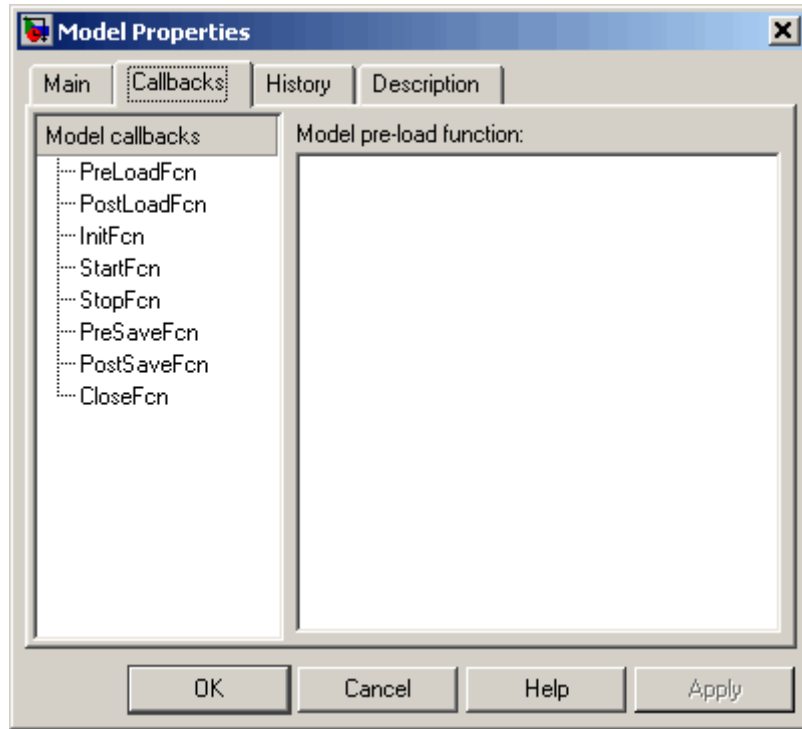
In the left pane, select the callback. In the right pane, enter the name of
the function you want to be invoked for the selected callback. See "Creating
Model Callback Functions" on page 3-53 for information on the callback
functions listed on this pane.

History Pane

The **History** pane allows you to enable, view, and edit this model's change
history.
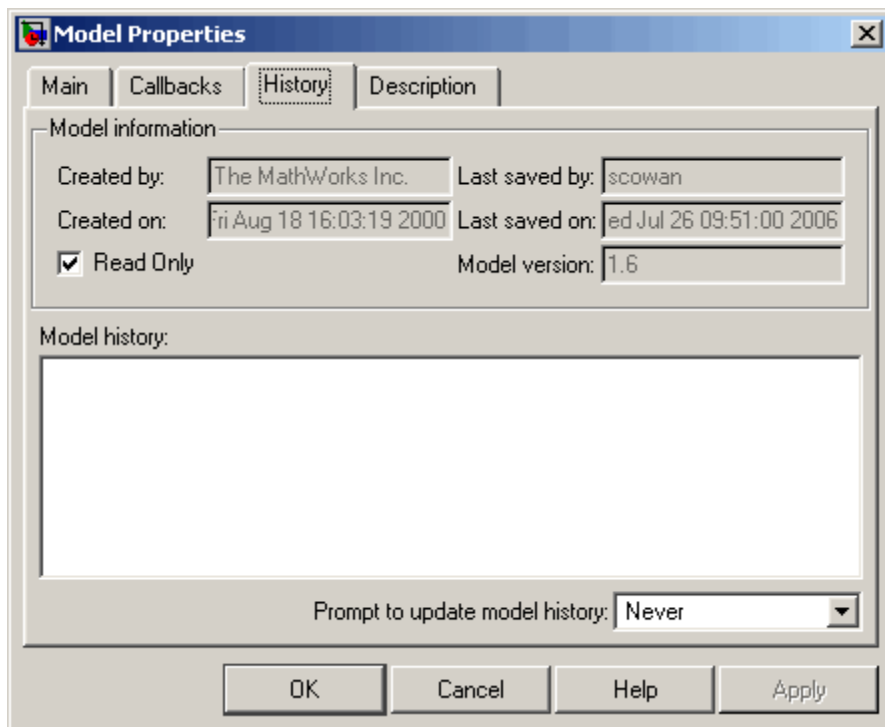
The **History** pane has two control groups: the **Model information** group and the **Model History** group.

Model Information Controls

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

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

• **Created by**

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

- **Created on**

  Date and time this model was created.

- **Last saved by**

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

- **Last saved on**

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

- **Model version**

  Version number for this model.

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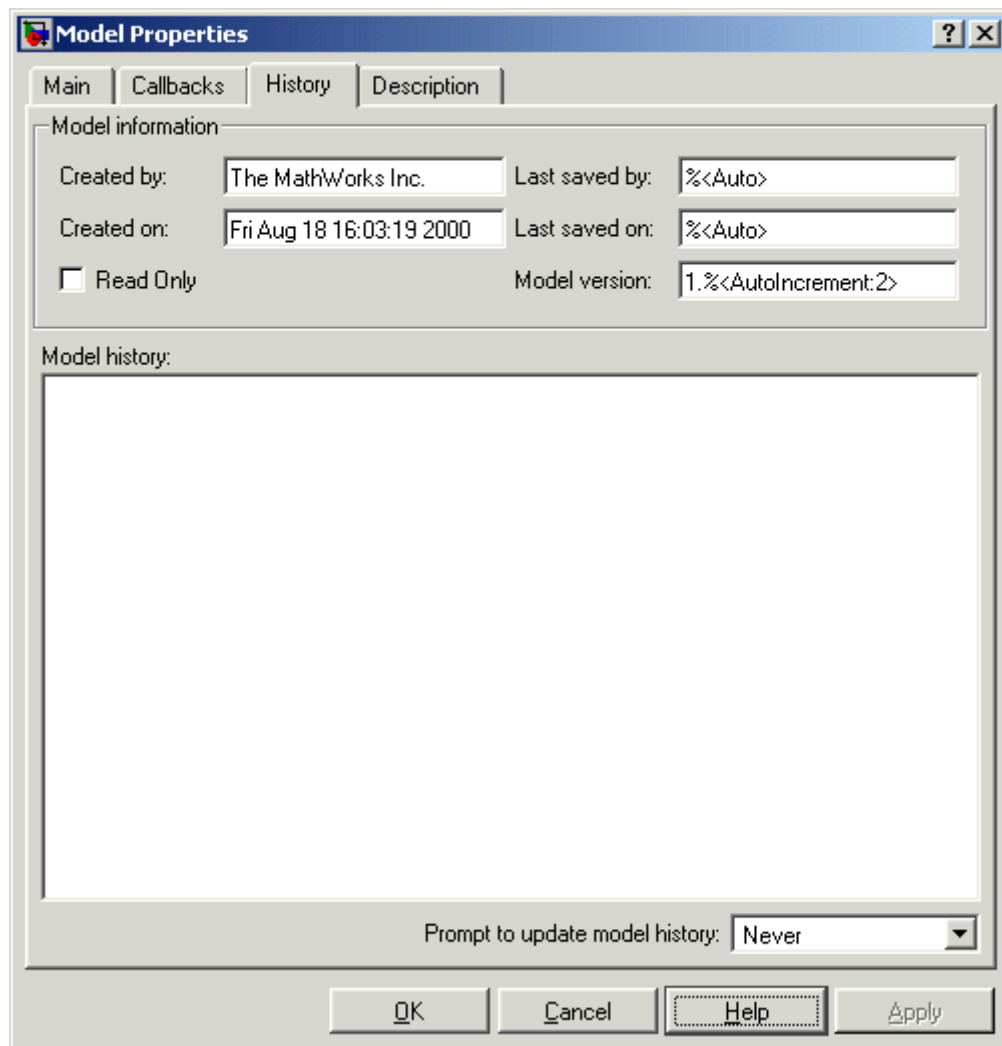
- **Created by**

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

- **Created on**

Date and time this model was created. Do not edit this field.

- **Last saved by**

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

- **Last saved on**

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

- **Model version**

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

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

  as

  ```
  1.2
  ```

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

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

  becomes

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

  and the model version number is reported as `1.3`.

Model History Controls

The model history controls group contains a scrollable text field and an option list. The text field displays the history for the model in a scrollable text field. To change the model history, edit the contents of this field. The option list allows you to enable or disable the Simulink software model history feature. To enable the history feature, select When saving model from the **Prompt to update model history** list. This causes the Simulink software to prompt you to enter a comment when saving the model. Typically you would enter any changes that you have made to the model since the last time you saved it. This information is stored in the model's change history log. See "Creating a Model Change History" on page 3-107 for more information. To disable the change history feature, select Never from the **Prompt to update model history** list.

Model Description Controls

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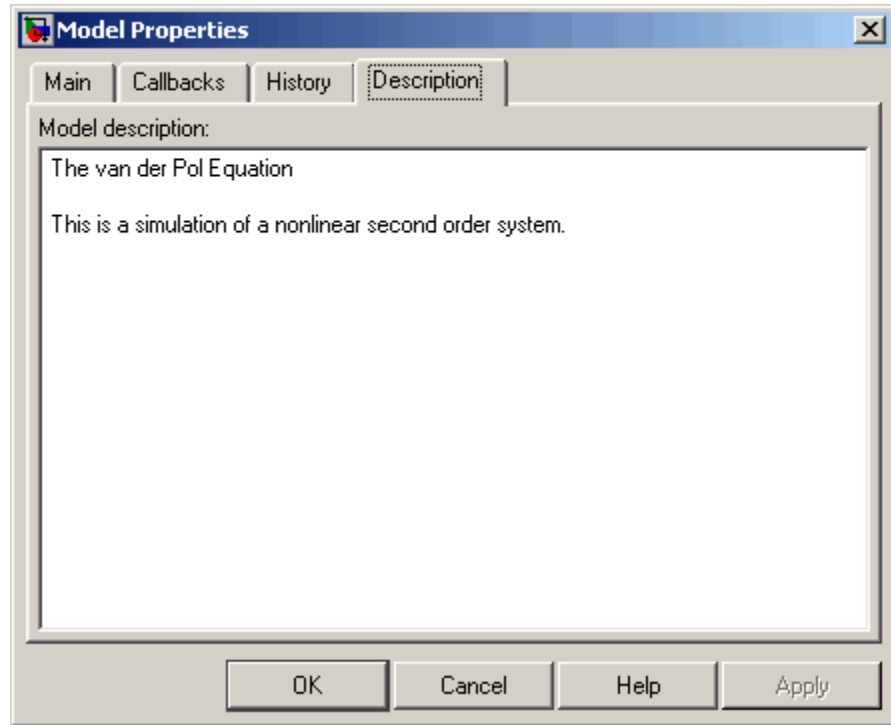
Creating a Model Change History

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

Logging Changes

To start a change history, select When saving model from the **Prompt to update model history** list on the **History** pane on the Model Properties dialog box. The next time you save the model, a **Log Change** dialog box is displayed.
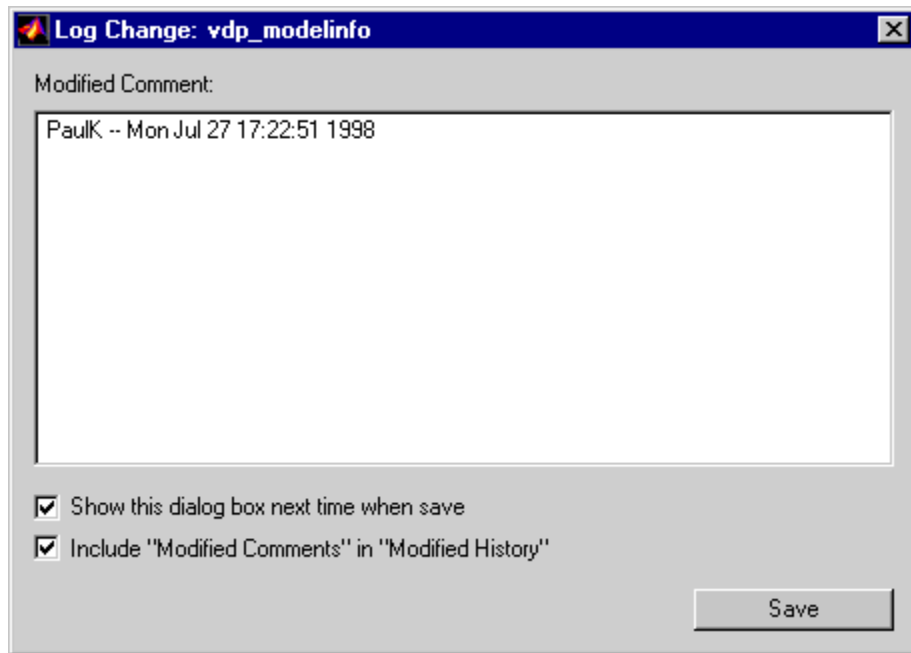
To add an item to the model's change history, enter the item in the **Modified Comments** edit field and click **Save**. If you do not want to enter an item for this session, clear the **Include "Modified Contents" in "Modified History"** option. To discontinue change logging, clear the **Show this dialog box next time when save** option.

Version Control Properties

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

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

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

Model Discretizer

In this section...
"What is the Model Discretizer?" on page 3-110
"Requirements" on page 3-110
"How to Discretize a Model from the Model Discretizer GUI" on page 3-111
"Viewing the Discretized Model" on page 3-120
"How to Discretize Blocks from the Simulink® Model" on page 3-123
"How to Discretize a Model from the MATLAB® Command Window" on page 3-134

What is the Model Discretizer?

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

Model Discretizer enables you to

- Identify a model's continuous blocks.

- Change a block's parameters from continuous to discrete.

- Apply discretization settings to all continuous blocks in the model or to selected blocks.

- Create configurable subsystems that contain multiple discretization candidates along with the original continuous block(s).

- Switch among the different discretization candidates and evaluate the resulting model simulations.

Requirements

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

How to Discretize a Model from the Model Discretizer GUI

To discretize a model:

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

The f14 model, shown below, demonstrates the steps in discretizing a model.
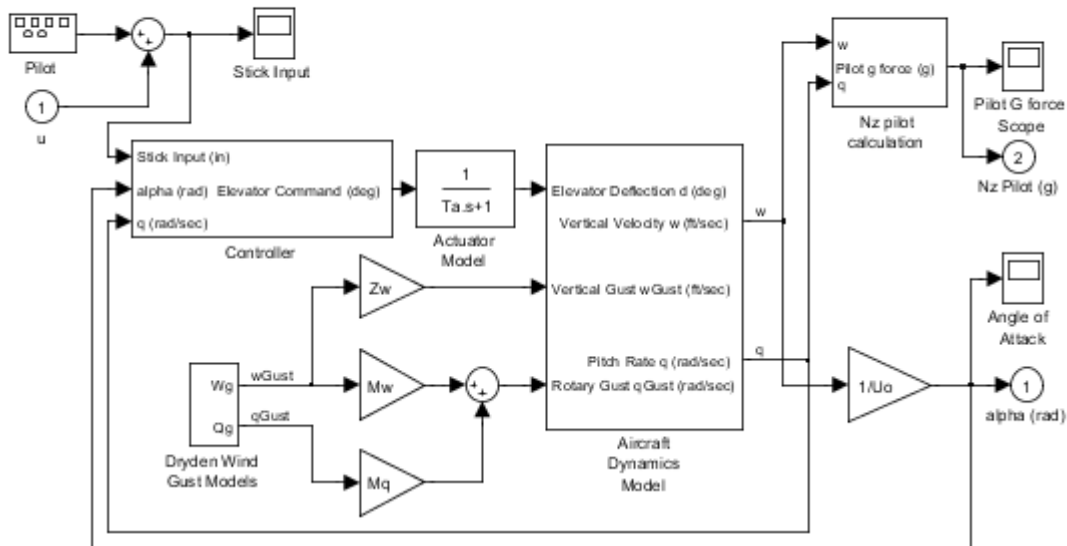
F-14 Flight Control
(an updated version of this demo is available
by running 'sldemo_f14')

Copyright 1990-2005 The MathWorks Inc.

Start Model Discretizer

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

The **Simulink Model Discretizer** appears.
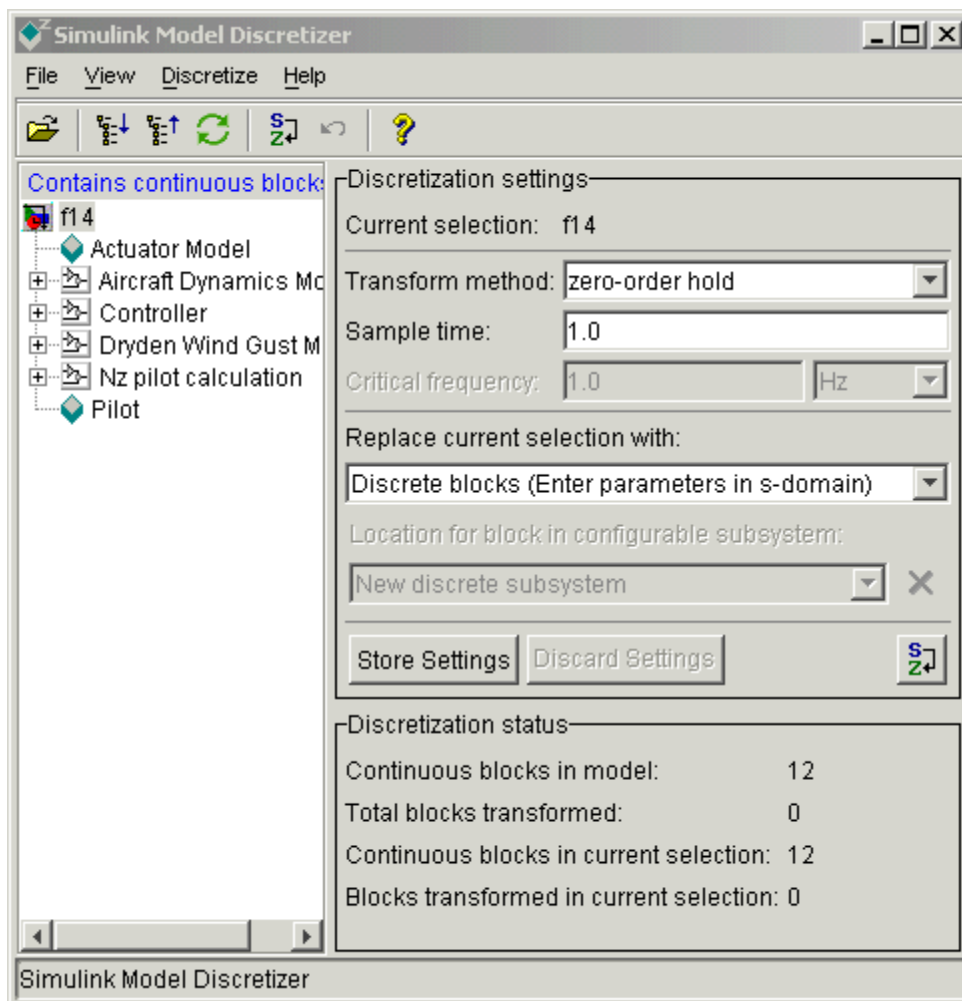
Alternatively, you can open Model Discretizer from the MATLAB® Command Window using the slmdldiscui function.

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

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

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

Specify the Transform Method

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

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

- zero-order hold

  Zero-order hold on the inputs.

- first-order hold

  Linear interpolation of inputs.

- tustin

  Bilinear (Tustin) approximation.

- tustin with prewarping

  Tustin approximation with frequency prewarping.

- matched pole-zero

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

Specify the Sample Time

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

You can specify an offset time by entering a two-element vector for discrete blocks or configurable subsystems. The first element is the sample time and the second element is the offset time. For example, an entry of [1.0 0.1]

would specify a 1.0 second sample time with a 0.1 second offset. If no offset is specified, the default is zero.

You can enter workspace variables when discretizing blocks in the s-domain. See "Discrete blocks (Enter parameters in s-domain)" on page 3-115.

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 3-115

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

- "Discrete blocks (Enter parameters in z-domain)" on page 3-116

  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 3-117

  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 3-118

  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 3-114.

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

The following figure shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the s-domain. The **Block Parameters** dialog box for each block appears below the block.
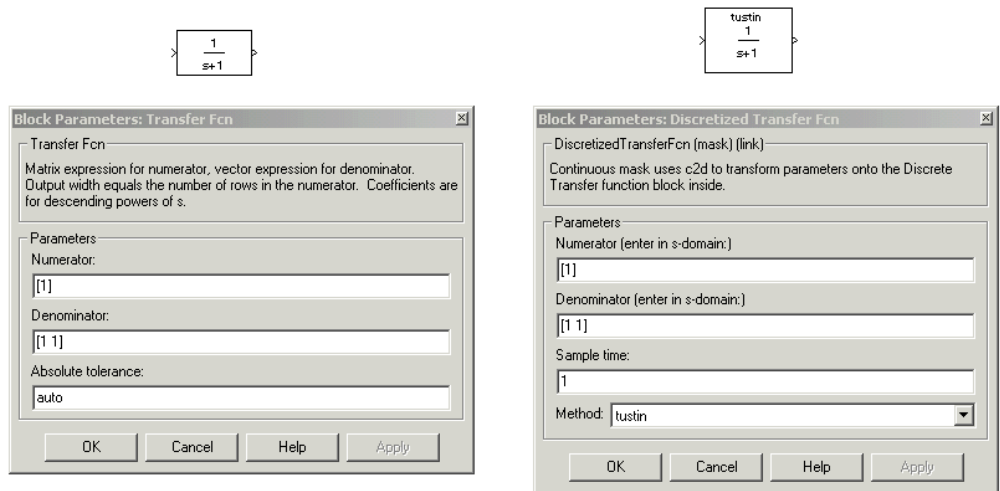
Discrete blocks (Enter parameters in z-domain). Creates a discrete block whose parameters are "hard-coded" values placed directly into the block's dialog box. Model Discretizer uses the c2d function to obtain the discretized parameters, if needed.

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

```
help c2d
```

The following figure shows a continuous Transfer Function block next to a Transfer Function block that has been discretized in the z-domain. The Block Parameters dialog box for each block appears below the block.
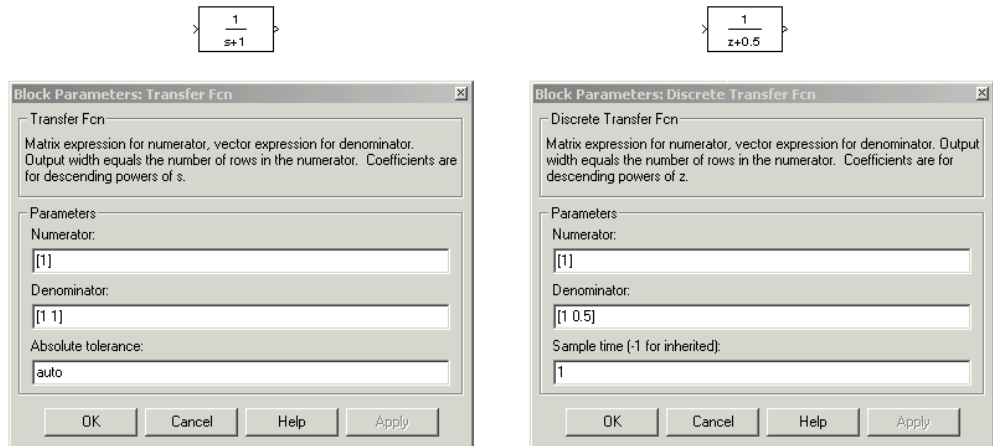
Note If you want to recover exactly the original continuous parameter values after the Model Discretization session, you should enter parameters in the s-domain.

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

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

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

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

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

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

Configurable subsystems are stored in a library containing the discretization
candidates and the original continuous block. The library will be named
<model name>_disc_lib and it will be stored in the current directory. For
example a library containing a configurable subsystem created from the f14
model will be named f14_disc_lib.

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

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

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

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

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

There are two methods for discretizing blocks.

Select Blocks and Discretize.

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

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

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

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

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

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

1 Enter the discretization settings for the current block.

2 Click **Store Settings**.

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

3 Repeat steps 1 and 2, as necessary.

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

Deleting a Discretization Candidate from a Configurable Subsystem

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

Undoing a Discretization

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

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

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

Viewing the Discretized Model

Model Discretizer displays the model in a hierarchical tree view.

Viewing Discretized Blocks

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

The following figure shows that the Aircraft Dynamics Model subsystem has been discretized into a configurable subsystem with three discretization candidates.
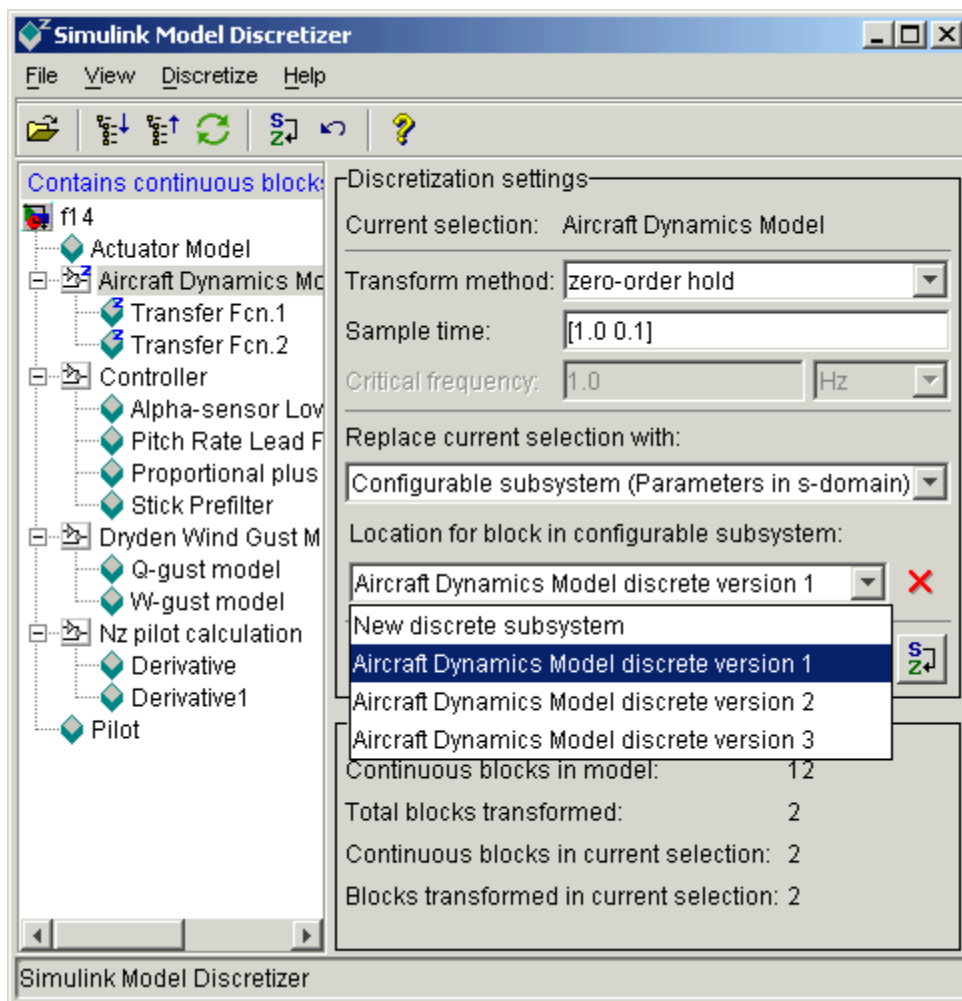
The other blocks in this f14 model have not been discretized.

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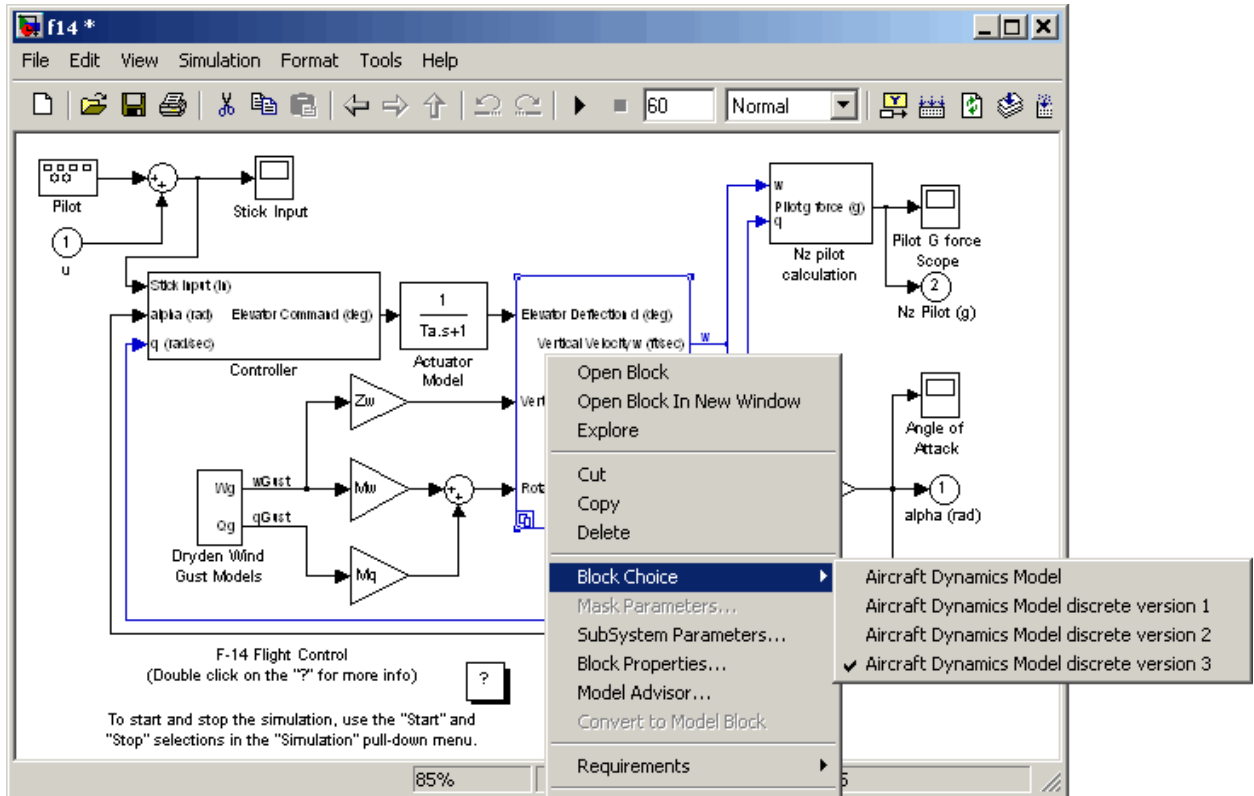
The following figure shows the library containing the Aircraft Dynamics Model configurable subsystem with the original continuous model and three discretization candidates.
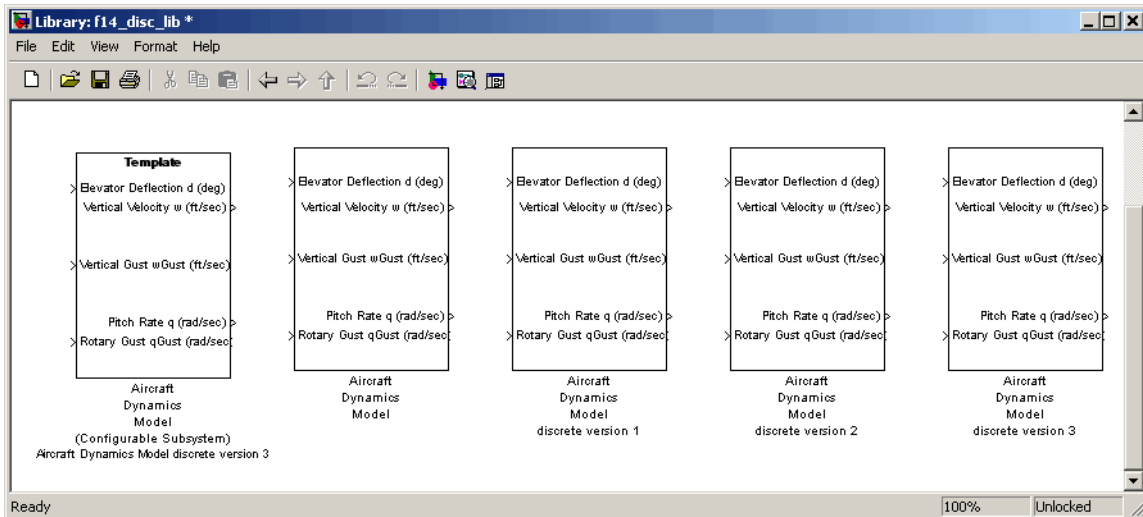
Refreshing Model Discretizer View of the Model

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

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

How to Discretize Blocks from the Simulink® Model

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

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

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

1 Open the `f14` model.

2 Open the Aircraft Dynamics Model subsystem in the `f14` model.
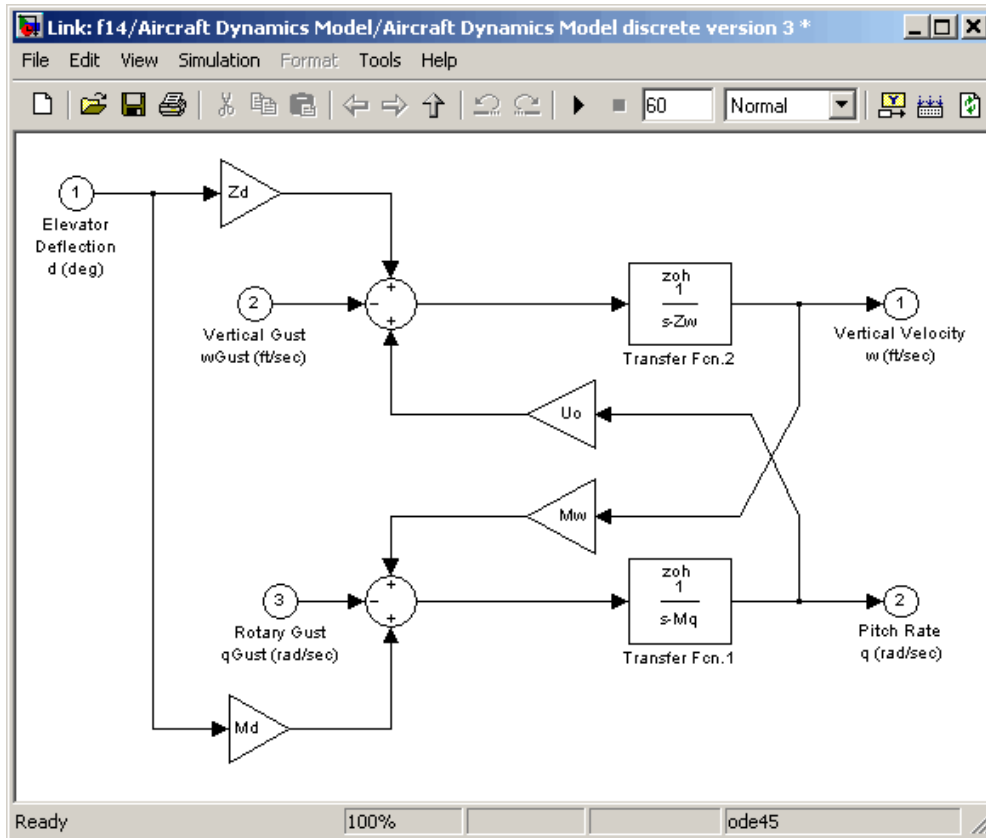
3 Open the Discretizing library window.

Enter `discretizing` at the MATLAB command prompt.

The **Library: discretizing** window opens.
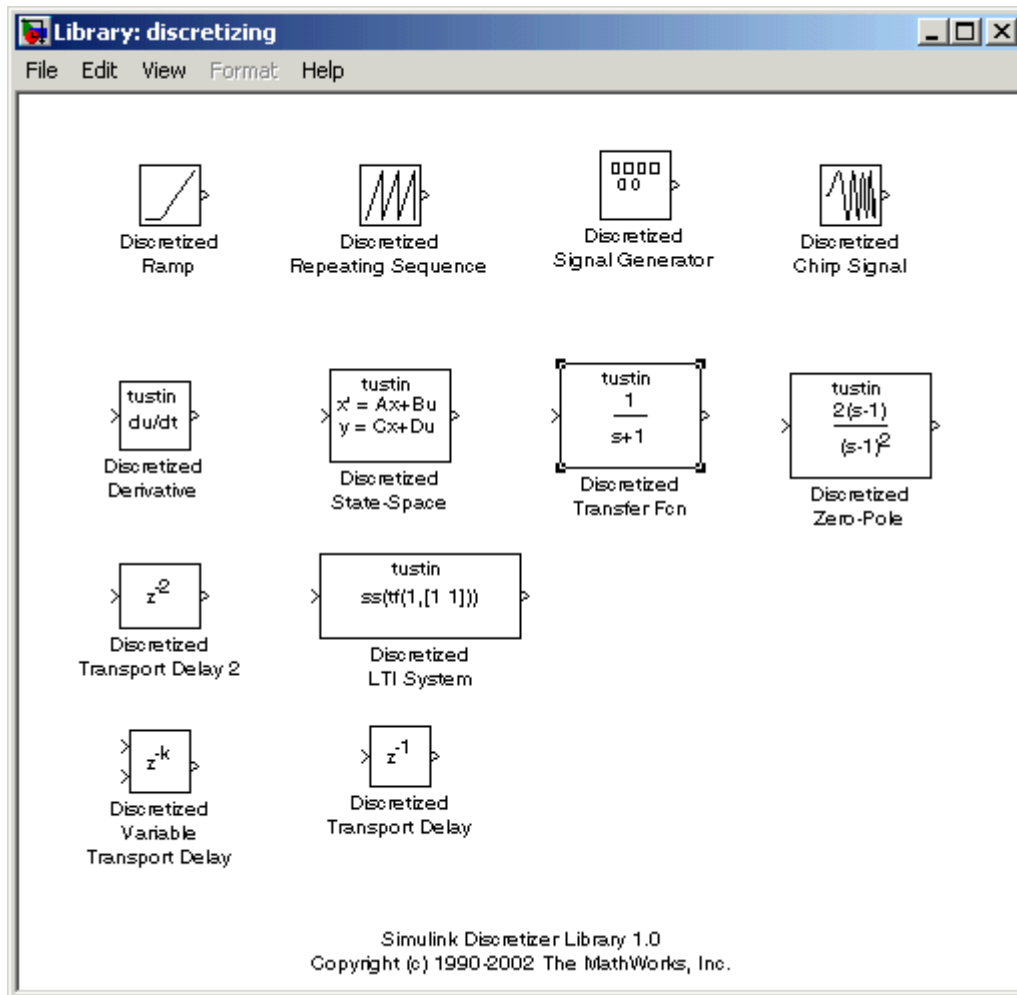
This library contains s-domain discretized blocks.

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

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

b Drag it into the **f14/Aircraft Dynamics Model** window.
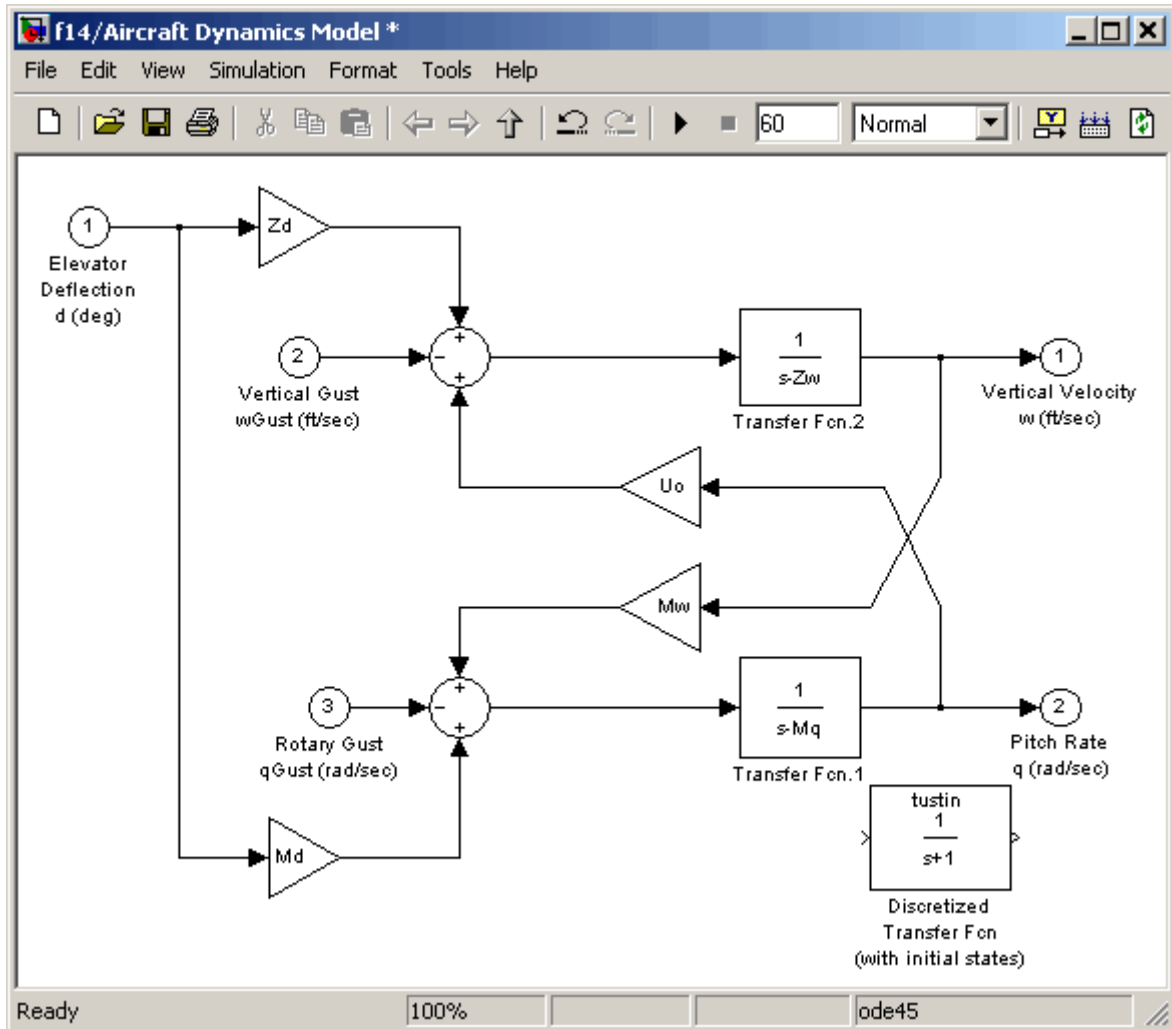
5 Open the parameter dialog box for the Transfer Fcn.1 block.

Double-click the Transfer Fcn.1 block in the **f14/Aircraft Dynamics Model** window.

The Block Parameters: Transfer Fcn.1 dialog box opens.
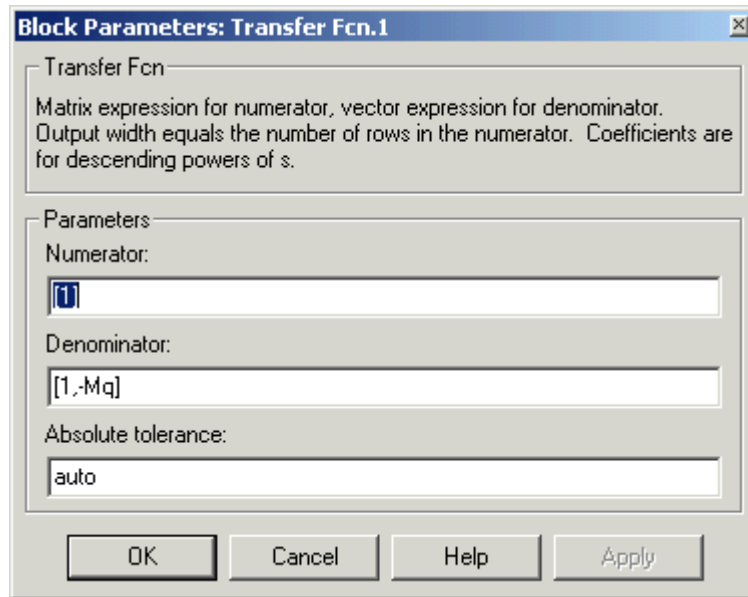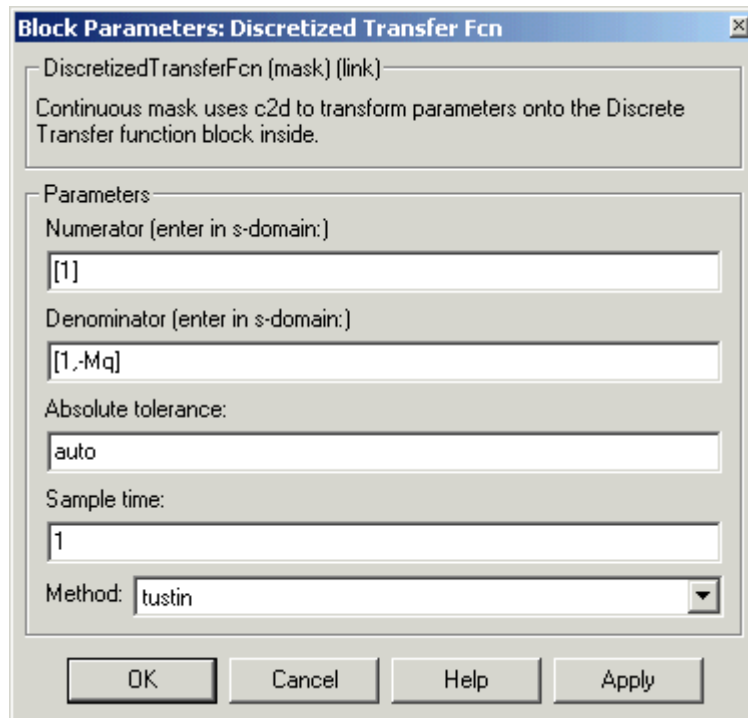
6 Open the parameter dialog box for the Discretized Transfer Fcn block.

Double-click the Discretized Transfer Fcn block in the **f14/Aircraft Dynamics Model** window.

The Block Parameters: Discretized Transfer Fcn dialog box opens.
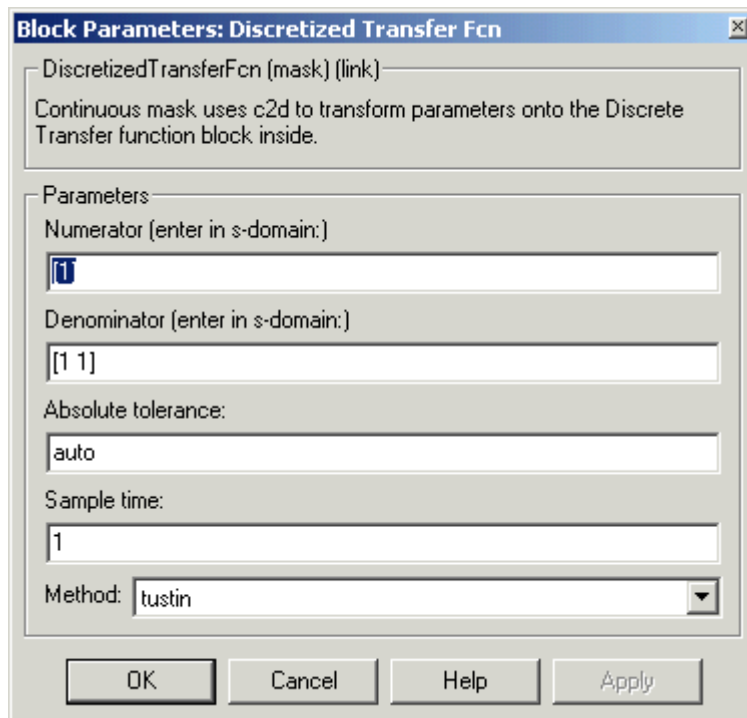
Copy the parameter information from the Transfer Fcn.1 block's dialog box to the Discretized Transfer Fcn block's dialog box.

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

OK   Cancel   Help   Apply

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

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

The parameter dialog box for the Discretized Transfer Fcn. now looks like this.
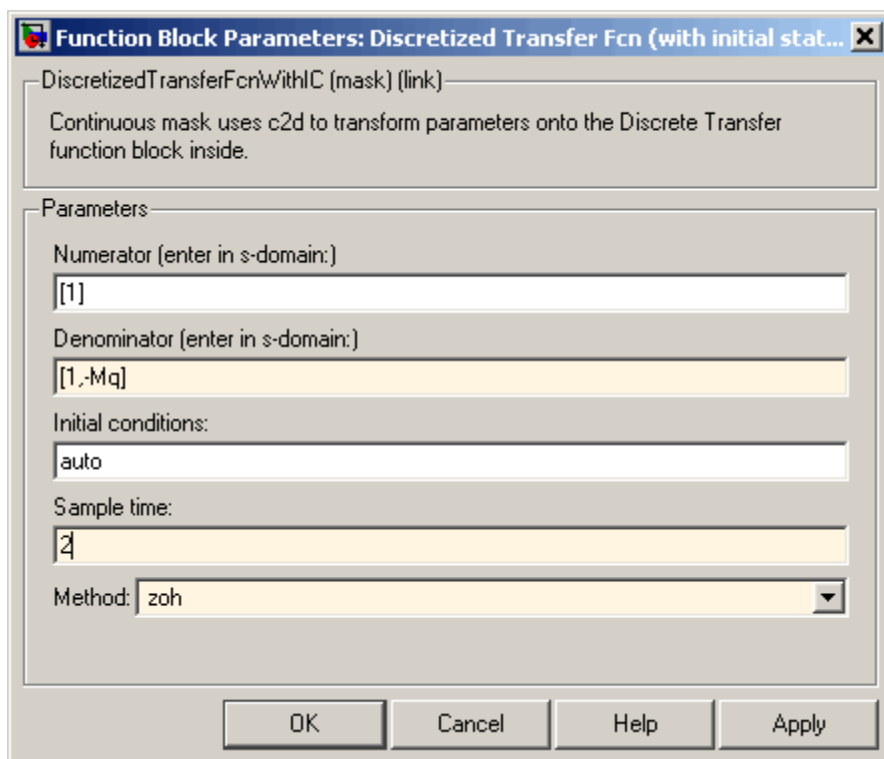
9 Click **OK**.

The **f14/Aircraft Dynamics Model** window now looks like this.
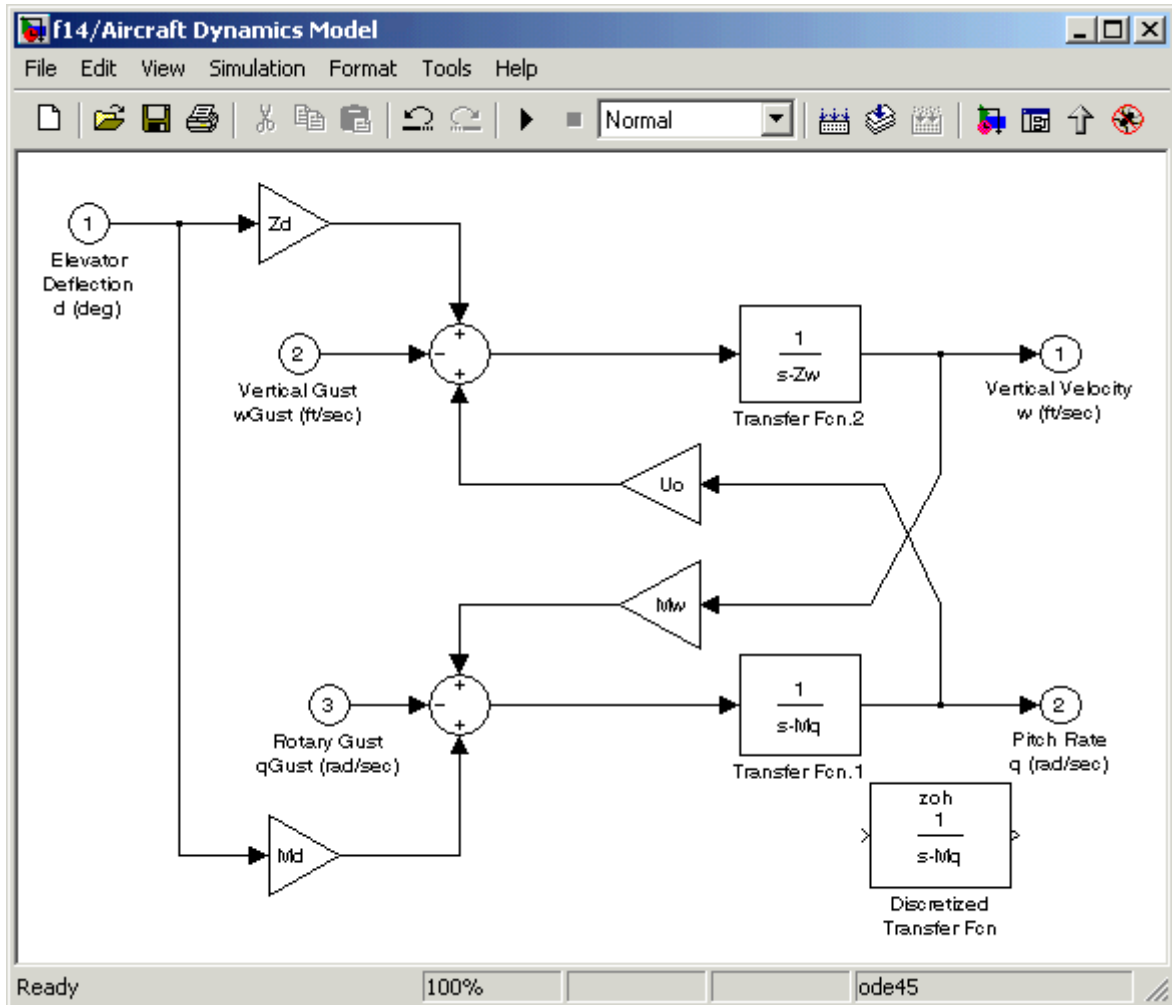
10 Delete the original Transfer Fcn.1 block.

    a Click the Transfer Fcn.1 block.

    b Press the **Delete** key.

The **f14/Aircraft Dynamics Model** window now looks like this.
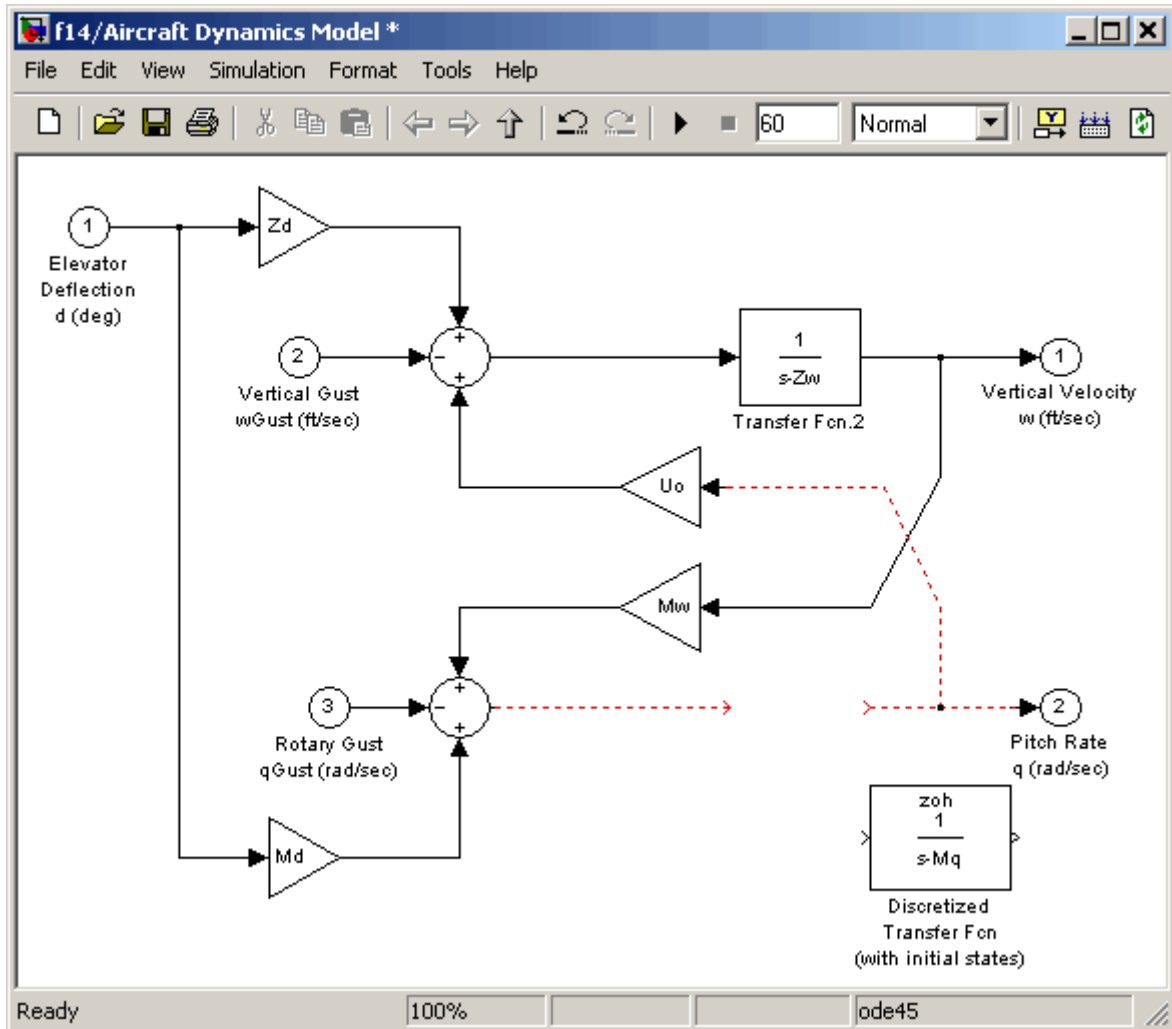
11 Add the Discretized Transfer Fcn block to the model.

    a Click the Discretized Transfer Fcn block.

    b Drag the Discretized Transfer Fcn block into position to complete the
model.

The **f14/Aircraft Dynamics Model** window now looks like this.
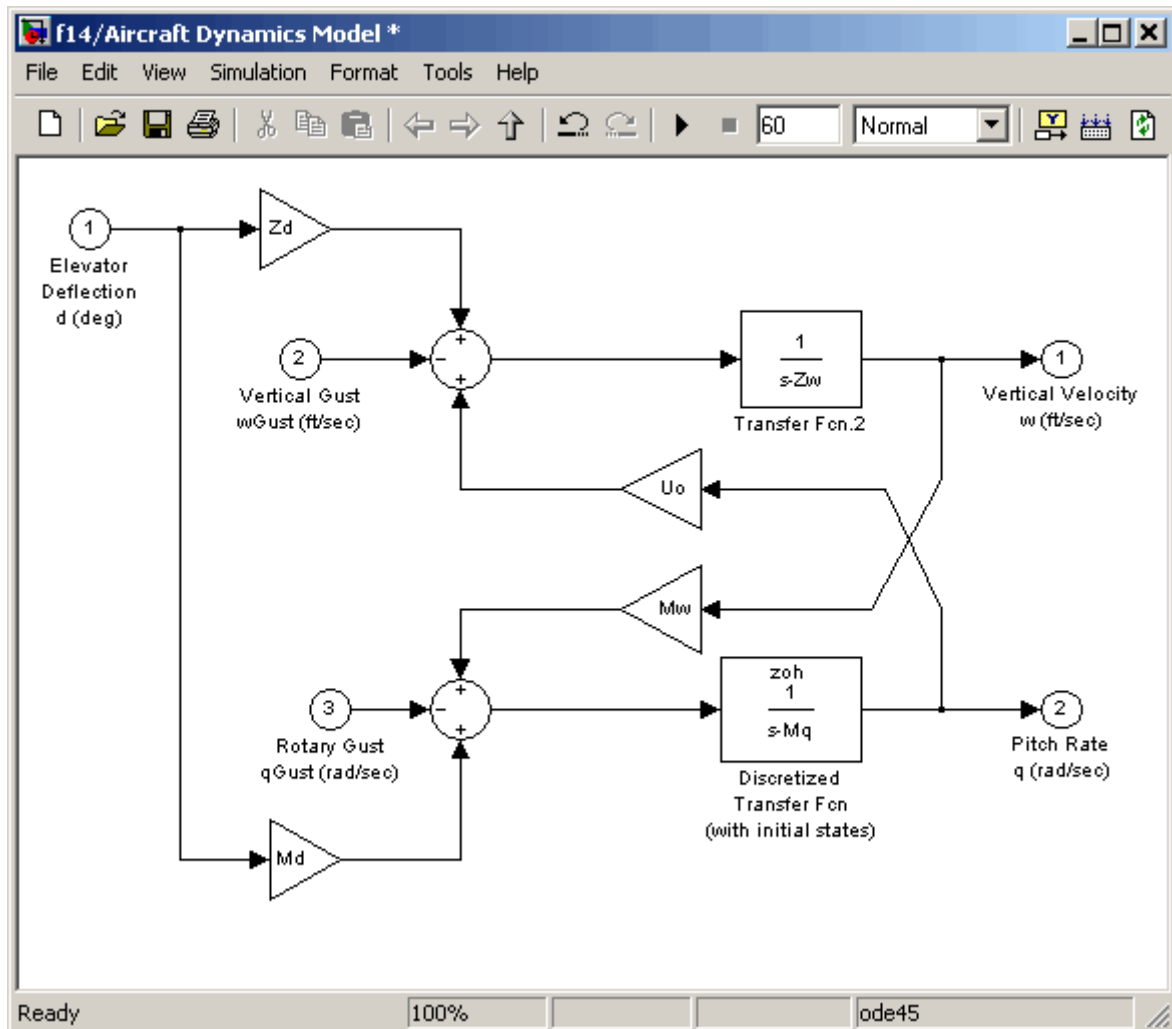
How to Discretize a Model from the MATLAB® Command Window

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

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

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

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

4

Creating Conditional Subsystems

About Conditional Subsystems

A subsystem is a set of blocks that have been replaced by a single block called a Subsystem block. This chapter describes a special kind of subsystem whose execution can be externally controlled. For information that applies to all subsystems, see "Creating Subsystems" on page 3-35.

A *conditional subsystem*, also known as a *conditionally executed subsystem*, is a subsystem whose execution depends on the value of an input signal. The signal that controls whether a subsystem executes is called the *control signal*. The signal enters the Subsystem block at the *control input*.

Conditional subsystems can be very useful when you are building complex models that contain components whose execution depends on other components. The following types of conditional subsystems are supported:

- An *enabled subsystem* executes while the control signal is positive. It starts execution at the time step where the control signal crosses zero (from the negative to the positive direction) and continues execution while the control signal remains positive. Enabled subsystems are described in more detail in "Enabled Subsystems" on page 4-4.

- A *triggered subsystem* executes once each time a trigger event occurs. A trigger event can occur on the rising or falling edge of a trigger signal, which can be continuous or discrete. Triggered subsystems are described in more detail in "Triggered Subsystems" on page 4-12.

- A *triggered and enabled subsystem* executes once on the time step when a trigger event occurs if the enable control signal has a positive value at that step. See "Triggered and Enabled Subsystems" on page 4-16 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 3-42 for more information.

Note The Simulink® software imposes restrictions on connecting blocks with a constant sample time to the output port of a conditional subsystem. See "Using Blocks with Constant Sample Times in Enabled Subsystems" on page 4-9 for more information.

Enabled Subsystems

In this section...
"Creating an Enabled Subsystem" on page 4-5
"Blocks an Enabled Subsystem Can Contain" on page 4-7
"Using Blocks with Constant Sample Times in Enabled Subsystems" on page 4-9

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

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

- If the input is a scalar, the subsystem executes if the input value is greater than zero.

- If the input is a vector, the subsystem executes if *any* of the vector elements is greater than zero.

For example, if the control input signal is a sine wave, the subsystem is alternately enabled and disabled, as shown in this figure. An up arrow signifies enable, a down arrow disable.
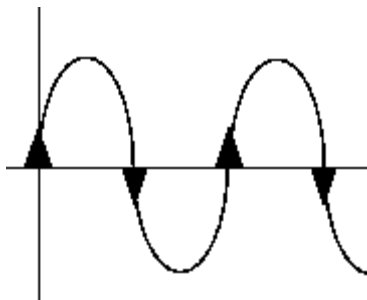
The Simulink® software uses the zero-crossing slope method to determine whether an enable is to occur. If the signal crosses zero and the slope is positive, the subsystem is enabled. If the slope is negative at the zero crossing, the subsystem is disabled.

Creating an Enabled Subsystem

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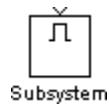
Subsystem

Setting Output Values While the Subsystem Is Disabled

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

Open each Outport block's dialog box and select one of the choices for the **Output when disabled** parameter, as shown in the 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.
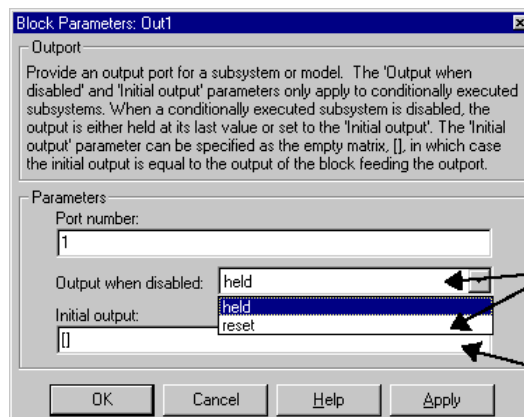
Select an option to set the Outport output while the subsystem is disabled.

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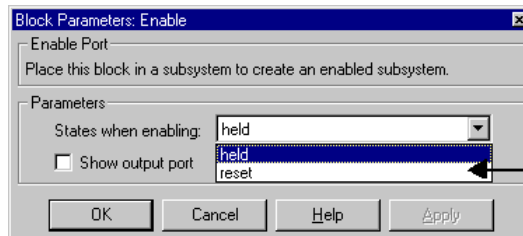
Select an option to set the states when the subsystem is reenabled.

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.
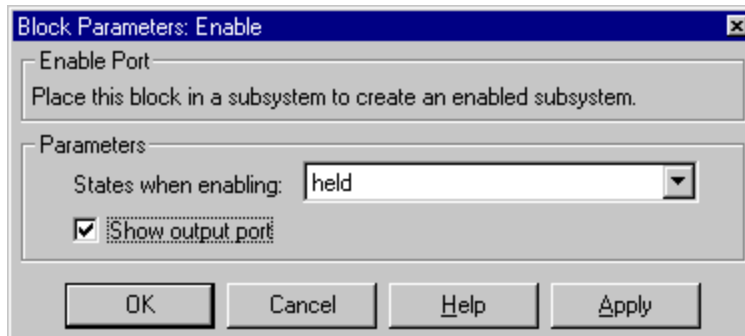
This feature allows you to pass the control signal down into the enabled subsystem, which can be useful where logic within the enabled subsystem is dependent on the value or values contained in the control signal.

Blocks an Enabled Subsystem Can Contain

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

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

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

- Block A, which has a sample time of 0.25 second

- Block B, which has a sample time of 0.5 second

- Block C, within the enabled subsystem, which has a sample time of 0.125 second

- Block D, also within the enabled subsystem, which has a sample time of 0.25 second

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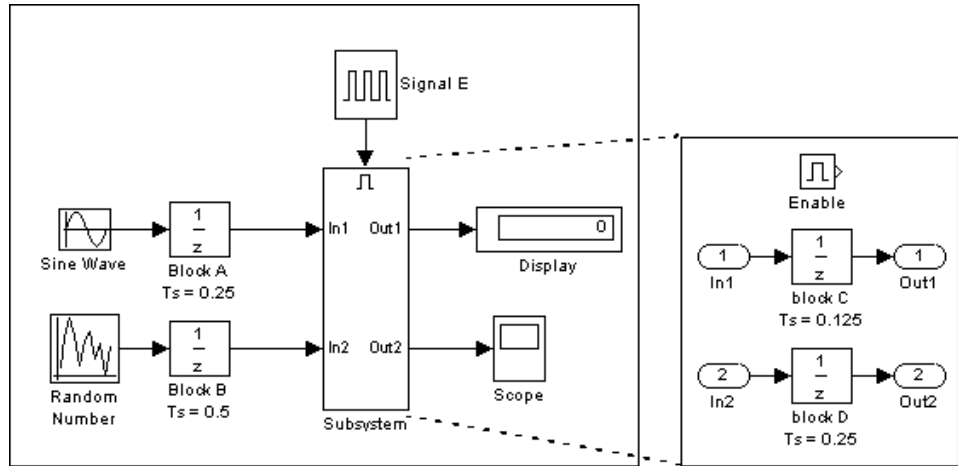
The chart below indicates when the discrete blocks execute.
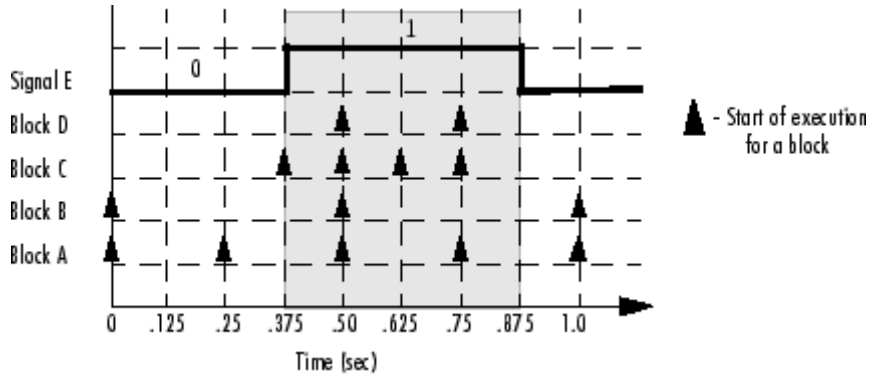
Blocks A and B execute independently of the enable control signal because they are not part of the enabled subsystem. When the enable control signal becomes positive, blocks C and D execute at their assigned sample rates until the enable control signal becomes zero again. Note that block C does not execute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

Using Blocks with Constant Sample Times in Enabled Subsystems

Certain restrictions are placed on connecting blocks with constant sample times (see "Constant Sample Time" on page 2-49) to the output port of a conditional subsystem.

- An error is displayed if you connect a Model or S-Function block with constant sample time to the output port of a conditional subsystem.

- The sample time of any built-in block with a constant sample time is converted to a different sample time, such as the fastest discrete rate in the conditional subsystem.

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

Consider the following model m1.mdl.
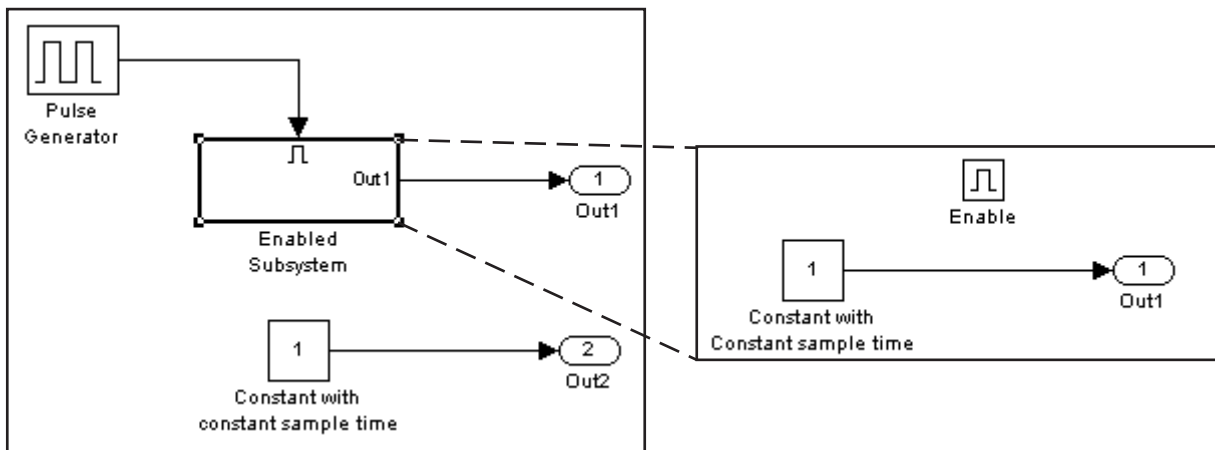
The two Constant blocks in this model have constant sample times. When you simulate the model, the Simulink software converts the sample time of the Constant block inside the enabled subsystem to the rate of the Pulse Generator. If you simulate the model with sample time colors displayed (see "Displaying Sample Time Colors" on page 3-10), The Pulse Generator and

Enabled Subsystem in red. However, the Constant and Outport blocks outside of the enabled subsystem are colored magenta, indicating that these blocks still have a constant sample time.

Suppose the model above is referenced from a Model block inside an enabled subsystem in a top-level model, as shown below. (See Chapter 5, "Referencing a Model".)
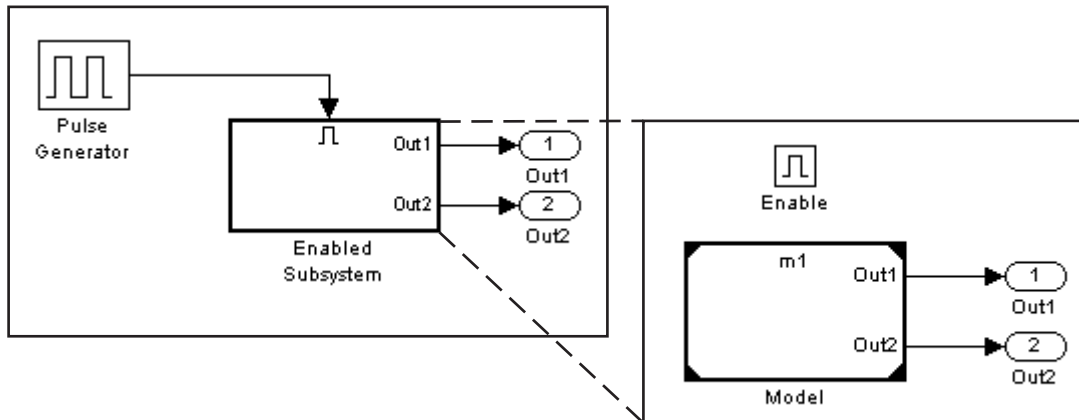
An error is invoked when you try to simulate the top model, indicating that the second output of the Model block may not be wired directly to the enabled subsystem's output port because it has a constant sample time. See Chapter 5, "Referencing a Model".

To avoid this error, insert a Signal Conversion block between the second output of the Model block and the enabled subsystem's Outport block, as shown below.
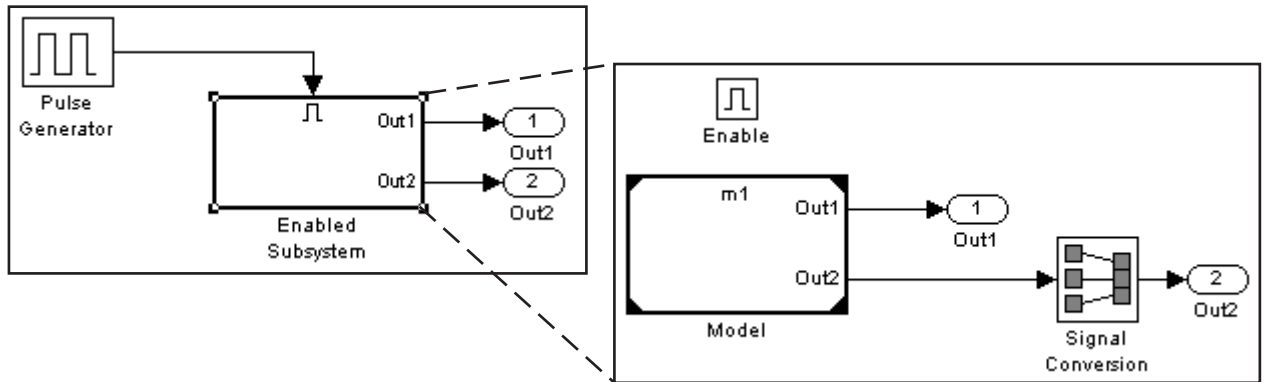
This model is run with no errors. With sample time colors displayed, the Model and Enabled Subsystem blocks are colored yellow, indicating that these are hybrid systems, that is, systems that contain multiple sample times.

Triggered Subsystems

In this section...
"Creating a Triggered Subsystem" on page 4-13
"Blocks That a Triggered Subsystem Can Contain" on page 4-15

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

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

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

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

- `either` triggers execution of the subsystem when the signal is either rising or falling.

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

For example, in the following timing diagram for a discrete system, a rising trigger (R) does not occur at time step 3 because the signal has remained at zero for only one time step when the rise occurs.
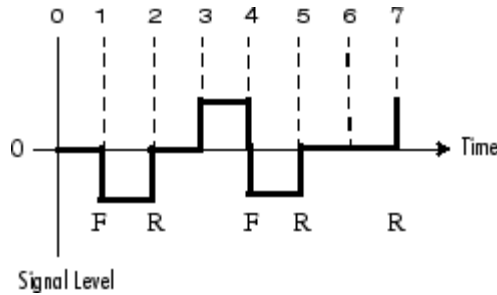
A simple example of a triggered subsystem is illustrated.
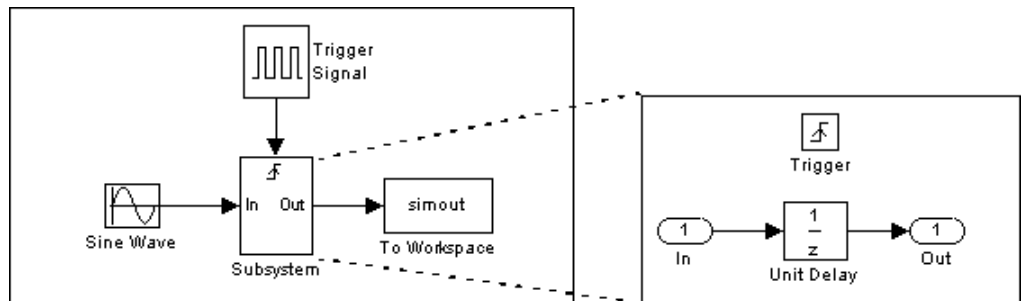
In this example, the subsystem is triggered on the rising edge of the square wave trigger control signal.

Creating a Triggered Subsystem

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

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

Select the trigger type.
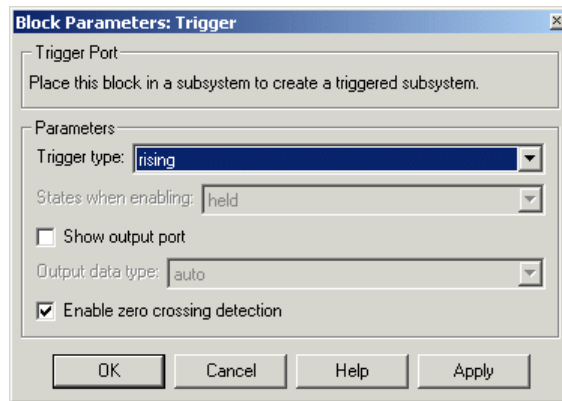
Different symbols are used on the Trigger and Subsystem blocks to indicate rising and falling triggers (or either). This figure shows the trigger symbols on Subsystem blocks.

Subsystem with Rising trigger

Subsystem with Falling trigger

Subsystem with Rising or Falling trigger
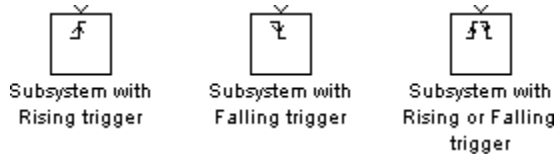
Outputs and States Between Trigger Events

Unlike enabled subsystems, triggered subsystems always hold their outputs at the last value between triggering events. Also, triggered subsystems cannot reset their states when triggered; states of any discrete blocks are held between trigger events.

Outputting the Trigger Control Signal

An option on the Trigger block dialog box lets you output the trigger control signal. To output the control signal, select the **Show output port** check box.

Block Parameters: Trigger

─ Trigger Port ─
Place this block in a subsystem to create a triggered subsystem.

─ Parameters ─
Trigger type: rising

States when enabling: held
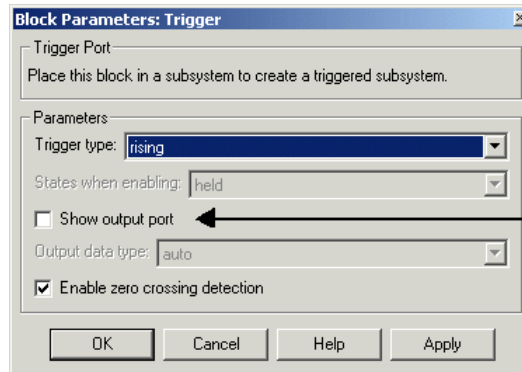
☐ Show output port ◄──────── Select this check box to show the output port.

Output data type: auto

☑ Enable zero crossing detection

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

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

In this section...
"Creating a Triggered and Enabled Subsystem" on page 4-17
"A Sample Triggered and Enabled Subsystem" on page 4-18
"Creating Alternately Executing Subsystems" on page 4-18

A third kind of conditional subsystem combines both types of conditional execution. The behavior of this type of subsystem, called a *triggered and enabled* subsystem, is a combination of the enabled subsystem and the triggered subsystem, as shown by this flow diagram.
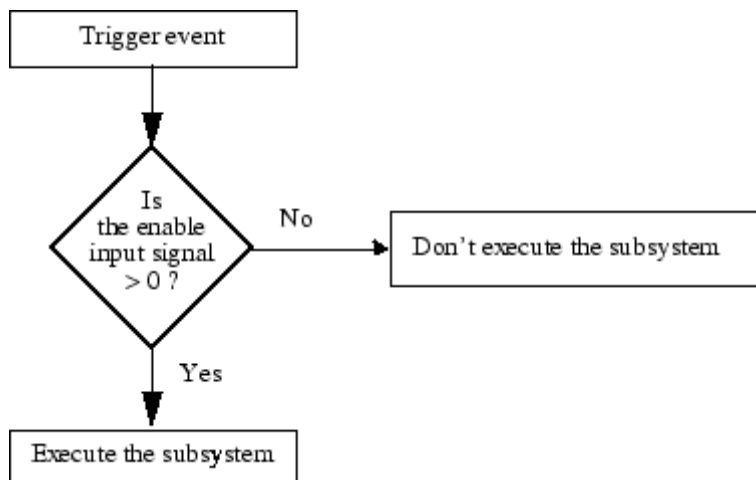
A triggered and enabled subsystem contains both an enable input port and a trigger input port. When the trigger event occurs, the enable input port is checked to evaluate the enable control signal. If its value is greater than zero, the subsystem is executed. If both inputs are vectors, the subsystem executes if at least one element of each vector is nonzero.

The subsystem executes once at the time step at which the trigger event occurs.

Creating a Triggered and Enabled Subsystem

You create a triggered and enabled subsystem by dragging both the Enable and Trigger blocks from the Ports & Subsystems library into an existing subsystem. The Simulink® software adds enable and trigger symbols and enable and trigger and enable control inputs to the Subsystem block.

Subsystem

You can set output values when a triggered and enabled subsystem is disabled as you would for an enabled subsystem. For more information, see "Setting Output Values While the Subsystem Is Disabled" on page 4-5. Also, you can specify what the values of the states are when the subsystem is reenabled. See "Setting States When the Subsystem Becomes Reenabled" on page 4-6.

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.
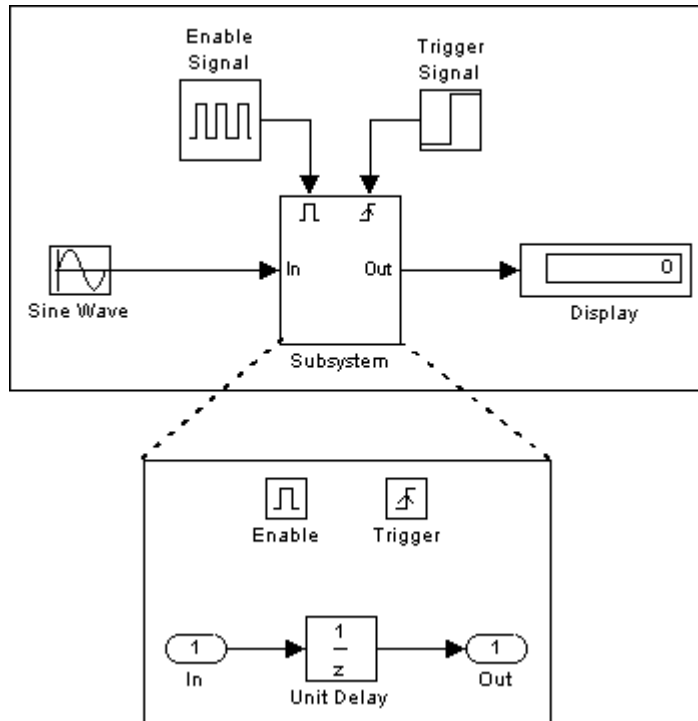
Creating Alternately Executing Subsystems

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

The following figure shows a model that uses two enabled blocks and a Merge block to model a full-wave rectifier – a device that converts AC current to pulsating DC current.
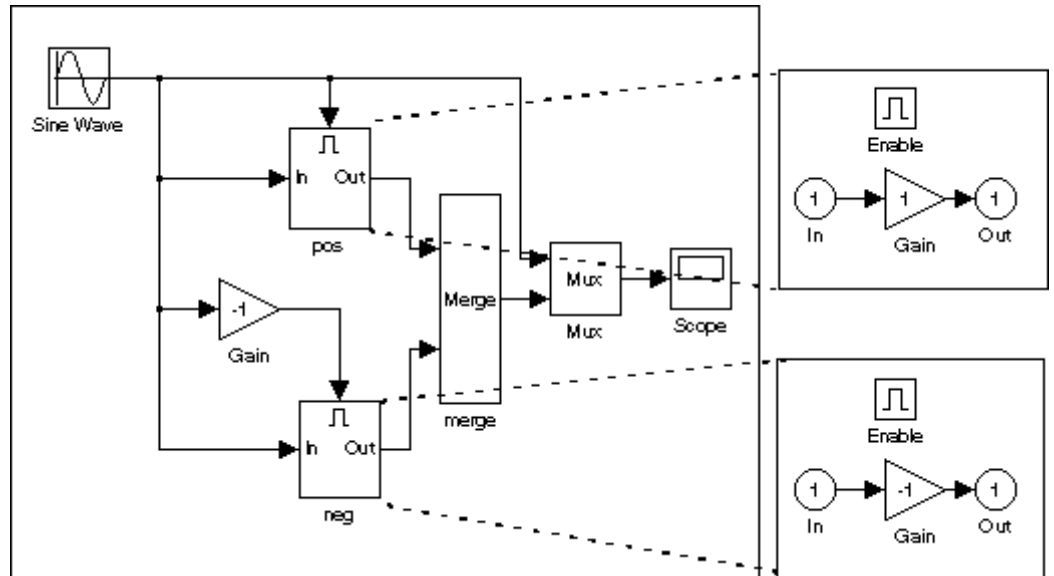
The block labeled "pos" is enabled when the AC waveform is positive; it passes the waveform unchanged to its output. The block labeled "neg" is enabled when the waveform is negative; it inverts the waveform. The Merge block passes the output of the currently enabled block to the Mux block, which passes the output, along with the original waveform, to the Scope block.

The Scope creates the following display.
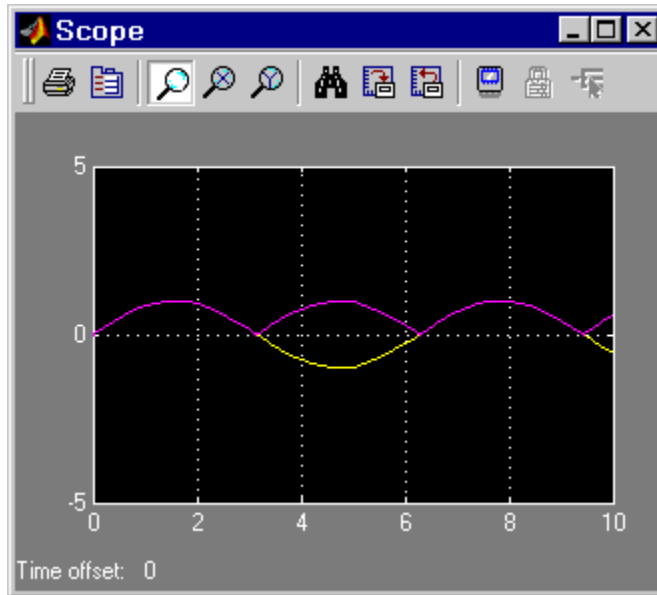
Function-Call Subsystems

A function-call subsystem is a subsystem that another block can invoke directly during a simulation. It is analogous to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods (see "Block Methods" on page 2-12) of the blocks that the subsystem contains in sorted order (see "How Simulink® Determines the Sorted Order" on page 6-36). The block that invokes a function-call subsystem is called the function-call initiator. Stateflow®, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

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

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

A function-call initiator can invoke a periodic function-call subsystem only once per time step and must invoke the subsystem periodically. If the initiator invokes a periodic function-call subsystem aperiodically, the simulation is halted and an error message displayed. The blocks in a periodic function-call subsystem can specify a noninherited sample time or inherited (-1) sample time. All blocks that specify a noninherited sample time must specify the sample time, 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, the simulation is halted and an error message is displayed.

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

Conditional Execution Behavior

In this section...
"Propagating Execution Contexts" on page 4-24
"Behavior for Switch Blocks" on page 4-25
"Displaying Execution Contexts" on page 4-25
"Disabling Conditional Execution Behavior" on page 4-26
"Displaying Execution Context Bars" on page 4-26

To speed simulation of a model, by default the Simulink® software avoids unnecessary execution of blocks connected to Switch, Multiport Switch, and of conditionally executed blocks, a behavior called *conditional execution (CE)* behavior. You can disable this behavior for all Switch and Multiport Switch blocks in a model, or for specific conditional subsystems. See "Disabling Conditional Execution Behavior" on page 4-26.

The following model illustrates conditional execution behavior.

Gain block's sorted order (1:2) is second (2) in the enabled subsystem's execution context (1).
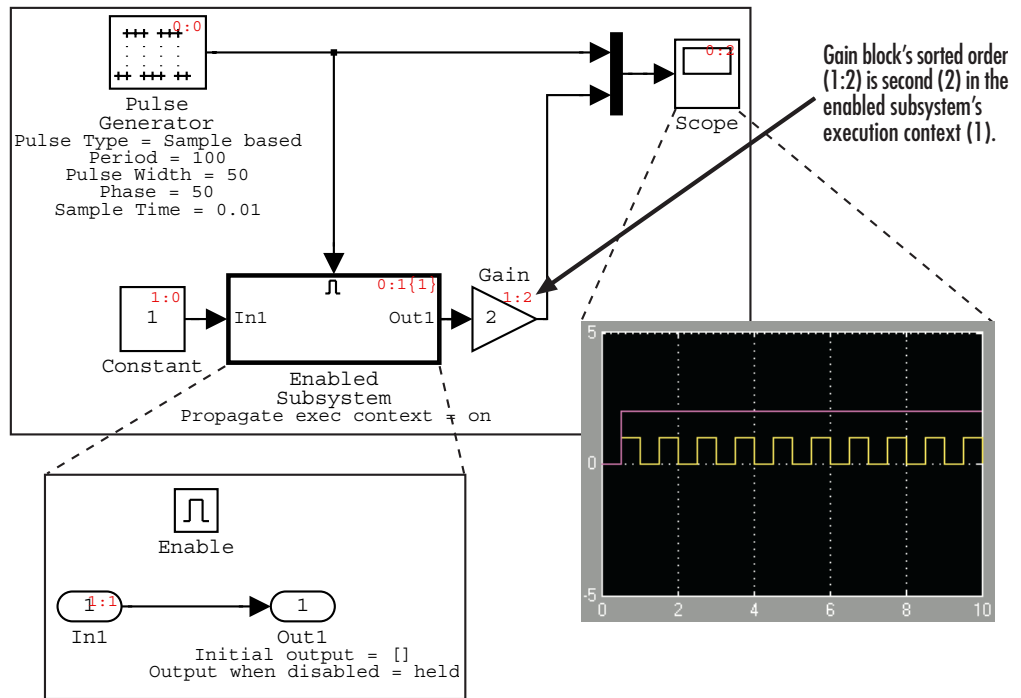
The outputs of the Constant block and Gain blocks are computed only while the enabled subsystem is enabled (that is, at time steps 0.5 to 1.0, 1.5 to 2.0, and so on). This is because the output of the Constant block is required and the input of the Gain block changes only while the enabled subsystem is enabled. When CE behavior is off, the outputs of the Constant and Gain blocks are computed at every time step, regardless of whether the outputs are needed or change.

In this example, the enabled subsystem is regarded as defining an execution context for the Constant and Gain blocks. Although the blocks reside graphically in the model's root system, the Simulink software invokes the blocks' methods during simulation as if the blocks reside in the enabled subsystem. This is indicated in the sorted order labels displayed on the diagram for the Constant and Gain blocks. The notations list the subsystem's (id = 1) as the execution context for the blocks even though the blocks exist graphically at the model's root level (id = 0).

Propagating Execution Contexts

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

When compiling, each block in the model is examined to determine whether it meets the following conditions:

- Its output is required only by a conditional subsystem or its input changes only as a result of the execution of a conditionally executed.

- The subsystem's execution context can propagate across its boundaries.

- The output of the block is not a testpoint (see "Working with Test Points" on page 8-70).

- The block is allowed to inherit its conditional execution context.

  The Simulink software does not allow some built-in blocks, e.g., the Delay block, ever to inherit their execution context. Also, S-Function blocks can inherit their execution context only if they specify the SS_OPTION_CAN_BE_CALLED_CONDITIONALLY option.

- The block is not a multirate block.

- Its sample time is inherited (-1).

If a block meets these conditions and execution context propagation is enabled for the associated conditional subsystem (see "Disabling Conditional Execution Behavior" on page 4-26), the Simulink software moves the block into the execution context of the subsystem. This ensures that the block's methods are executed during the simulation loop only when the corresponding conditional subsystem executes.

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

Behavior for Switch Blocks

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

Displaying Execution Contexts

To determine the execution context to which a block belongs, select **Sorted order** from the model window's **Format** menu. The sorted order index for each block in the model is displayed in the upper-right corner of the block. The index has the format *s:b*, where *s* specifies the subsystem to whose execution context the block belongs and *b* is an index that indicates the block's sorted order in the subsystem's execution context, e.g., 0:0 indicates that the block is the first block in the root subsystem's execution context.

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

The sorted order index of conditional subsystems is expanded to include the system ID of the subsystem itself in curly brackets as illustrated in the following figure.
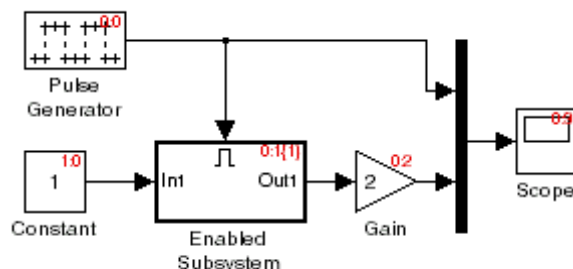
In this example, the sorted order index of the enabled subsystem is 0:1{1}. The 0 indicates that the enabled subsystem resides in the model's root system. The first 1 indicates that the enabled subsystem is the second block

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

Disabling Conditional Execution Behavior

To disable conditional execution behavior for all Switch and Multiport Switch blocks in a model, turn off the `Conditional input branch execution` optimization on the **Optimization** pane of the Configuration Parameters dialog box (see "Optimization Pane"). To disable conditional execution behavior for a specific conditional subsystem, 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

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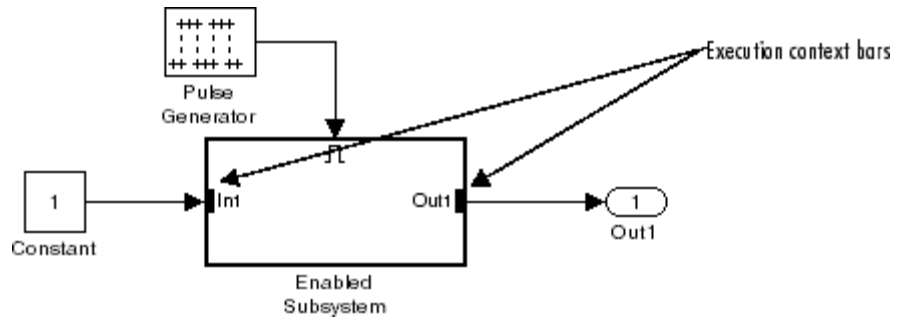
To display the bars, select **Execution Context Indicator** from model editor's **Format > Block Displays** menu.

5

Referencing a Model

Overview of Model Referencing

In this section...
"About Model Referencing" on page 5-3
"Referenced Model Advantages" on page 5-5
"Model Referencing Demos" on page 5-6
"Model Referencing Resources" on page 5-7

About Model Referencing

You can include one model in another by using Model blocks. Each instance of a Model block represents a reference to another model, called a *referenced model* or *submodel*. For simulation and code generation, the referenced model effectively replaces the Model block that references it. The model that contains a referenced model is its *parent model*.

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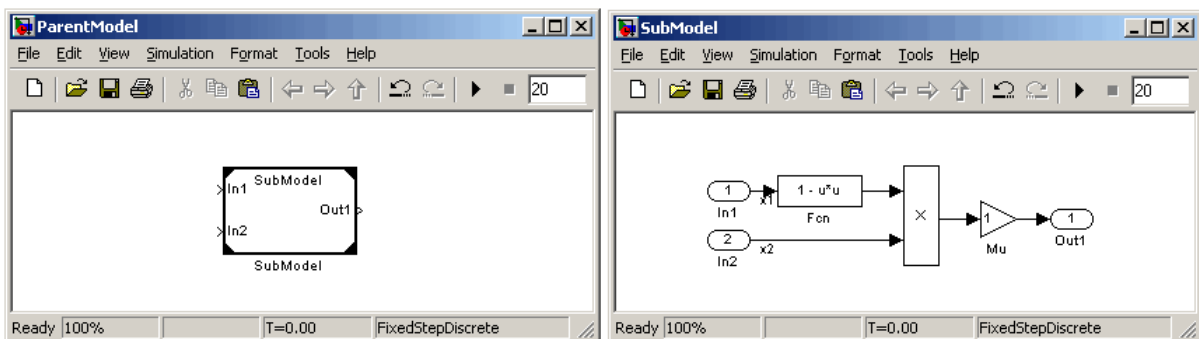
You can use the ports on a Model block to connect the submodel to other elements of the parent model. Connecting a signal to a Model block port

has the same effect as connecting the signal to the corresponding port in the submodel.
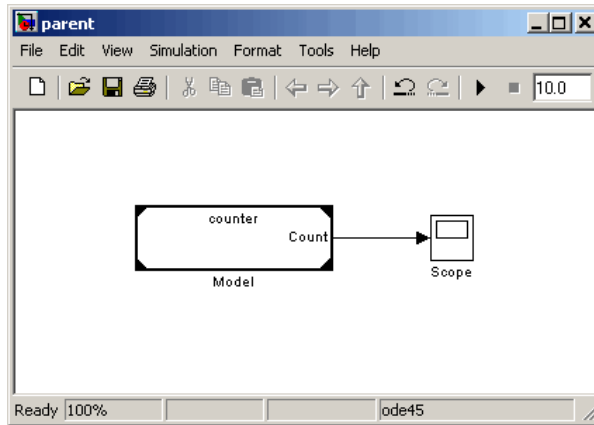
A referenced model can itself contain Model blocks and thus reference lower-level models, and so on to any depth. The topmost model in a hierarchy of referenced models is called the *top model*. Where only one level exists, the parent model and top model are the same. To prevent cyclic inheritance, a Model block cannot refer directly or indirectly to a model that is superior to it in the model reference hierarchy, as shown in this figure:
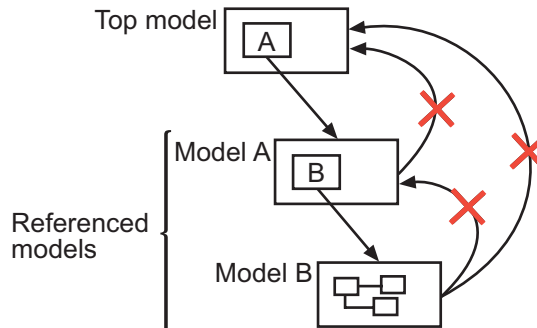
A parent model can contain multiple Model blocks that reference the same submodel as long as the submodel does not define global data. The submodel can also appear in other parent models at any level. You can parameterize a referenced model to provide tunability for all instances of the model, or let

different Model blocks specify different values for variables that define the submodel's behavior. See "Parameterizing Model References" on page 5-26 for details.

By default, the Simulink® software executes a top model interpretively, just as it would if the model did not include submodels. Simulink can execute a referenced model interpretively, as if it were an atomic subsystem, or by compiling the submodel to code and executing the code. See "Referenced Model Simulation Modes" on page 5-13 for details.

You can use any referenced model as a standalone model, provided that it does not depend on any data that is available only from a higher-level model. An appropriately configured model can function as both a standalone model and as a referenced model without requiring any change to the model itself or to any entities derived from it.

Referenced Model Advantages

Like subsystems, referenced models allow you to organize large models hierarchically; Model blocks can represent major subsystems. Like libraries, referenced models allow you to use the same capability repeatedly without having to redefine it. However, referenced models provide several advantages that are unavailable with subsystems and/or library blocks:

- **Modular development**

  You can develop a referenced model independently from the models in which it is used.

- **Inclusion by reference**

  You can reference a model multiple times without having to make redundant copies, and multiple models can reference the same model.

- **Incremental loading**

  A referenced model is not loaded until it is needed, which speeds up model loading.

- **Accelerated simulation**

  Simulink can convert a referenced model to code and simulate the model by running the code, which is faster than interactive simulation.

- **Incremental code generation**

  Accelerated simulation requires code generation only if the model has changed since code was previously generated. Otherwise the existing code can be reused.

- **Independent configuration sets**

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

Model Referencing Demos

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

1 In the MATLAB Command Window, type

    demos

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

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

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

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

This category contains model referencing demos, including:

- Component-Based Modeling with Model Reference — `sldemo_mdlref_basic`

- Visualizing Model Reference Architectures — `sldemo_mdlref_depgraph`

- Interface Specification Using Bus Objects — `sldemo_mdlref_bus`

- Parameterizing Model Reference — `sldemo_mdlref_paramargs`

- Converting Subsystems to Model Reference —
  `sldemo_mdlref_conversion`

- Model Reference Function-Call — `sldemo_mdlref_fcncall`

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

Model Referencing Resources

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

- The Model block, which represents a model that is included as a referenced model in another model.

- The **Configuration Parameters > Diagnostics > Model Referencing** pane, which controls the diagnosis of problems encountered in model referencing. See "Diagnostics Pane: Model Referencing" for details.

- The **Configuration Parameters > Model Referencing** pane, which provides options that control model referencing and list files on which referenced models depend. See "Model Referencing Pane" for details.

Creating a Model Reference

A model becomes a submodel when a Model block in some other model references it. Any model can function as a submodel, and such use does not preclude using it as a separate model also. To create a reference to a model (submodel) in another model (parent model):

1 If the directory containing the submodel to be referenced is not on the MATLAB® path, add the directory to the MATLAB path.

2 In the submodel:

- Enable **Configuration Parameters > Optimization > Inline parameters**. You must enable **Inline parameters** for all models in a model reference hierarchy except the hierarchy's top model. See "Inline Parameter Requirements" on page 5-23 for details.

- Set **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** to `One` if the model will be used at most once in any hierarchy, or to `Multiple` if it will be used more than once. To reduce overhead, specify `Multiple` only when necessary. You can also set the option to `Zero`, which precludes referencing the model.

3 Create an instance of the Model block in the parent model by dragging a Model block instance from the Ports & Subsystems library to the parent model. The new block is initially unresolved (specifies no submodel) and has the following appearance:
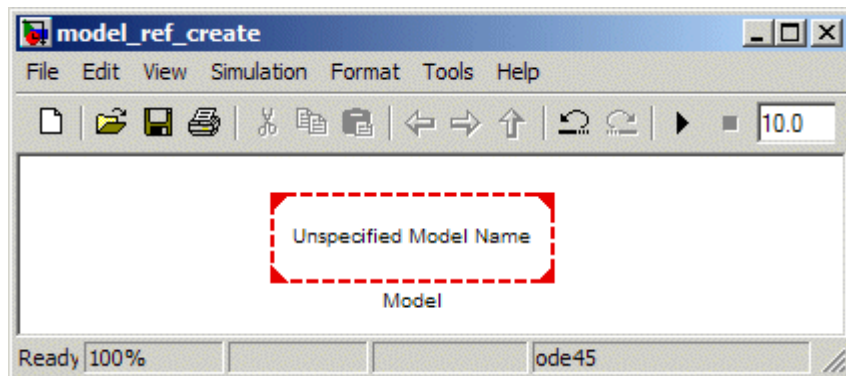
4 Open the new Model block's parameter dialog box by double-clicking the Model block. See "Navigating a Model Block" for more about accessing Model block parameters.
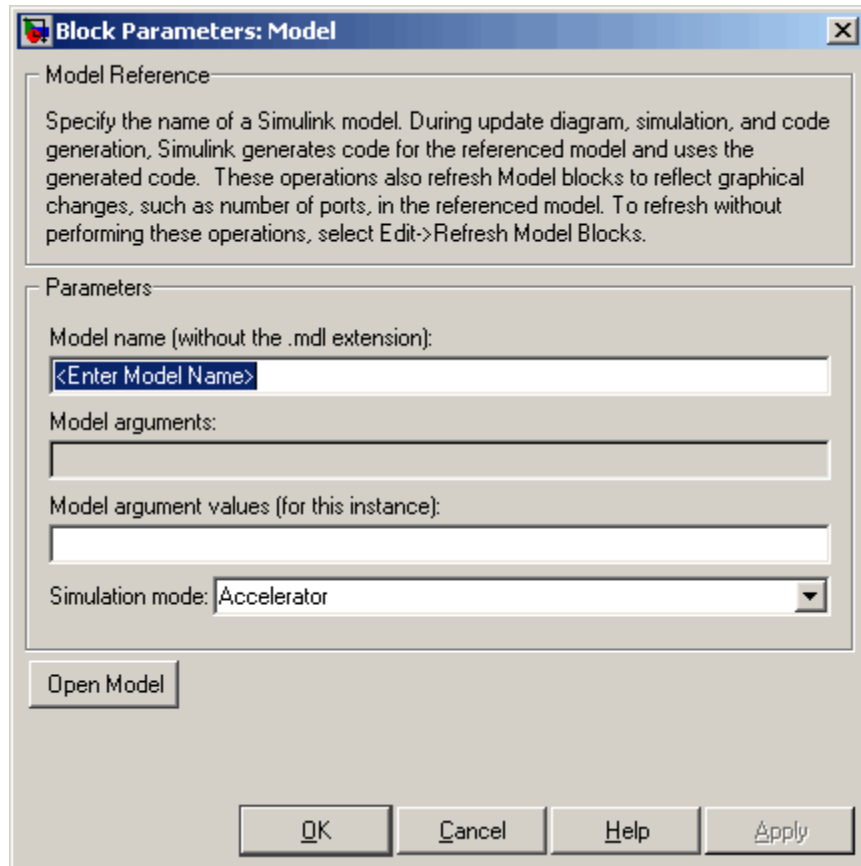
5 Enter the name of the submodel in the **Model name** field. This name must contain fewer than 60 characters. (See "Name Length Requirement" on page 5-19.)

- For information about **Model Arguments** and **Model argument values**, see "Using Model Arguments" on page 5-28.

- For information about the **Simulation mode**, see "Referenced Model Simulation Modes" on page 5-13.

6 Click **OK** or **Apply**.

If the referenced model contains any root-level inputs or outputs, Simulink®
displays corresponding input and output ports on the Model block instance
that you have created. Use these ports to connect the referenced model to
other ports in the parent model. For information about connecting blocks in a
parent model to a referenced model that has bus inputs or outputs, see "Bus
Usage Requirements" on page 5-25.

Converting a Subsystem to a Referenced Model

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

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

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

- Saves the contents of the subsystem as a new model. Simulink automatically provides a model name that is based on the block name and is unique in the MATLAB® path. This name always contains fewer than 60 characters.
- Creates and opens an untitled model that contains a Model block whose referenced model is the new model that contains the contents of the subsystem.

If an error occurs during the conversion, the result depends on the error:

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

Once you have successfully created a Model block and referenced model from a subsystem, you can delete the subsystem block from the source model and copy the Model block to the subsystem block's location. All signals will automatically reconnect. The source model is now a parent model that contains a referenced model.

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

Referenced Model Simulation Modes

In this section...
"About Referenced Model Simulation Modes" on page 5-13
"Specifying the Simulation Mode" on page 5-14
"Mixing Simulation Modes" on page 5-14
"Accelerating a Freestanding or Top Model" on page 5-15

About Referenced Model Simulation Modes

Simulink® executes the top model in a model reference hierarchy just as it would if no referenced models existed. All Simulink simulation modes are available to the top model. Simulink can execute a referenced model in either of two modes:

- **Normal mode** — Simulink executes the submodel interpretively, as if the submodel were an atomic subsystem implemented directly within the parent model. Normal mode is slower than Accelerator mode, and works with only *one* instance of a given model in a reference hierarchy, but it requires no delay for code generation and works with most Simulink tools.

- **Accelerator mode** — Simulink creates a MEX-file for the submodel, then executes the submodel by running the MEX-file. Accelerator mode is faster than Normal mode, and works with *multiple* submodel instances, but it requires compilation and does not work with most Simulink tools.

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

Note Do not confuse Accelerator mode execution of a referenced model with:

- Accelerator mode execution of a freestanding or top model, as described in Chapter 19, "Accelerating Models"

- Rapid Accelerator mode execution of a freestanding or top model, as described in "Running Rapid Simulations".

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

Specifying the Simulation Mode

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

1 Access the Model block's parameter dialog box. (See "Navigating a Model Block".)

2 Set the **Simulation mode** field to `Normal` or `Accelerator`.

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

Mixing Simulation Modes

Simulink models execute in Normal mode by default. When a top model executes in Normal mode, it can contain both Normal mode and Accelerator mode submodels. When the same submodel appears more than once in a hierarchy, at most one of these instances can specify Normal mode. All the rest must specify Accelerator mode.

Accelerator mode takes precedence over Normal mode when the two are mixed. Thus a model that executes in Normal mode can include submodels that execute in Accelerator mode, but a model that executes in Accelerator mode cannot include any submodels that execute in Normal mode. When a Normal mode submodel is subordinate to an Accelerated mode submodel, Simulink posts a warning and temporarily overrides the Normal mode specification.

Accelerating a Freestanding or Top Model

You can use Simulink Accelerator mode (see Chapter 19, "Accelerating Models") or Rapid Accelerator mode (see "Running Rapid Simulations") to achieve faster execution of any Simulink model, including a top model in a model reference hierarchy.

When you execute a top model in Simulink Accelerator mode or Rapid Accelerator mode, all submodels execute in Accelerator mode. For any submodel that specifies Normal mode, Simulink posts a warning and temporarily overrides the Normal mode specification.

Simulation Targets

In this section...
"About Simulation Targets" on page 5-16
"Building Simulation Targets" on page 5-17

About Simulation Targets

A *simulation target*, or *SIM target*, is a MEX-file that implements a referenced model that executes in Accelerator mode. Simulink® invokes the simulation target as needed during simulation to compute the behavior and outputs of the referenced model. Simulink uses the same simulation target for all Accelerator mode instances of a given referenced model anywhere in a reference hierarchy.

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

- Hardware target — A platform for which Real-Time Workshop® generates code

- System target — A file that tells Real-Time Workshop how to generate code for particular purpose

- Rapid Simulation target (RSim) — A system target file supplied with Real-Time Workshop

- Model reference target — A library module that contains Real-Time Workshop code for a referenced model

Simulink creates a simulation target only for a submodel that has one or more Accelerator mode instances in a reference hierarchy. A submodel that executes only in Normal mode always executes interpretively and does not use a simulation target. When one instance of a submodel executes in Normal mode, and one or more instances execute in Accelerator mode, Simulink creates a simulation target for the Accelerator mode instance(s), but the Normal mode instance does not use it.

Because Accelerator mode requires code generation, it imposes some requirements and limitations that do not apply to Normal mode. Aside

from these constraints, you can generally ignore simulation targets and their details when you execute a referenced model in Accelerator mode. See "Limitations on Accelerator Mode Referenced Models" on page 5-47 for details.

Building Simulation Targets

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

You can change this default behavior to change the rebuild criteria or specify that Simulink always or never rebuilds targets. See "Rebuild options" for details. You can command Simulink to generate simulation targets for Accelerator mode referenced models at any time by updating the model's diagram or by executing the slbuild command with appropriate arguments at the MATLAB® command line.

While generating a simulation target, Simulink displays status messages at the MATLAB command line to enable you to monitor the target generation process, 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 a subdirectory of the working directory. This subdirectory is named slprj. If slprj does not exist, Simulink creates it. Subdirectories in slprj provide separate places for simulation code, Real-Time Workshop code, and other files.

Reducing Change Checking Time

You can reduce the time that Simulink spends checking whether any or all simulation targets need to be rebuilt by setting configuration parameter values as follows:

- In all referenced models throughout the hierarchy, set **Configuration Parameters > Diagnostics > Data Validity > Signal resolution** to Explicit only. (See "Signal resolution".)

- In any referenced model for which you want to minimize change checking time, set **Configuration Parameters > Model Referencing > Rebuild options** to `If any changes in known dependencies detected`. (See "Rebuild options".)

These parameter values exist in a referenced model's configuration set, not in the individual Model block, so setting either value for any instance of a referenced model sets it for all instances of that model.

Simulink® Model Referencing Requirements

About Model Referencing Requirements

A model reference hierarchy must satisfy various Simulink® requirements, as described in this section. Some limitations also apply, as described in "Simulink® Model Referencing Limitations" on page 5-43.

Name Length Requirement

The name of a referenced model must contain fewer than 60 characters, exclusive of the .mdl suffix. An error occurs if the name of a referenced model is too long.

Configuration Parameter Requirements

A referenced model uses a configuration set in the same way that any other model does, as described in "Configuration Sets" on page 14-37. By default, every model in a hierarchy has its own configuration set, which it uses in the same way that it would if the model executed independently.

Because each model can have its own configuration set, configuration parameter values can be different in different models. Furthermore, some parameter values are intrinsically incompatible with model referencing. Simulink's response to an inconsistent or unusable configuration parameter depends on the parameter:

- Where an inconsistency has no significance, or a trivial resolution exists that carries no risk, Simulink ignores or resolves the inconsistency without posting a warning.

- Where a nontrivial and possibly acceptable solution exists, Simulink resolves the conflict silently; resolves it with a warning; or generates an error. See "Model configuration mismatch" for details.

- Where no acceptable resolution is possible, Simulink generates an error. You must then change some or all parameter values to eliminate the problem.

When a model reference hierarchy contains many submodels that have incompatible parameter values, or a changed parameter value must propagate to many submodels, manually eliminating all configuration parameter incompatibilities can be tedious. You can control or eliminate such overhead by using configuration references to assign an externally-stored configuration set to multiple models. See "Referencing Configuration Sets" on page 14-47 for details.

Note Configuration parameters on the Real-Time Workshop® pane of the Configuration Parameters dialog have no effect on simulation in either Normal or Accelerated mode. Real-Time Workshop parameters affect only code generation by Real-Time Workshop itself. Although Accelerated mode simulation requires code generation to create a simulation target, Simulink uses default values for all Real-Time Workshop parameters when generating the target, and restores the original parameter values after code generation is complete.

The tables in the following sections list Configuration parameter options that can cause problems if set in certain ways, or if set differently in a referenced model than in a parent model. Where possible, Simulink resolves violations of these requirements automatically, but most cases require changes to the parameters in some or all models.

Configuration Requirements for All Referenced Model Simulation

Dialog Box Pane	Option	Requirement
Solver	Start time	The start time of the top model and all referenced models must be the same, but need not be zero.
	Stop time	Simulink uses the top model's **Stop time** for simulation, overriding any differing **Stop time** in a submodel.
	Type Solver	The top model's **Type** and **Solver** apply throughout the hierarchy. See "Solver Requirements" on page 5-22.
Data Import/Export	Initial state	Can be on for the top model, but must be off for a referenced model.
Optimization	Inline parameters	Can be on or off for a top model, but must be on for a referenced model. See "Inline Parameter Requirements" on page 5-23.
	Application lifespan (days)	Must be the same for top and referenced models.

Dialog Box Pane	Option	Requirement
Model Referencing	**Total number of instances allowed per top model**	Must not be Zero in a referenced model. Specifying One rather than Multiple is preferable or required in some cases. See "Model Instance Requirements" on page 5-24.
Hardware Implementation	**Embedded hardware** options	All values must be the same for top and referenced models.

Solver Requirements. Model referencing works with both fixed-step and variable-step solvers. All models in a model reference hierarchy use the same solver, which is always the solver specified by the top model. An error occurs if the solver type specified by the top model is incompatible with the solver type specified by any submodel, as shown in the following table:

Top Model Solver Type	Submodel Solver Type	Compatibility
Fixed Step	Fixed Step	Allowed
Variable Step	Variable Step	Allowed
Variable Step	Fixed-step	Allowed unless the submodel is multi-rate and specifies both a discrete sample time and a continuous sample time
Fixed Step	Variable Step	Error

If an incompatibility exists between the top model solver and any submodel solver, one or both models must change as needed to use compatible solvers. For information about solvers, see "Solvers" on page 2-18 and "Choosing a Solver" on page 14-11.

Inline Parameter Requirements. Simulink requires **Configuration Parameters > Optimization > Inline parameters** (see "Inline parameters") to be enabled for all referenced models in a reference hierarchy. The top model can enable or disable inline parameters. If a referenced model disables inlined parameters, and you try to build the parent model:

- For a Normal mode submodel, Simulink generates an error and cancels the build. All models and compiled files remain unchanged after the failed build.

- For an Accelerator mode submodel, Simulink temporarily enables inline parameters, posts no warning, and builds the model. Inline parameters remain disabled after the build completes.

Simulink ignores tunable parameter specifications in the **"Model Parameter Configuration Dialog Box"** for both the top model and referenced models. Consequently, you cannot use this dialog box to override the inline parameters optimization for selected parameters and thereby permit them to be tuned. "Parameterizing Model References" on page 5-26 describes alternate techniques.

Model Instance Requirements. A referenced model must specify that it is available for such use, and whether it can be used at most once or can have multiple instances. **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** provides this specification. See "Total number of instances allowed per top model" for more information. The possible values for this parameter are:

- Zero — The model cannot be referenced. An error occurs if a reference to the model occurs in another model.

- One — The model can be referenced at most once in a model reference hierarchy. An error occurs if more than one instance exists. This value may be preferable or required.

- Multiple — The model can be referenced more than once in a hierarchy, provided that it contains no constructs that preclude multiple reference. An error occurs if the model cannot be multiply referenced, even if only one reference exists.

Setting **Total number of instances allowed per top model** to Multiple for a model that is referenced only once can reduce execution efficiency slightly, but does not affect data values that result from simulation or from executing code generated by Real-Time Workshop. Specifying Multiple when only one model instance exists facilitates later reusing the model in the same hierarchy, or multiple times in a different hierarchy, without having to change or rebuild the model.

Some model properties and constructs require **Total number of instances allowed per top model** to be set to One, limiting the model to being used only once in a hierarchy. For details, see "General Reusability Limitations" on page 5-44 and "Accelerator Mode Reusability Limitations" on page 5-48.

Model Structure Requirements

The following requirements relate to the structure of a model reference hierarchy independently of configuration parameter requirements.

Signal Propagation Requirements

The signal name must explicitly appear on any signal line connected to an Outport of a referenced model. A signal that is connected by an unlabeled line to an Outport of a referenced model cannot propagate out of the Model block to the parent model.

Bus Usage Requirements

A bus that propagates between a parent model and a referenced model must be nonvirtual, and the same bus object must specify the properties of the bus in both the parent and the referenced model. This object must be defined in the MATLAB® workspace. See "Using Buses" on page 9-5 for more information.

Sample Time Requirements

The first nonvirtual block connected to a root-level Inport or Outport of a referenced model must have the same sample time as the port to which it connects. You can use Rate Transition blocks to match input and output sample times as illustrated in the following diagram.
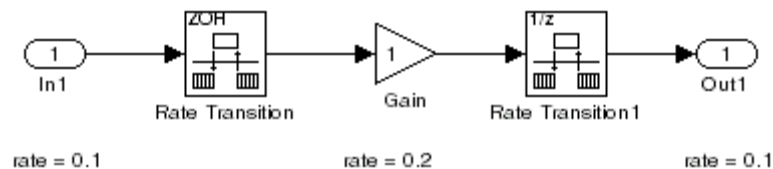
Parameterizing Model References

In this section...

Why Parameterize Model References?

Due to the constraints described in "Inline Parameter Requirements" on page 5-23, you cannot use the "Model Parameter Configuration Dialog Box" to tune parameters in referenced models.

Simulink® provides three other techniques that you can use to parameterize referenced models:

- Global Nontunable Parameters
- Global Tunable Parameters
- Model Arguments

You cannot parameterize a referenced model by using symbols that match definitions on the workspace of

Global Nontunable Parameters

A *global nontunable parameter* is a MATLAB® variable or a Simulink.Parameter object whose storage class is auto. The parameter can exist on the MATLAB workspace or any model workspace visible to all referenced models that use the parameter.

Using a global nontunable parameter in a referenced model allows you to control the behavior of the referenced model by setting the parameter value before simulation begins. All instances of the model use the same value. You cannot change the value during simulation, but you can change it between one simulation and the next. The change requires rebuilding the model in which

the change occurs, but not any models that it references. See "Specifying Numeric Parameter Values" on page 6-10 for details.

Global Tunable Parameters

A *global tunable parameter* is a `Simulink.Parameter` object whose storage class is other than `auto`. The parameter exists on the MATLAB workspace.

Using a global tunable parameter in a referenced model allows you to control the behavior of the referenced model by setting the parameter value. All instances of the model use the same value. You can change the value during simulation or between one simulation and the next. The change does not requires rebuilding the model in which the change occurs, or any models that it references. See "Changing the Values of Block Parameters During Simulation" on page 6-16 for details.

If you want to reference an existing model that uses tunable parameters defined with the "Model Parameter Configuration Dialog Box", you must change the model to implement tunability in some other way. To facilitate this task, Simulink provides a command that converts tunable parameters specified in the Model Parameter Configuration dialog box to global tunable parameters. See `tunablevars2parameterobjects` for details.

Model Arguments

You can also use model arguments to specify different behavior for different references to the same model. This is the only technique that lets you specify different behaviors for different instances of the same model. See "Using Model Arguments" on page 5-28 for details.

Using Model Arguments

About Model Arguments

Model arguments let you parameterize references to the same model so that each instance of the model behaves differently. Without model arguments, a variable in a referenced model has the same value in every instance of the model. Declaring a variable to be a model argument allows each instance of the model to use a different value for that variable.

To create model arguments for a referenced model, you create MATLAB® variables in the model workspace, then add the variables to a list of model arguments associated with the model. You can then specify values for those variables separately in each Model block that references the model. The values specified in the Model block replace the values of the MATLAB variables for that instance of the model.

A referenced model that uses model arguments might also appear as a top model or a standalone model. No Model block then exists to provide model argument values, and the model uses the values of the MATLAB variables themselves, as defined in the model workspace. Thus the same model can be used without change as a top model, a standalone model, and a parameterized referenced model.

The demo model sldemo_mdlref_paramargs demonstrates techniques for using model arguments. The demo passes model argument values to referenced models through masked Model blocks. Such masking can be convenient, but is independent of the definition and use of model arguments themselves. See Chapter 17, "Creating Block Masks" for details about masking.

The rest of this section describes techniques for declaring and using model arguments to parameterize a referenced model independently of any Model block masking. The steps are:

- Create MATLAB variables in the model workspace.

- Register the variables to be model arguments.

- Assign values to those arguments in Model blocks.

Creating the MATLAB® Variables

To create MATLAB variables that will be used as model arguments:

1 Open the model for which you want to define model arguments.

2 Open the Model Explorer.

3 Select the model's workspace in Model Explorer's **Model Hierarchy** pane:
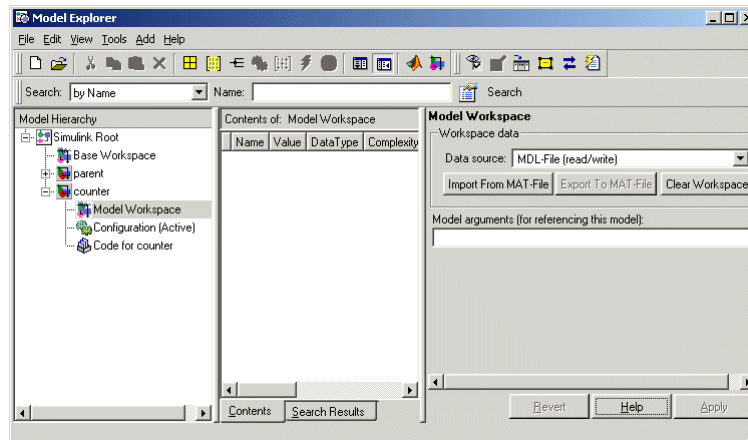
4 From Model Explorer's **Add** menu, select **MATLAB Variable**.

A new MATLAB variable appears in the **Contents** pane with a default name and value.

5 In the **Contents** pane:

a Change the default name of the new MATLAB variable to a name that you want to declare as a model argument.

b If you will also use the model as a top or standalone model, specify the value that the variable should have in that context. This value must be numeric.

c If the variable type does not match the dimensions and complexity of the model argument, specify a value that has the correct type. This type must be numeric.

6 Repeat adding and naming MATLAB variables until you have defined all the variables that you need.

Registering the Model Arguments

To register MATLAB variables as model arguments:

1 Again select the model's workspace in Model Explorer's **Model Hierarchy** pane.

The Dialog pane displays the Model Workspace dialog.

2 In the Model Workspace dialog, enter the names of the MATLAB variables that you want to declare as model arguments as a comma-separated list in the **Model arguments** field.

For example, if you added two MATLAB variables named init_value and incr, and declared them to be model arguments, the Contents and Dialog panes of the Model Explorer could look like this:
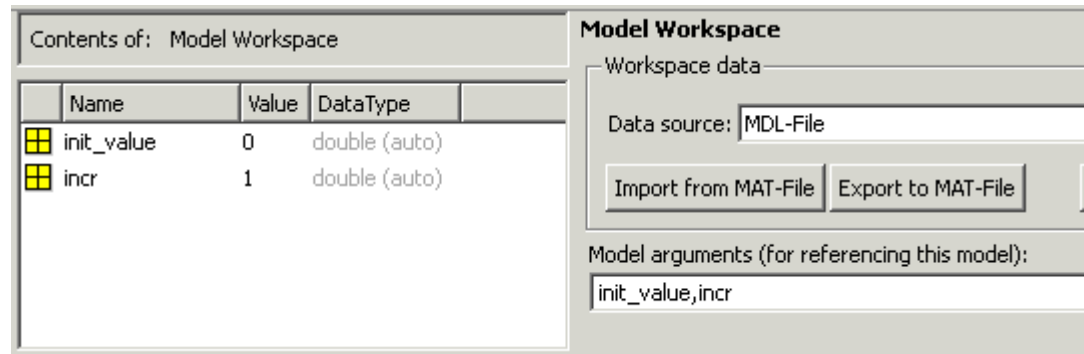
3 Click **Apply** to confirm the entered names.

Assigning Model Argument Values

If a model declares model arguments, you must assign values to those arguments in each Model block that references the model. Failing to assign a value to a model argument causes an error: the value of the model argument does *not* default to the value of the corresponding MATLAB variable. That value is available only to a standalone or top model. To assign values to a referenced model's arguments:

1 Open the Model block's parameter dialog box by right-clicking the block and choosing **Model Reference Parameters** from the context menu.
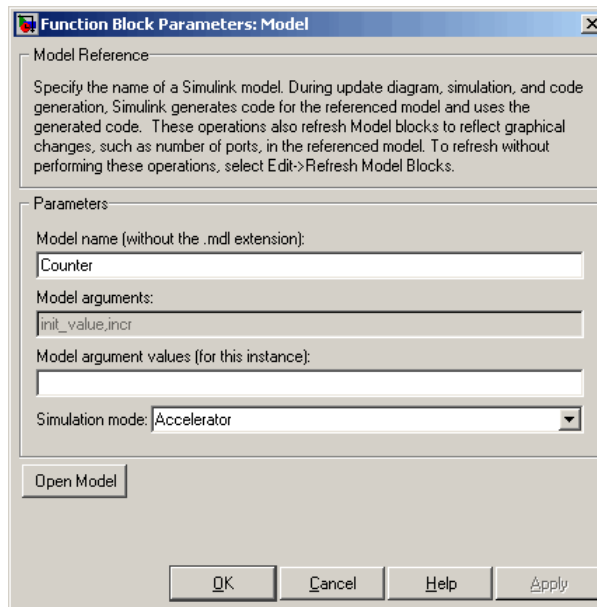
The second field, **Model arguments**, specifies the same MATLAB variables, in the same order, that you previously typed into the **Model arguments** field of the Model Workspace dialog. This field cannot be edited. It provides a reminder of which model arguments need values assigned, and in what order.

2 In the **Model argument values** field, enter a comma-delimited list of values for the model arguments that appear in the **Model arguments** field. The values are assigned to arguments in positional order, so they must appear in the same order as the corresponding arguments.
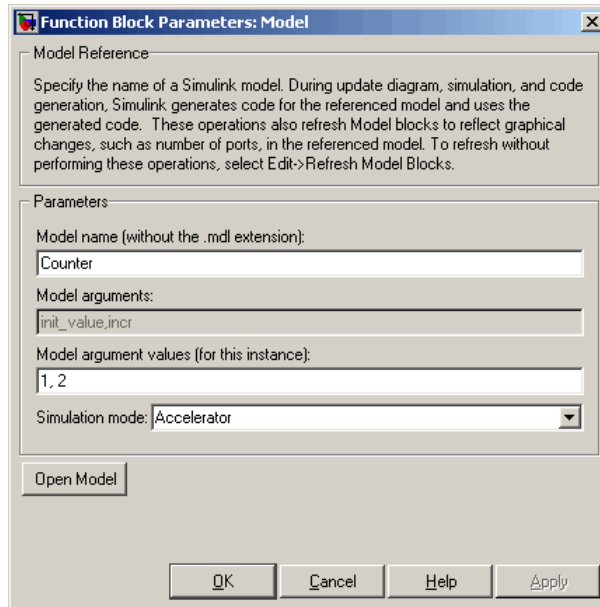
You can enter the values as literal values, variable names, MATLAB expressions, and Simulink® parameter objects. Any symbols used resolve to values as described in "Hierarchical Symbol Resolution" on page 3-70. All values must be numeric (including objects with numeric values).

The value for each argument must have the same dimensions and complexity as the MATLAB variable that defines the model argument in the model workspace. The data types need not match. If necessary, the Simulink software will cast a model argument value to the data type of the corresponding MATLAB variable.

3 Click **OK** or **Apply** to confirm the values for the Model block.

When the model executes in the context of that Model block, the **Model arguments** will have the values specified in the Model block's **Model argument values** field.

Refreshing Model Blocks

Refreshing a Model block updates its internal representation to reflect changes in the interface of the model that it references. For example, you must refresh a Model block if its referenced model has gained or lost a port. When more than one Model block references a model whose interface has changed, all of the Model blocks must be refreshed. Changes that have no effect on a referenced model's interface to its parent do not require refreshing.

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 menu.

Simulink® provides diagnostics that you can use to detect changes in the interfaces of referenced models that could require refreshing the Model blocks that reference them. The diagnostics include:

- **"Model block version mismatch"**
- **"Port and parameter mismatch"**

Examining a Model Reference Hierarchy

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

- "Using the Model Dependency Viewer" on page 13-49 — Show the structure of a model reference hierarchy and allows you to open any referenced model.

- view_mdlrefs function — Invoke the Model Dependency Viewer to display a graph of model reference dependencies.

- find_mdlrefs function — Finds all models directly or indirectly referenced by a given model.

Displaying Version Numbers

To display the version numbers of the models referenced by a model, choose **Model block version** from the **Block displays** submenu of the parent model's **Format** menu. Simulink displays the version numbers in the icons of the corresponding Model block instances.
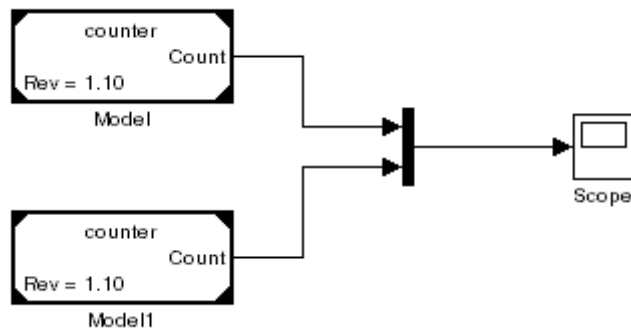
The version number displayed on a Model block's icon refers to the version of the model used to create the block, or used most recently to refresh the block. See "Managing Model Versions" on page 3-95 and "Refreshing Model Blocks" on page 5-34 for more information.

Inheriting 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 specifies the required rates. Otherwise, the referenced model inherits its sample time from the parent model.

Without the ability to inherit sample times, a Model block could not be placed in a triggered, function call, or iterator subsystem. Additionally, allowing a Model block to inherit sample time maximizes its reuse potential. For example, a model might fix the data types and dimensions of all its input and output signals, but could be reused with different sample times, for example, discrete at 0.1, discrete at 0.2, triggered, and so on.

A referenced model inherits its sample time if and only if all the following 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 and the **Fixed-step size** is auto.

- It does not contain any S-functions that make use of their specific sample time internally.

- 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, as listed in "Blocks That Preclude Sample-Time Inheritance" on page 5-37.

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

If you want a Model block to be used in a model where it can inherit a sample time, you must constrain the solver declared for that model. On the **Solver**

configuration pane, set solver **Type** to `Fixed-step` and **Periodic sample time constraint** to `Ensure sample time independent`.

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`. If the model is unable to inherit sample times, this setting causes Simulink® to display an error message when building the model. See "Periodic sample time constraint" for more about this option.

To determine the intrinsic sample time of a referenced model (or the fastest intrinsic sample time for multirate referenced models), first update some 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 simulation to produce unexpected or erroneous results. For this reason, when building a submodel that does not need to run at a specified rate, Simulink checks whether the model contains any blocks, including any S-Function blocks, whose outputs are functions of the inherited simulation time. If so, Simulink specifies a default sample time and displays an error if you have set the **Periodic sample time constraint** on the **Solver** configuration parameters dialog pane to `Ensure sample time independent`. See "Periodic sample time constraint" for more about this option.

The outputs of the following built-in blocks depend on their 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.

To avoid simulation errors with referenced models that inherit their sample time, you must not 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" for details.

Defining Function-Call Models

In this section...
"About Function-Call Models" on page 5-39
"Function-Call Model Demo" on page 5-39
"Creating a Function-Call Model" on page 5-39
"Referencing a Function-Call Model" on page 5-40
"Function-Call Model Requirements" on page 5-41

About Function-Call Models

Simulink® allows certain blocks, such as a Function-Call Generator or an appropriately configured custom S-function, to control execution of a referenced model during a time step, using a function-call signal. See "Function-Call Subsystems" on page 4-21 for more information. A referenced model capable of being invoked in this way is called a *function-call model*.

Function-Call Model Demo

To view a function-call model demo, select **Simulink > Modeling Features > Model Reference > Model Reference Function-Call** from the **Demos** pane of the MATLAB® Help Browser or execute sldemo_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.
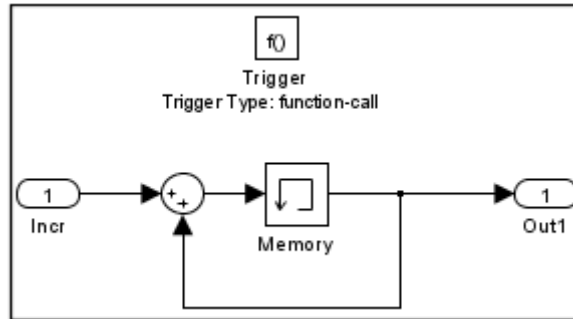
4 Ensure that the model satisfies the conditions imposed on function-call models. See "Function-Call Model Requirements" on page 5-41 for details.

You can now simulate the function-call model either by itself or by running a model that references the function-call model directly 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 5-8 for details.

The top of the Model block displays a function-call port corresponding to the function-call trigger port in the function-call model.
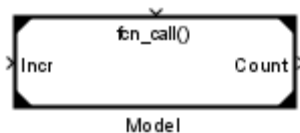
2 Connect a Stateflow® chart, Function-Call Generator block, or other function-call-generating block to the Model block's function-call port. The signal connected to the port must be scalar.

3 Connect the Model blocks inputs and outputs if any to the appropriate blocks in the parent model.
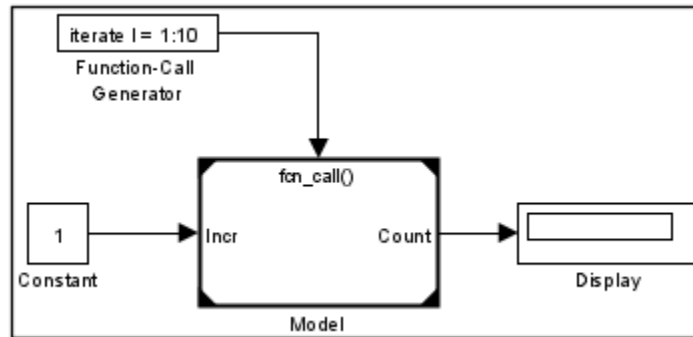
4 Create and connect any other blocks required to implement the parent model.

5 Ensure that the referencing model satisfies the conditions for a model to reference other models. See "Simulink® Model Referencing Requirements" on page 5-19 and "Simulink® Model Referencing Limitations" on page 5-43 for details.

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

Function-Call Model Requirements

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

- A function-call model cannot have an outport that is driven only by Ground blocks, including hidden Ground blocks inserted by Simulink. To meet this requirement, do the following:

  a Insert a Signal Conversion block into the signal connected to the outport.

  b Enable the inserted block's **Override optimizations and always copy signal** option.

- If the function-call model specifies a fixed-step solver and contains one or more blocks that use absolute or elapsed time, the referencing model must trigger the function-call model at the rate specified by the `'Fixed-step size'` option on the **Solver** page of the **Configuration Parameters**

dialog. Otherwise, the referencing model may trigger the function-call model at any rate.

- A function-call model must not have direct internal connections between its root-level input and output ports. Simulink does not honor the None and Warning settings for the **Invalid root Inport/Outport block connection** diagnostic for a referenced function-call model. It reports all invalid root port connections as errors.

- If the **Sample time type** is periodic, the sample-time period must not contain an offset.

- The signal connected to a Model block's function-call port must be scalar.

Simulink® Model Referencing Limitations

Requirements

The following Simulink® limitations apply to model referencing. In addition, a model reference hierarchy must satisfy all of the requirements listed in "Simulink® Model Referencing Requirements" on page 5-19.

Limitations on All Model Referencing

Index Base Limitations

In the following two cases, Simulink does not propagate 0-based or 1-based indexing information to referenced-model root-level ports connected to blocks that accept indexes, like the Assignment block, or produce indexes, like the For Iterator block.

- If a root-level input port of the referenced model is connected to index inputs in the model that have different 0-based or 1-based indexing settings, Simulink does not set the 0-based or 1-based indexing property of the root-level Inport.

- If a root-level output port of the referenced model is connected to index outputs in the model that have different 0-based or 1-based indexing settings, Simulink does not set the 0-based or 1-based indexing property of the root-level Outport.

In these cases, the lack of propagation can cause Simulink to fail to detect incompatible index connections.

General Reusability Limitations

If a referenced model has any of the following properties, the model must specify **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** as One. No other instances of the model can exist in the hierarchy. If the parameter is not set correctly, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model references another model which has been set to single instance

- The model contains a state or signal with non-auto storage class

- The model uses any of the following Stateflow® constructs:
  - Machine-parented data
  - Machine-parented events
  - Stateflow graphical functions

Simulink® Tool Limitations

- Working with the Simulink Debugger in a parent model, you can set breakpoints at Model block boundaries, allowing you to look at the block's input and output values, but you cannot set a breakpoint inside the submodel that the Model block references. See Chapter 18, "Simulink® Debugger" for more information.

- The Model Coverage tool, which is part of Simulink Verification and Validation, works for at most one referenced model at a time. That model must execute in Normal mode. See "Using Model Coverage" for more information.

- Simulink Design Verifier does not work with model referencing.

Stateflow® Limitations

- A model that contains a Stateflow chart cannot be referenced multiple times in the same model reference hierarchy if:
  - The Stateflow chart contains exported graphical functions.
  - The Stateflow model contains machine-parented data or events.

Other Limitations

- Referenced models cannot use asynchronous rates internally. However, a function-call model referenced in a top model can be triggered by an asynchronous source within the top model. See "Defining Function-Call Models" on page 5-39 for more information.

- Mask callbacks cannot add Model blocks or change existing Model block parameter values. Violating this requirement generates an error. See Chapter 17, "Creating Block Masks" for more information.

- A referenced model can input or output only those user-defined data types that are fixed-point or defined by `Simulink.DataType` or `Simulink.Bus` objects.

- Model blocks referencing models that contain assignment blocks that are not in an iterator subsystem cannot be placed in an iterator subsystem.

- If you want to initialize the states of a model that references other models with states, you must specify the initial states in structure format.

- The Model Browser does not display Model blocks in its tree view. Use the Model Explorer to browse a referenced model hierarchy.

- A referenced model cannot directly access the signals in a multi-rate bus. Connecting Multi-Rate Buses to Referenced Models describes a technique for overcoming this limitation.

- A continuous sample time cannot be propagated to a Model block that is sample-time independent.

- You cannot log the output of a Ground block in a referenced model even if you testpoint it.

- Goto/From blocks cannot cross model reference boundaries.

- You cannot print a referenced model from a top model.

Limitations on Normal Mode Referenced Models

Simulink® Tool Limitations

- Enabling the Simulink Profiler on a parent model does not enable profiling for referenced models. Profiling must be enabled separately for each submodel. See "Capturing Performance Data" on page 19-33.

- Model coverage cannot be specified for any referenced model if coverage is specified for the top model. Coverage can be specified for at most one referenced model at a time. That model must execute in Normal mode. See "Using Model Coverage".

Other Limitations

- When the same submodel appears more than once in a hierarchy, at most one of these instances can specify Normal mode. All the rest must specify Accelerator mode.

Limitations on Accelerator Mode Referenced Models

Customization Limitations

- Accelerator mode simulation ignores custom code settings in the **Configuration Parameter** dialog box and custom code blocks when generating the simulation target for a referenced model.

- Some restrictions exist on grouped custom storage classes in referenced models. See "Custom Storage Class Limitations" for details.

- Data type replacement is not supported for simulation target code generation for referenced models.

- Simulation targets do not include Stateflow target custom code.

Data Logging Limitations

- To Workspace blocks, Scope blocks, and all types of runtime display, such as the display of port values and signal values, have no effect when specified in referenced models executing in Accelerator mode. The result during simulation is the same as if the constructs did not exist.

- Referenced models executing in Accelerator mode cannot log data to MAT-files. If data logging is enabled for a referenced model, Simulink disables the option before code generation and re-enables it afterwards.

Accelerator Mode Reusability Limitations

If a referenced model has any of the following properties, and the model executes in Accelerator mode, the model must specify **Configuration Parameters > Model Referencing > Total number of instances allowed per top model** as One. No other instances of the model can exist in the hierarchy, in either Normal mode or Accelerator mode. If the parameter is not set correctly, or more than one instance of the model exists in the hierarchy, an error occurs. The properties are:

- The model contains a subsystem that is marked as function
- The model contains an S-function that is:
  - Inlined but has not set the option SS_OPTION_WORKS_WITH_CODE_REUSE
  - Not inlined
- The model contains a function-call subsystem that:
  - Has been forced by Simulink to be a function
  - Is called by a wide signal

S-Function Limitations

- If a referenced model contains an S-function that should be inlined using a Target Language Compiler file, the S-function must use the `ssSetOptions` macro to set the `SS_OPTION_USE_TLC_WITH_ACCELERATOR` option in its `mdlInitializeSizes` method. The simulation target will not inline the S-function unless this flag is set.

- The Real-Time Workshop® S-function target does not support model referencing.

- A referenced model cannot use noninlined S-functions in the following cases:

  - The model uses a variable-step solver.

  - The S-function was generated by Real-Time Workshop.

  - The S-function supports use of fixed-point numbers as inputs, outputs, or parameters.

  - The model is referenced more than once in the model reference hierarchy. To work around this limitation, make copies of the referenced model, assign different names to the copies, and reference a different copy at each location that needs the model.

Simulink® Tool Limitations

- Simulink tools that require access to a model's internal data or configuration (including Model Coverage, the Report Generator, the Simulink Debugger, and the Simulink Profiler) have no effect on referenced models executing in Accelerator mode. Specifications made and actions taken by such tools are ignored and effectively do not exist.

Subsystem Limitations

- If a subsystem contains Model blocks, you cannot build a subsystem module by right-clicking the subsystem or by using **Tools > Real-Time Workshop > Build subsystem**.

- If you generate code for an atomic subsystem as a reusable function, inputs or outputs that connect the subsystem to a referenced model can affect code reuse, as described in "Reusable Code and Referenced Models".

Target Limitations

- Real-Time Workshop `grt_malloc` targets do not support model reference.

- The Real-Time Workshop S-function target does not support model referencing.

Other Limitations

- Errors or unexpected behavior can occur if a Model block is part of a cycle, the Model block is a direct feedthrough block, and an algebraic loop results. See "Model Blocks and Direct Feedthrough" for details.

- The **External mode** option is not supported. If it is enabled, it is ignored by Accelerator mode.

6

Working with Blocks

About Blocks

In this section...
"What are Blocks?" on page 6-2
"Block Data Tips" on page 6-2
"Virtual Blocks" on page 6-2

What are Blocks?

Blocks are the elements from which the Simulink® software builds models. You can model virtually any dynamic system by creating and interconnecting blocks in appropriate ways. This section discusses how to use blocks to build models of dynamic systems.

Block Data Tips

Information about a block is displayed in a pop-up window when you allow the pointer to hover over the block in the diagram view. To disable this feature or control what information a data tip includes, select **Block data tips options** from the Simulink **View** menu.

Virtual Blocks

When creating models, you need to be aware that Simulink blocks fall into two basic categories: nonvirtual 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.

Block Name	Condition Under Which Block Is Virtual
Enable	Virtual unless connected directly to an Outport block.
From	Always virtual.
Goto	Always 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 only when the block's **Number of input dimensions** parameter specifies 1 and its **Index Option** specifies either Select all, Index vector (dialog), or Starting index (dialog).
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

In this section...

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 7-13 for more information.

Note The names of Sum, Mux, Demux, Bus Creator, and Bus Selector blocks are hidden when you copy them from the Simulink block library to a model. This is done to avoid unnecessarily cluttering the model diagram. (The shapes of these blocks clearly indicate their respective functions.)

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

1 Select the block you want to copy.

2 Choose **Copy** from the **Edit** menu.

3 Make the target model window the active window.

4 Choose **Paste** from the **Edit** menu.

Simulink assigns a name to each copied block. If it is the first block of its type in the model, its name is the same as its name in the source window. For example, if you copy the Gain block from the Math library into your model window, the name of the new block is Gain. If your model already contains a block named Gain, Simulink adds a sequence number to the block name (for example, Gain1, Gain2). You can rename blocks; see "Manipulating Block Names" on page 6-30.

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. An error message is displayed 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.

Note The model editor sorts block names alphabetically when generating names for copies pasted into a model. This can cause the names of pasted blocks to be out of order. For example, supposed you copy a row of 16 gain blocks named Gain, Gain1, Gain2...Gain15 and paste them into the model. The names of the pasted blocks occur in the following order: Gain16, Gain17, Gain24...Gain23.

Deleting Blocks

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

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

Working with Block Parameters

In this section...
"About Block Parameters" on page 6-8
"Mathematical Versus Configuration Parameters" on page 6-8
"Setting Block Parameters" on page 6-9
"Specifying Numeric Parameter Values" on page 6-10
"Checking Parameter Values" on page 6-12
"Changing the Values of Block Parameters During Simulation" on page 6-16
"Inlining Parameters" on page 6-18
"Block Properties Dialog Box" on page 6-20
"State Properties Dialog Box" on page 6-27

About Block Parameters

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

Mathematical Versus Configuration Parameters

Block parameters fall into two broad categories. A *mathematical parameter* is a parameter used to compute the value of a block's output, for example, the Gain parameter of a Gain block. All other parameters are *configuration parameters*, for example, a Gain block's Name parameter. In general, you can change the values of mathematical but not configuration parameters

during simulation (see "Changing the Values of Block Parameters During Simulation" on page 6-16).

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 6-28 for more information).

- **Block Properties** dialog box

  Specifies various attributes that are common to all blocks (see "Block Properties Dialog Box" on page 6-20 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 6-9 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 13-2 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 6-10
- "Resolving Variable References in Block Parameter Expressions" on page 6-11
- "Using Parameter Objects to Specify Parameter Values" on page 6-11
- "Determining Parameter Data Types" on page 6-12

Using Workspace Variables in Parameter Expressions

Block parameter expressions can include variables defined in the model's mask and model workspaces and in the MATLAB workspace. Using a workspace variable facilitates updating a model that sets multiple block parameters to the same value, i.e., it allows you to update multiple parameters by setting the value of a single workspace variable. For more information, see "Resolving Symbols" on page 3-69 and "Specifying Numeric Values with Symbols" on page 3-71.

Using a workspace variable also allows you to change the value of a parameter during simulation without having to open a block's parameter dialog box. For more information, see "Changing the Values of Block Parameters During Simulation" on page 6-16.

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. If the variable is not defined in any workspace, Simulink halts compilation of the model and displays an error message. See "Resolving Symbols" on page 3-69 and "Specifying Numeric Values with Symbols" on page 3-71 for more information.

Using Parameter Objects to Specify Parameter Values

You can use `Simulink.Parameter` objects in parameter expressions to specify parameter values. For example, `K` and `2*K` are both valid parameter expressions where `K` is a workspace variable that references a `Simulink.Parameter` object. In both cases, Simulink uses the parameter object's `Value` property as the value of `K`. For more information, see "Resolving Symbols" on page 3-69 and "Specifying Numeric Values with Symbols" on page 3-71.

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 6-19 and "Parameterizing Model References" on page 5-26 for more information).

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

Determining Parameter Data Types

When Simulink compiles a model, each of the model's blocks determines a data type for storing the values of its parameters whose values are specified by MATLAB parameter expressions.

Most blocks use internal rules to determine the data type assigned to a specific parameter. Exceptions include the Gain block, whose parameter dialog box allows you to specify the data type assigned to the compiled value of its Gain parameter. You can configure your model to check whether the data type assigned to a parameter can accommodate the parameter value specified by the model (see "Data Validity Diagnostics Overview").

Obtaining Parameter Information. You can use get_param to find the system and block parameter values for your model. See "Model and Block Parameters" for a list of arguments get_param accepts.

The model's signal attributes and parameter expressions must be evaluated before some parameters are properly reported. This evaluation occurs during the simulation compilation phase. Alternatively, you can compile your model without first running it, and then obtain parameter information. For instance, to access the port width, data types and dimensions of the blocks in your model, enter the following at the command prompt:

```
modelname([],[],[],'compile')
q=get_param(gcb,'PortHandles');
get_param(q.Inport,'CompiledPortDataType')
get_param(q.Inport,'CompiledPortWidth')
get_param(q.Inport,'CompiledPortDimensions')
modelname([],[],[],'term')
```

Checking Parameter Values

Several blocks perform range checking of their mathematical parameters. Generally, blocks that allow you to enter minimum and maximum values check to ensure that the values of applicable parameters lie within the specified range. See the following topics for more information:

- "Blocks That Perform Parameter Range Checking" on page 6-13

- "Specifying Ranges for Parameters" on page 6-13

• "Performing Parameter Range Checking" on page 6-14

Blocks That Perform Parameter Range Checking

The following blocks perform range checking for their parameters:

Block	Parameters Checked
Constant	**Constant value**
Data Store Memory	**Initial value**
Gain	**Gain**
Interpolation Using Prelookup	**Table data**
Lookup Table	**Table data**
Lookup Table (2-D)	**Table data**
Lookup Table (n-D)	**Table data**
Relay	**Output when on** **Output when off**
Repeating Sequence Interpolated	**Vector of output values**
Repeating Sequence Stair	**Vector of output values**
Saturation	**Upper limit** **Lower limit**

Specifying Ranges for Parameters

In general, use the **Output minimum** and **Output maximum** parameters
that appear on a block parameter dialog box to specify a range of valid values
for the block parameters. The following exceptions apply:

• For the Gain block, use the **Parameter minimum** and **Parameter maximum** fields to specify a range for the **Gain** parameter.

• For the Data Store Memory block, use the **Minimum** and **Maximum** fields
to specify a range for the **Initial value** parameter.

When specifying minimum and maximum values that constitute a range,
enter only expressions that evaluate to a scalar, real number with `double` data

type. The default value, `[]`, is equivalent to `-Inf` for the minimum value and `Inf` for the maximum value. The scalar values that you specify are subject to expansion, for example, when the block parameters that Simulink checks are nonscalar (see "Scalar Expansion of Inputs and Parameters" on page 8-24).

Note You cannot specify the minimum or maximum value as `NaN`.

Specifying Ranges for Complex Numbers. When you specify a minimum or maximum value for a parameter that is a complex number, the specified minimum and maximum apply separately to the real part and to the imaginary part of the complex number. If the value of either part of the number is less than the minimum, or greater than the maximum, the complex number is outside the specified range. No range checking occurs against any combination of the real and imaginary parts, such as (`sqrt(a^2+b^2)`)

Performing Parameter Range Checking

You can initiate parameter range checking in the following ways:

- When you click the **OK** or **Apply** button on a block parameter dialog box, the block performs range checking for its parameters. However, the block checks only the parameters that it can readily evaluate. For example, the block does not check parameters that use an undefined workspace variable.

- When you start a simulation or select **Update Diagram** from the Simulink **Edit** menu, Simulink performs parameter range checking for all blocks in that model.

Simulink performs parameter range checking by comparing the values of applicable block parameters with both the specified range (see "Specifying Ranges for Parameters" on page 6-13) and the block data type. That is, Simulink performs the following check:

```
DataTypeMin ≤ MinValue ≤ VALUE ≤ MaxValue ≤ DataTypeMax
```

where

- `DataTypeMin` is the minimum value representable by the block data type.

- `MinValue` is the minimum value the block should output, specified by, e.g., **Output minimum**.

- `VALUE` is the numeric value of a block parameter.

- `MaxValue` is the maximum value the block should output, specified by, e.g., **Output maximum**.

- `DataTypeMax` is the maximum value representable by the block data type.

When Simulink detects a parameter value that violates the check, it displays an error message. For example, consider a model that contains a Constant block whose

- **Constant value** parameter specifies the variable `const`, which you have yet to define in a workspace.

- **Output minimum** and **Output maximum** parameters are set to `2` and `8`, respectively.

- **Output data type** parameter is set to `uint8`.

In this situation, Simulink does not perform parameter range checking when you click the **OK** button on the Constant block dialog box because the variable `const` is undefined. But suppose you define its value by entering

```
const = 10
```

at the MATLAB prompt, and then you update the diagram (see "Updating a Block Diagram" on page 1-13). Simulink displays the following error message:
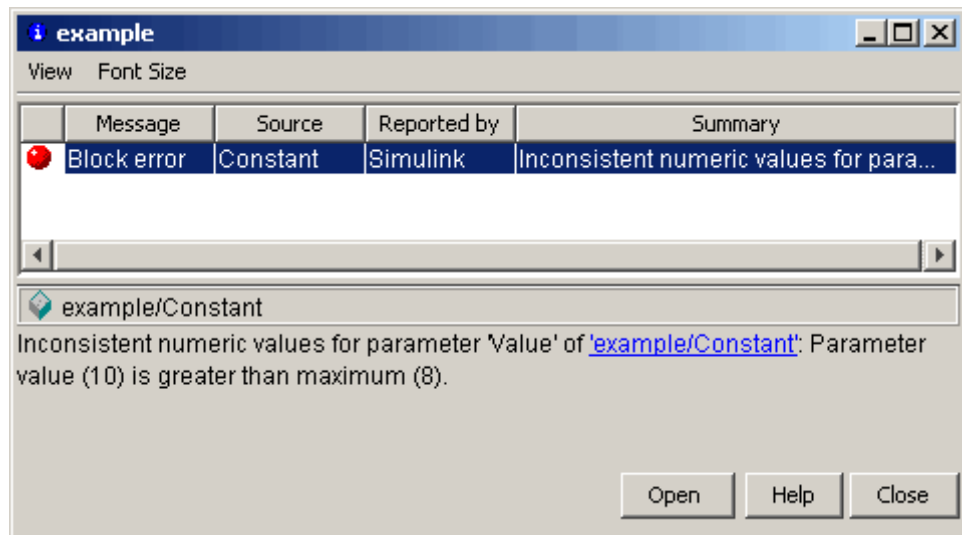
Changing the Values of Block Parameters During Simulation

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

Note You cannot tune inline parameters. See "Inlining Parameters" on page 6-18 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 6-17). To use the block's parameter dialog box, open the block's parameter dialog box, change the value displayed in the dialog box, and click the dialog box's **OK** or **Apply** button.

You can also tune a parameter at the MATLAB command line, using either the set_param command or by assigning a new value to the MATLAB workspace variable that specifies the parameter's value. In either case, you must update the model's block diagram for the change to take effect (see "Updating a Block Diagram" on page 1-13).

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 6-17) causes a running simulation to pause. While the simulation is paused, you can edit the parameter values displayed on the dialog box. However, you must close the dialog box to have the changes take effect and allow the simulation to continue. Similarly, starting a simulation causes any open dialog boxes associated with source blocks with tunable parameters to close.

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

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

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

- Simulink source blocks, including
  - Band-Limited White Noise
  - Chirp Signal
  - Constant
  - Pulse Generator
  - Ramp
  - Random Number
  - Repeating Sequence
  - Signal Generator
  - Sine Wave
  - Step
  - Uniform Random Number
- User-developed masked subsystem blocks that have one or more tunable parameters and one or more output ports, but no input ports.
- S-Function and M-file (level 2) S-Function blocks that have one or more tunable parameters and one or more output ports but no input ports.

Inlining Parameters

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

Specifying Some Parameters as 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" 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 6-11), 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.

The dialog box contains the following tabbed panes:

- "General Pane" on page 6-21

- "Block Annotation Pane" on page 6-23

- "Callbacks Pane" on page 6-25

General Pane

This pane allows you to set the following properties.
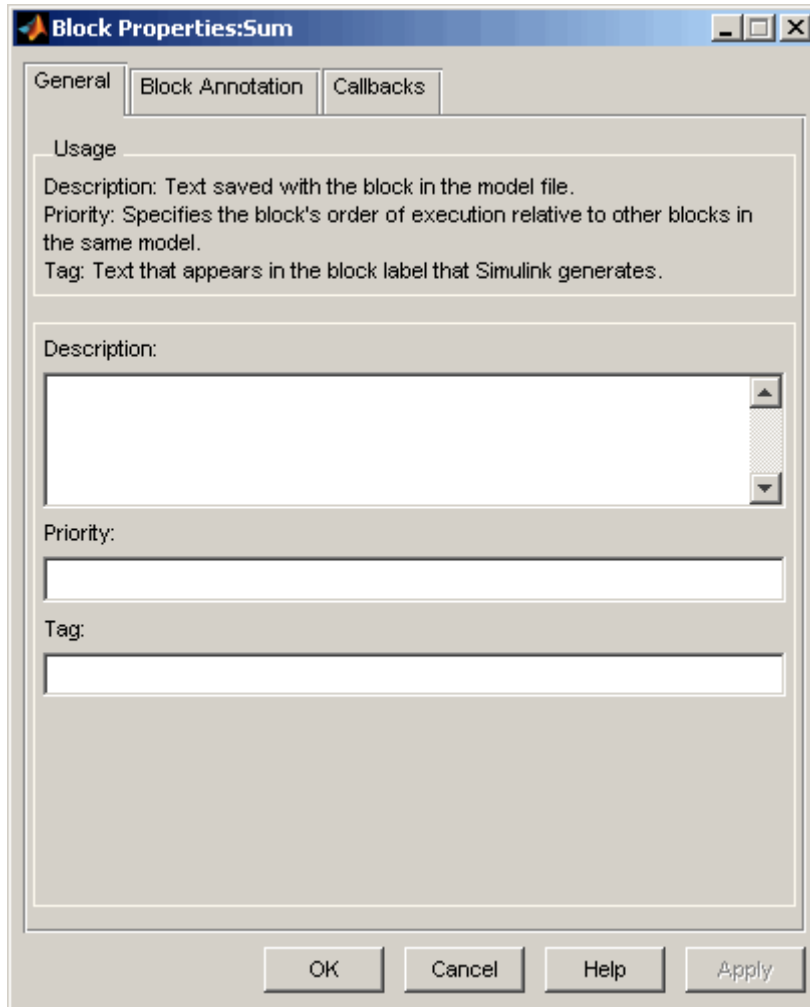
Description. Brief description of the block's purpose.

Priority. Execution priority of this block relative to other blocks in the model. See "Assigning Block Priorities" on page 6-39 for more information.

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

Block Annotation Pane

The block annotation pane allows you to display the values of selected block parameters in an annotation that appears beneath the block's icon.
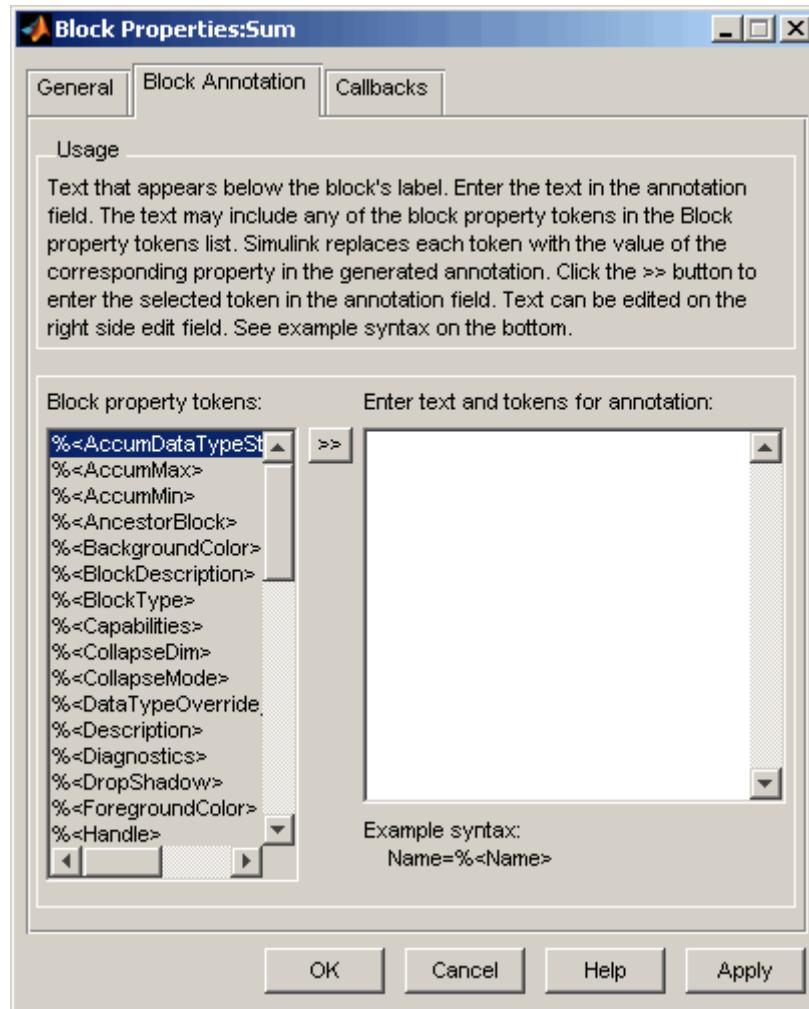
Enter the text of the annotation in the text field that appears on the right side of the pane. The text can include any of the block property tokens that appear in the list on the left side of the pane. A block property token is simply the

name of a block parameter preceded by %< and followed by >. When displaying the annotation, the Simulink software replaces the tokens with the values of the corresponding block parameters. For example, suppose that you enter the following text and tokens for a Product block:

```
Multiplication = %<Multiplication>
Sample time = %<SampleTime>
```

In the model editor window, the annotation appears as follows:
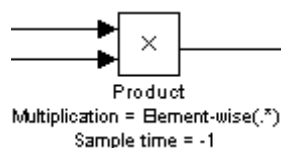
The block property token list on the left side of the pane lists all the parameters that are valid for the currently selected block (see "Model and Block Parameters" in the *Simulink Reference*). To add one of the listed tokens to the text field on the right side of the pane, select the token and then click the button between the list and the text field.

You can also create block annotations programmatically. See "Creating Block Annotations Programmatically" on page 6-26.

Callbacks Pane

The **Callbacks Pane** allows you to specify implementations for a block's callbacks (see "Using Callback Functions" on page 3-52).
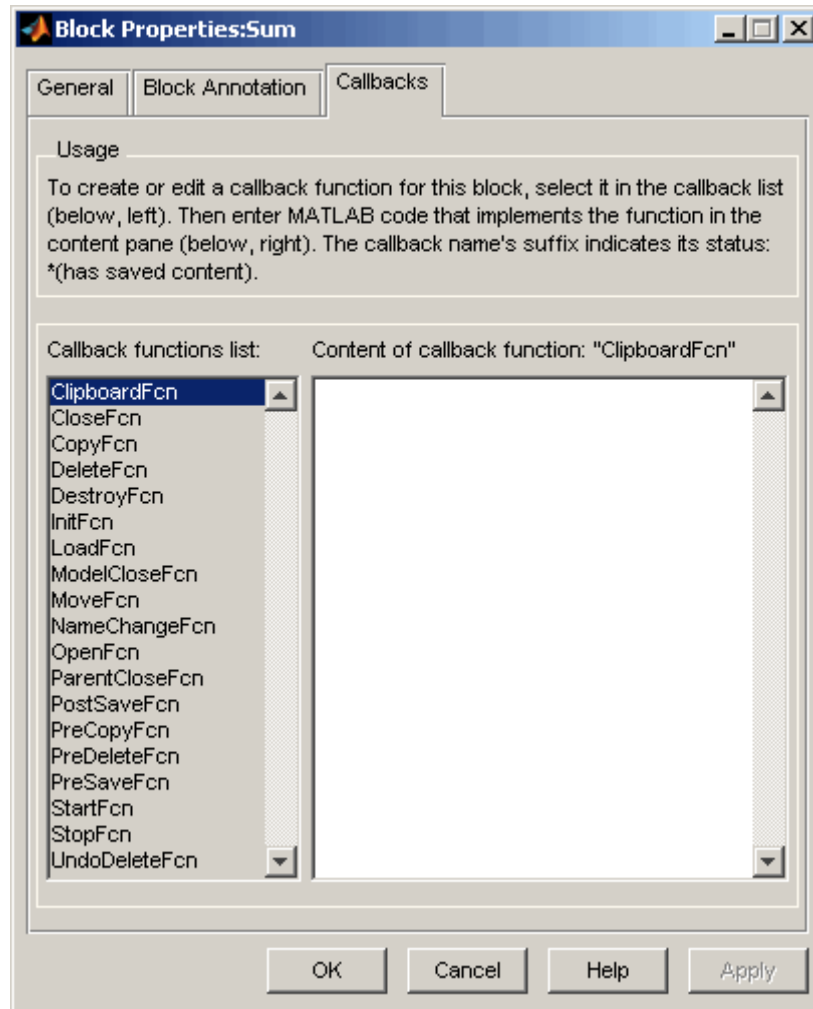
To specify an implementation for a callback, select the callback in the callback list on the left side of the pane. Then enter MATLAB commands that implement the callback in the right-hand field. Click **OK** or **Apply** to save the

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

Creating Block Annotations Programmatically

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

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

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

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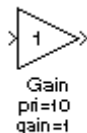
If a parameter's value is not a string or an integer, Simulink displays `N/S` (not supported) for the parameter's value. If the parameter name is invalid, Simulink displays `???` as the parameter value.

State 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

In this section...

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.
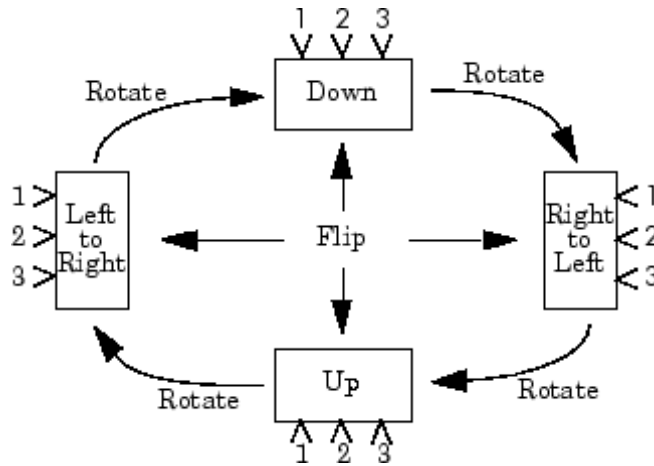
Resizing a Block

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:
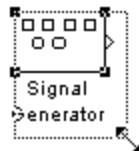
Displaying Parameters Beneath a Block

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 6-20)

- 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 6-28.

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

Block Output Example

For many blocks, Simulink® can display block outputs as data tips on the block diagram while a simulation is running.

van der Pol Equation

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

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu
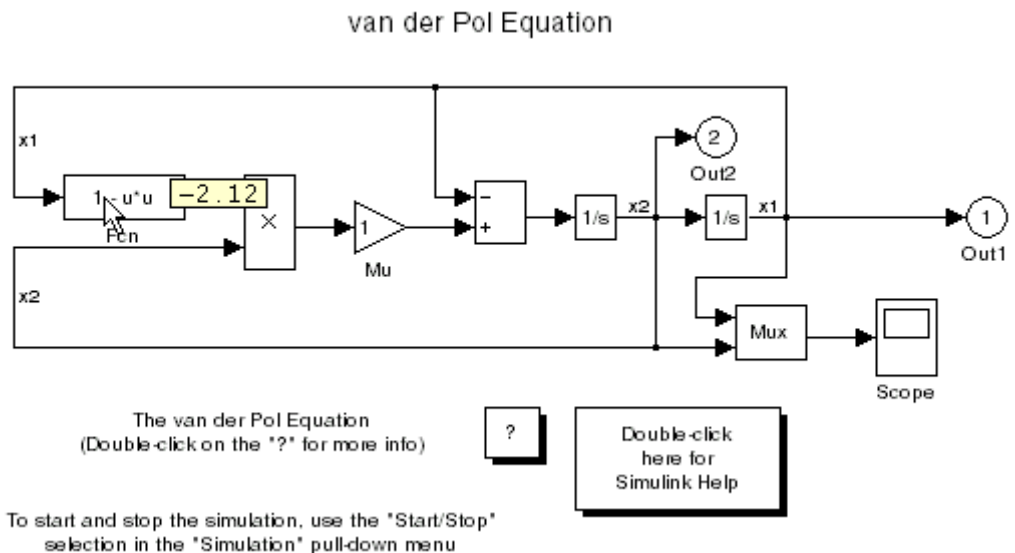
Additionally, you can specify whether and when to display block outputs and the size and format of the output displays and the rate at which Simulink updates them during a simulation.

> **Note** Port values will not be displayed in subsystems that have direct feed through for all of its signals. To display port values in subsystems, add a unity gain block to at least one of the signals.

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 Simulink version that runs on the Microsoft® Windows® operating system, 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.

Click to show/hide block output when hovering
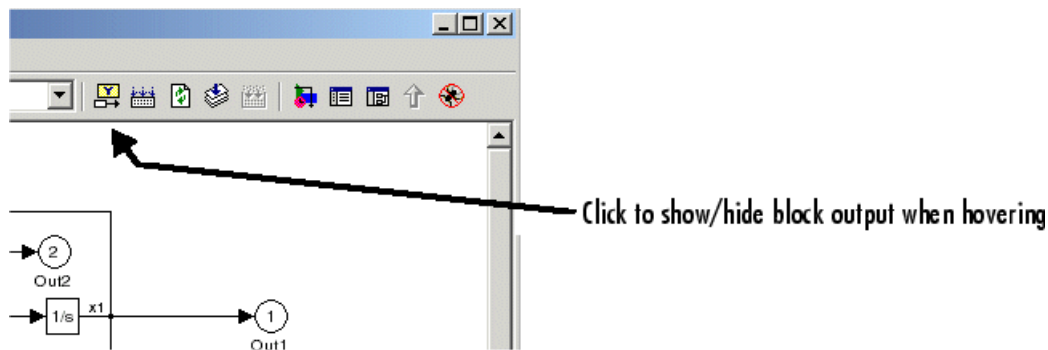
Port Values Display Options

To specify other display options, select **Port Values > Options** from the Model Editor's **View** menu. The **Block Output Display Options** dialog box appears.
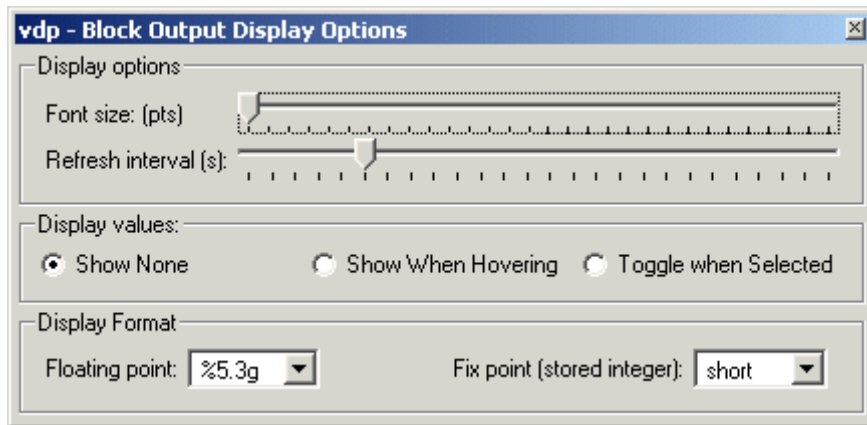
To increase the size of the output display text, move the **Font size** slider to the right. To increase the rate at which Simulink updates the displays, move the **Refresh interval** slider to the left.

Controlling and Displaying the Sorted Order

What is Block 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 2-12) 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.

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 6-37) 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 2-31.

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 **Format > Block Displays > Sorted Order** from the Simulink menu. Selecting this option causes Simulink to display a notation in the top right corner of each block in a block diagram.
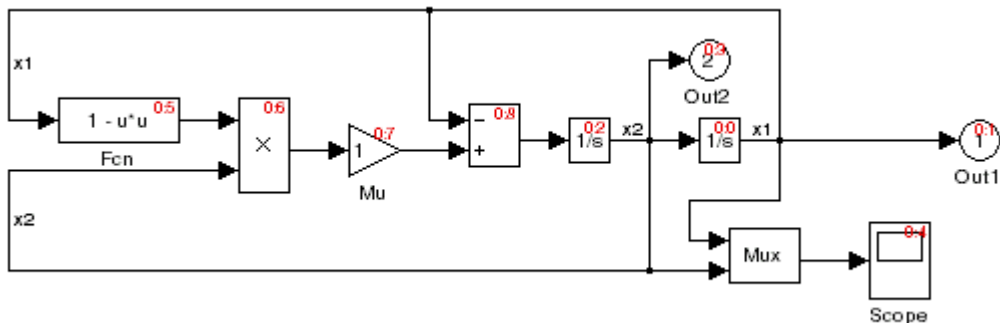
The notation for most blocks has the format *s*:*b*, where *s* specifies the index of the subsystem to whose execution context (see "Conditional Execution Behavior" on page 4-22) the block belongs and *b* specifies the block's position in the sorted order for that execution context.

The sorted order of a Function-Call Subsystem cannot be determined at compile time. For these subsystems, Simulink therefore 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.

A bus-capable block does not execute as a unit, and therefore does not have a unique sorted order. Such a block displays its sorted order as *s*:B. See "Bus-Capable Blocks" on page 9-8 for more information.

A virtual block, such as the Mux block in the preceding figure, exists only graphically. Virtual blocks do not execute, so they are not part of a model's sorted order and do not display any sorted order notation.

Assigning Block Priorities

You can assign priorities to nonvirtual blocks or virtual subsystem blocks in a model (see "Virtual Blocks" on page 6-2). 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 6-20).

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: Solver").

Accessing Block Data During Simulation

In this section...
"About Block Run-Time Objects" on page 6-40
"Accessing a Run-Time Object" on page 6-40
"Listening for Method Execution Events" on page 6-41
"Synchronizing Run-Time Objects and Simulink® Execution" on page 6-42

About Block Run-Time Objects

Simulink® provides an application programming interface, called the block run-time interface, that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to access block run-time data from the MATLAB® command line, the Simulink Debugger, and from Level-2 M-file S-functions (see "Writing S-Functions in M" in the online Simulink documentation).

Note You can use this interface even when the model is paused or is running or paused in the debugger.

The block run-time interface consists of a set of Simulink data object classes (see "Working with Data Objects" on page 10-27) 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 6-2) do not have run-time objects. Blocks eliminated during model compilation as an optimization also do not have run-time objects (see "Block reduction"). A run-time object exists only while the model containing the block is running or paused. If the model is stopped, `get_param` returns an empty handle. When you stop or pause a model, all existing handles for run-time objects become empty.

Listening for Method Execution Events

One application for the block run-time API is to collect diagnostic data at key points during simulation, such as the value of block states before or after blocks compute their outputs or derivatives. The block run-time API provides an event-listener mechanism that facilitates such applications. For more information, see the *Simulink Reference* for the `add_exec_event_listener` command. For an example of using method execution events, enter

```
sldemo_msfcn_lms
```

at the MATLAB command line. This Simulink model contains the S-function `adapt_lms.m`, which performs a system identification to determine the coefficients of an FIR filter. The S-function's `PostPropagationSetup` method initializes the block run-time object's DWork vector such that the second vector stores the filter coefficients calculated at each time step.

In the Simulink model, double-clicking on the annotation below the S-function block executes its `OpenFcn`. This function first opens a figure for plotting the FIR filter coefficients. It then executes the function `add_adapt_coef_plot.m` to add a `PostOutputs` method execution event to the S-function's block run-time object using the following lines of code.

```
% Get the full path to the S-function block
blk = 'sldemo_msfcn_lms/LMS Adaptive';

% Attach the event-listener function to the S-function
h   = add_exec_event_listener(blk, ...
```

```
                    'PostOutputs', @plot_adapt_coefs);
```

The function plot_adapt_coefs.m is registered as an event listener that is executed after every call to the S-function's Outputs method. The function accesses the block run-time object's DWork vector and plots the filter coefficients calculated in the Outputs method. The calling syntax used in plot_adapt_coefs.m follows the standard needed for any listener. The first input argument is the S-function's block run-time object, and the second argument is a structure of event data, as shown below.

```
function plot_adapt_coefs(block, eventData)

% The figure's handle is stored in the block's UserData
hFig = get_param(block.BlockHandle,'UserData');
tAxis = findobj(hFig, 'Type','axes');

tAxis = tAxis(2);
tLines = findobj(tAxis, 'Type','Line');

% The filter coefficients are stored in the block run-time
%   object's second DWork vector.
est = block.Dwork(2).Data;

set(tLines(3),'YData',est);
```

Synchronizing Run-Time Objects and Simulink® Execution

Run-time objects can be used at the MATLAB command line to obtain the value of a block's output by entering the following commands.

```
rto = get_param(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.

Working with Block Libraries

About Block Libraries

Libraries are collections of blocks intended to serve as prototypes for creating instances of block types in models. Simulink® software uses a special type of model file to store block libraries.

You create instances of block types by dragging and dropping or copying library blocks into models. When you copy a library block into a model, Simulink software creates a *link* between the instance, called a *reference block*, and its prototype in the library. The link allows changes in the prototype to propagate automatically to the instances in a model. Libraries ensure that your models automatically include the most recent versions of blocks developed by yourself or others.

Simulink comes with a library of commonly used block types called the Simulink block library. See "Starting the Simulink® Engine" on page 1-2 for information on displaying and using this library. Additional libraries are available from The MathWorks. You can also create your own block libraries (see "Creating Block Libraries" on page 7-20).

Note Although the Real-Time Workshop® block library is provided with Simulink to enable model sharing, help for blocks in that library is available only if you have a Real-Time Workshop license.

Working with Reference Blocks

In this section...

About Reference Blocks

A *reference block* is an instance of a block type in a model that contains a link to a library block that serves as the block type's prototype. The link consists of the path of the library block that serves as the instance's prototype. The link allows the reference block to update whenever the corresponding prototype in the library changes ("Updating a Reference Block" on page 7-4). This ensures that your model always uses the latest version of the block.

Note The data tip for a reference block shows the name of the library block it references (see "Block Data Tips" on page 6-2).

You can change the values of a reference block's parameters but you cannot mask the block or edit its mask. Also, you cannot set callback parameters for a reference block. If the reference block's prototype is a subsystem, you can make nonstructural changes to the contents of the referenced subsystem (see "Modifying Reference Blocks" on page 7-4).

Creating a Reference Block

To create a reference block in a model or another library:

1 Open your model.

2 Open the Simulink® Library Browser (see "Opening the Library Browser" on page 7-12).

3 Use the Library Browser to find the library block that serves as a prototype of the block you want to create (see "Browsing Block Libraries" on page 7-13 and "Searching Block Libraries" on page 7-17).

4 Drag the library block from the Library Browser's **Library** pane and drop it into your model.

Updating a Reference Block

Simulink updates out-of-date reference blocks in a model or library at these times:

- When the model or library is loaded

- When you select **Update Diagram** from the **Edit** menu or run the simulation

- When you use the `find_system` command

- When you query the `LinkStatus` parameter of a block, using the `get_param` command (see "Determining Link Status" on page 7-8)

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

Modifying Reference Blocks

You cannot make structural changes to reference blocks, such as adding or deleting lines or blocks to the block diagram of a masked subsystem. If you want to make such changes, you must disable the reference block's link to its library prototype (see "Disabling Links to Library Blocks" on page 7-7).

You can, however, change the values of any masked subsystem reference block parameter that does not alter the block's structure, e.g., by adding or deleting lines, blocks, or ports. An example of a nonstructural change is a change to the value of a mathematical block parameter, such as the Gain parameter of the Gain block. A link to a library block from a reference block whose parameter values differ from those of the corresponding library block is

called a *parameterized link*. When saving a model containing a parameterized link, Simulink saves the changes to the local copy of the subsystem together with the path to the library copy in the model's model (.mdl) file. When you reopen the system, Simulink copies the library subsystem into the loaded model and applies the saved changes.

Tip To determine whether a reference block's parameter values differ from those of its library prototype, open the reference block's block diagram in an editor window. The title bar of the editor window displaying the subsystem displays "parameterized link" if the reference block parameter values differ from the library block's parameter values.

Self-Modifying Linked Subsystems

Simulink allows linked subsystems to change their own structural contents without disabling the link. This allows you to create masked subsystems that modify their structural contents based on mask parameter dialog box values. See "Creating Self-Modifying Masks for Library Blocks" on page 17-48 for more information.

Finding a Reference Block's Library Block Prototype

To find the source library and block linked to a reference block, select the reference block. Then choose **Go To Library Link** from the **Link Options** submenu of the model window's **Edit** or context menu. If the library is open, Simulink selects and highlights the library block and makes the source library the active window. If the library is not open, Simulink opens it and selects the library block.

Getting Information About Library Blocks Referenced by a Model

Use the libinfo command to get information about reference blocks in a system.

Working with Library Links

Displaying Library Links

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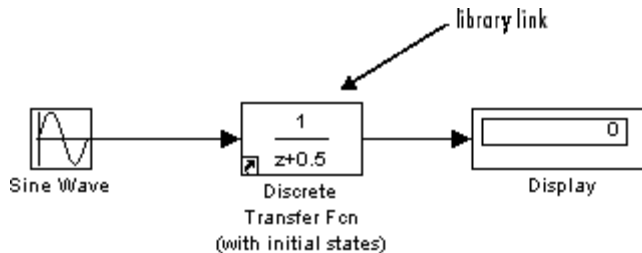
This arrow allows you to tell at a glance whether a block represents a link to a library block or a local instance of a block. To enable display of library links, select **Library Link Display** from the model window's **Format** menu and then select either **User** (displays only links to user libraries)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

Disabling Links to Library Blocks

Simulink allows you to disable the link between a reference block in a model and the library block that serves as its prototype. Disabling a link allows you to make structural changes to a subsystem reference block. To disable a link, select the link, choose **Link options** from the model window's **Edit** or context menu, then choose **Disable link**. Simulink displays "disabled link" in the title bar of a Model Editor window that displays the local copy of a disabled link to a masked subsystem in a library. To restore a disabled link, choose **Restore link** from the **Link Options** menu.

Note If you attempt to use the Model Editor 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 disables the link and 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.

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.

Determining Link Status

All blocks have a `LinkStatus` parameter and a `StaticLinkStatus` parameter that indicate whether the block is a reference block. The parameters can have these values.

Status	Description
none	Block is not a reference block.
resolved	Link is resolved.
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.

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 standalone 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 standalone, especially if the model includes blocks from third-party libraries or optional Simulink blocksets. It is possible that a library block invokes functions supplied with the library and hence can run only if the library is installed on the system running the model. Further, breaking a link can cause a model to fail when you install a new version of the library on a system. For example, suppose a block invokes a function that is supplied with the library. Now suppose that a new version of the library eliminates the function. Running a model with an unlinked copy of the block results in invocation of a now nonexistent function, causing the simulation to fail. To avoid such problems, you should generally avoid breaking links to third-party libraries and optional Simulink blocksets.

Fixing Unresolved Library Links

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

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

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

Reference Block Name

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

- Delete the unlinked reference block and copy the library block back into your model.

- Add the directory that contains the required library to the MATLAB path and select **Update Diagram** from the **Edit** menu.

- Double-click the unlinked reference block to open its dialog box (see the Bad Link block reference page). On the dialog box that appears, correct the pathname in the **Source block** field and click **OK**.

Browsing and Searching Block Libraries

In this section...

About the Simulink® Library Browser

The Simulink® Library Browser (see "Library Browser") allows you to browse and search the block libraries installed on your system. In particular, you can use the Library Browser to determine at a glance what libraries are installed on your system, view their contents, and search for library blocks by name. You can also use the Library Browser to copy blocks from block libraries into your models.

Tip You can use the Library Browser to browse libraries that you create as well as standard Simulink and Simulink blockset libraries. See "Adding Libraries to the Library Browser" on page 7-30 for more information.

Opening the Library Browser

You can open the Library Browser by

- clicking the **Library Browser** button

  in the toolbar of the MATLAB® desktop or a Simulink model editor window

  or

- entering

      simulink

  at the MATLAB command line.

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

Browsing Block Libraries

- "Determining What Block Libraries Are Installed on your System" on page 7-13
- "Viewing the Contents of a Block Library" on page 7-15
- "Getting Help for a Library Block" on page 7-17

Determining What Block Libraries Are Installed on your System

To determine what block libraries are installed on your system, open the Simulink Library Browser (see "Opening the Library Browser" on page 7-12). The Library Browser's **Libraries** pane displays a tree-structure directory of the block libraries installed on your system.

Libraries pane
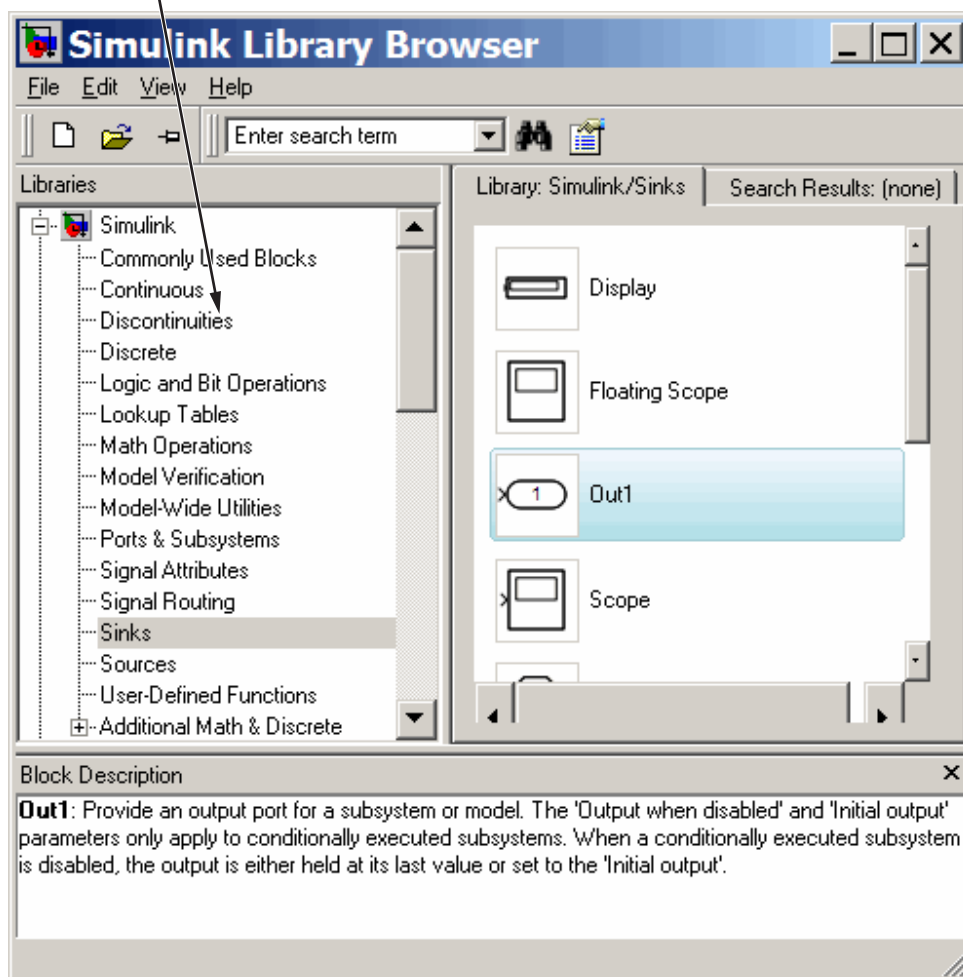
Tip

- If the directory is too big to fit into the **Libraries** pane, the pane displays scroll bars. Use the scroll bars to move off-screen libraries into view.

- You can expand or collapse directory entries to show or hide sublibraries. To expand or collapse an entry, click the entry's +/- button.

- Click a library's name in the directory to display the library's contents in the Library Browser's **Library** pane.

Viewing the Contents of a Block Library

To view the contents of a block library, select the library's entry in the Library Browser's **Libraries** pane. The Library Browser displays the contents in its **Library** pane.

Contents of the Simulink
Commonly Used Blocks
library
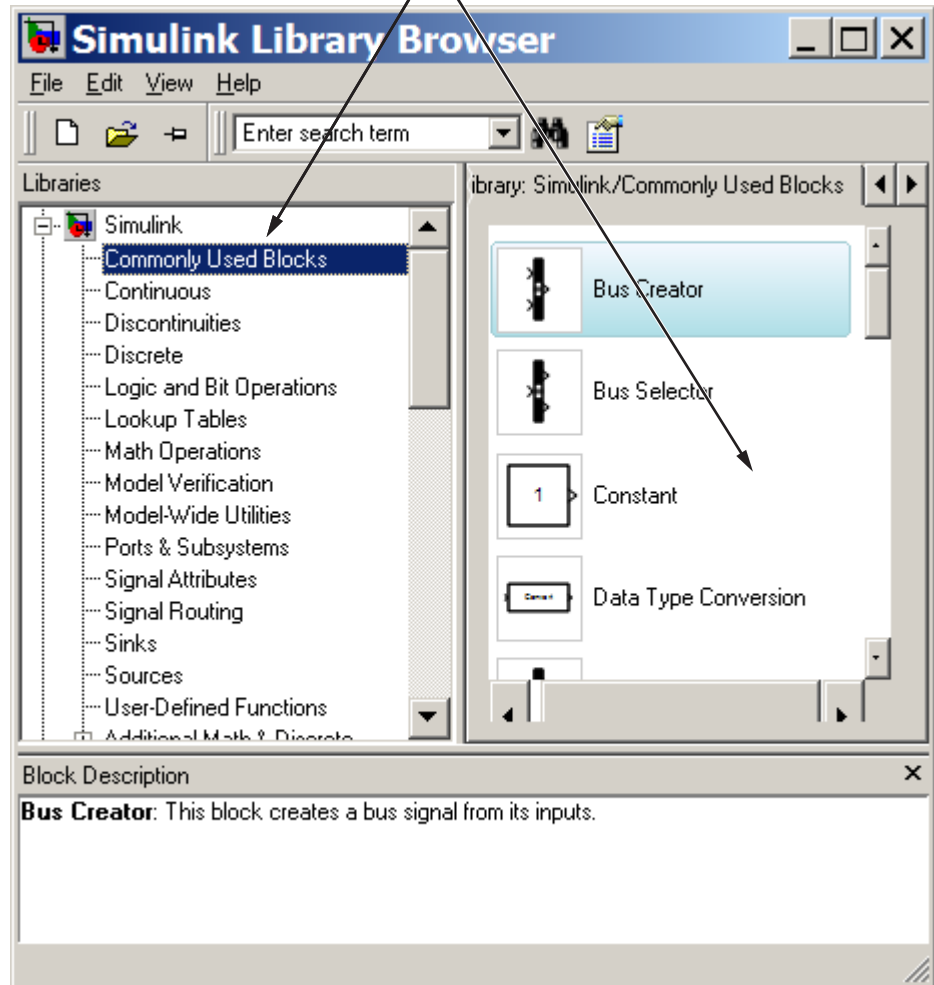
Getting Help for a Library Block

To get help for a library block displayed in the Library Browser's **Library** pane, select the block and then select **Help** from the block's context (right-click) menu or the Library Browser's **Help** menu.

Tip The Library Browser's **Block Description** pane displays a brief description of the selected block's purpose and usage.

Searching Block Libraries

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

1 Enter the character string in the text field of the Library Browser's **Search** tool.

2 Uncheck, if necessary, all the options on the **Search** tool's options menu.

Note The options are unchecked by default.

3 Select the tool's **Search** button.

The **Search** tool finds the blocks whose names contain a string that matches, without regard to character case, the specified string and displays the results in the Library Browser's **Found** pane.

Search text   Search button   Click to display search options menu

Search results

Click to hide search results
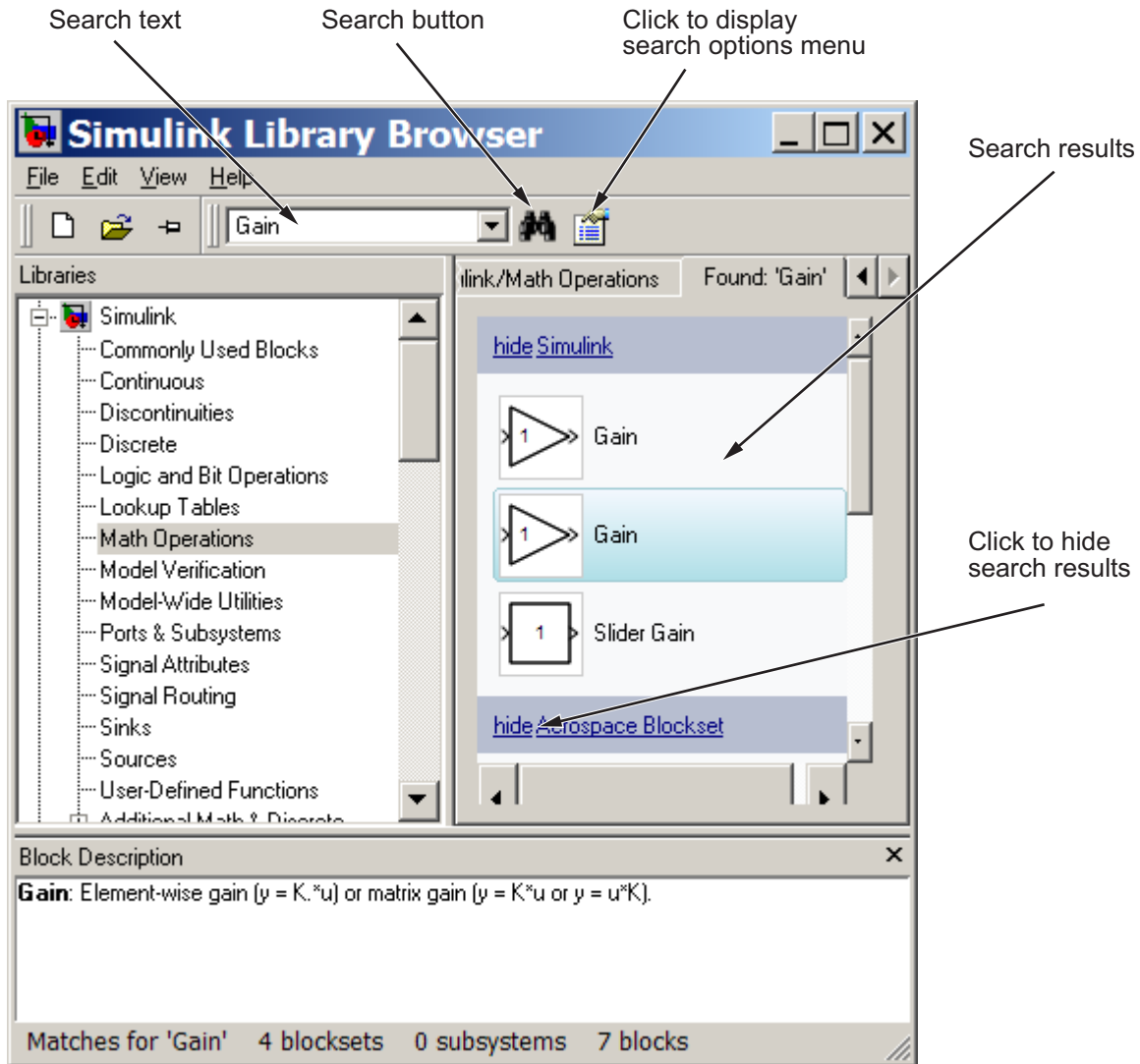
Tip Use the **Search** tool's options menu to find blocks whose names match character case, whole words, or MATLAB regular expressions.

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.

Creating Block Libraries

In this section...
"Creating a Library" on page 7-20
"Creating a Sublibrary" on page 7-21
"Modifying a Library" on page 7-21
"Locking Libraries" on page 7-22
"Making Backward-Compatible Changes to Libraries" on page 7-22

Creating a Library

You can create your own block library and add it to the Simulink® Library Browser (see "Adding Libraries to the Library Browser" on page 7-30).

Tip If your library contains many blocks, consider grouping the blocks into a hierarchy of sublibraries (see "Creating a Sublibrary" on page 7-21).

To create a library:

1 Select **Library** from the **New** submenu of the **File** menu.

Simulink creates a model (`*.mdl`) file for storing the new library and displays the file in a new model editor window.

Tip You can also use the `new_system` command to create the library and the `open_system` command to open the new library.

---

2 Drag blocks from models or other libraries into the new library.

> **Note** If you want to be able to create links in models to a block in the library, you must provide a mask (see Chapter 17, "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.

3 Save the library's model file under a new name.

Creating a Sublibrary

Creating a sublibrary entails inserting a reference in the model (`.mdl`) file of one library to the model file of another library. The referenced file is called a *sublibrary* of the parent (i.e., referencing) library. The sublibrary is said to be included by reference in the parent library.

To include a library in another library as a sublibrary:

1 Open the parent library.

2 Unlock the parent library (see "Modifying a Library" on page 7-21).

3 Add a Subsystem block to the parent library.

4 Delete the subsystem's default input and output ports.

5 Create a mask for the subsystem that displays text or an image that conveys the sublibrary's purpose.

6 Set the subsystem's OpenFcn parameter to the name of the sublibrary's model file.

7 Save the parent library.

Modifying a Library

When you open a library, it is automatically locked and you cannot modify its contents. To unlock the library, select **Unlock Library** from the **Edit** menu. Closing the library window locks the library.

Locking Libraries

To lock a block library, save and close the library or set its Lock parameter to 'on' at the MATLAB® command line, using the set_param command. Locking a library prevents a user from inadvertently modifying a library, for example, by moving a block in the library or adding or deleting a block from the library. If you attempt to modify a locked library, Simulink displays a dialog box that allows you to unlock the library and make the change. You must then relock the library from the MATLAB command line to prevent further changes.

Making Backward-Compatible Changes to Libraries

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

Forwarding Tables

Library forwarding tables enable Simulink to update models to reflect changes in the names or locations of the library blocks that they reference. For example, suppose that you rename a block in a library. You can use a forwarding table for that library to enable Simulink to update models that reference the block under its old name to reference it under its new name.

Simulink allows you to associate a forwarding table with any library. The forwarding table for a library specifies the old locations and new locations of blocks that have moved within the library or to another library. You associate a forwarding table with a library by setting its ForwardingTable parameter to a cell array of two-element cell arrays, each of which specifies the old and new path of a block that has moved. For example, the following command creates a forwarding table and assigns it to a library named Lib1.

```
set_param('Lib1', 'ForwardingTable', {{'Lib1/A', 'Lib2/A'}
{'Lib1/B', 'Lib1/C'}});
```

The forwarding table specifies that block A has moved from Lib1 to Lib2. and that block B is now named C. Suppose that you open a model that contains links to Lib1/A and Lib1/B. Simulink updates the link to Lib1/A to refer to Lib2/A and the link to Lib1/B to refer to Lib1/C. The changes become permanent when you subsequently save the model.

Creating Aliases for Mask Parameters

Simulink lets you create aliases, i.e., alternate names, for a mask's parameters. A model can then refer to the mask parameter by either its name or its alias. This allows you to change the name of a mask parameter in a library block without having to recreate links to the block in existing models (see "Example: Using Mask Parameter Aliases to Create Backward-Compatible Parameter Name Changes" on page 7-23).

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 17-6 in `mymdl`.

3 Name the masked subsystem Line.

4 Set the masked subsystem's annotation property (see "Block Annotation Pane" on page 6-23) to display the value of its `m` and `b` parameters, i.e., to

```
m = %<m>
b = %<b>
```

The library appears as follows:
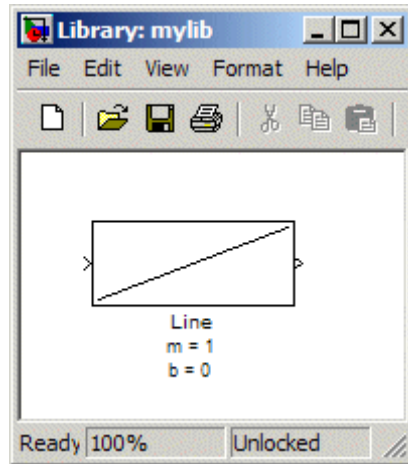
5 Save mylib.

6 Create a model named mymdl.

7 Create an instance of the Line block in mymdl.

8 Rename the instance LineA.

9 Change the value of LineA's m parameter to -1.5.

10 Change the value of LineA's b parameter to 3.

11 Set LineA's annotation property to display the values of its m and b parameters.
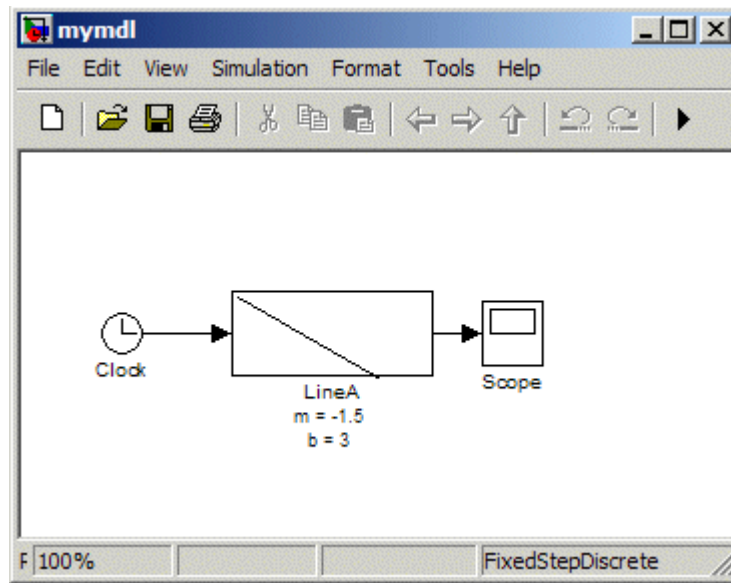
12 Configure mymdl to use a fixed-step, discrete solver with a 0.1-s step size.

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.
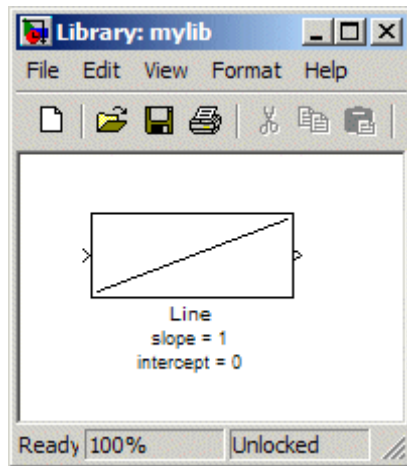
20 Save `mylib`.

21 Reopen `mymdl`.
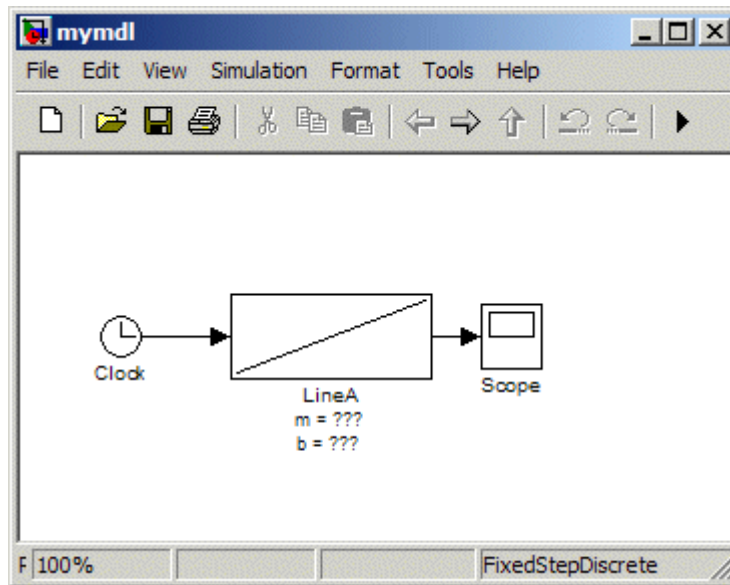
Note that LineA's icon has reverted to the appearance of its library master (i.e., `mylib/Line`) and that its annotation displays question marks for the values of m and b. These changes reflect the parameter name changes in the library block . In particular, Simulink cannot find any parameters named 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`.
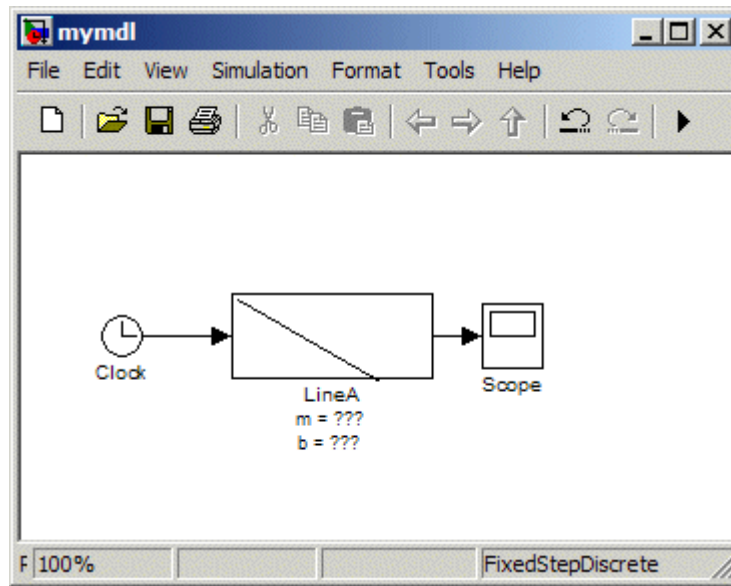
Note that LineA's appearance now reflects the value of the slope parameter under its original name, i.e., m. This is because when Simulink opened the model it found that m is an alias for slope and assigned the value of m stored in the model file to LineA's slope parameter.

27 Change LineA's block annotation property to reflect LineA's parameter name changes, i.e., replace

```
m = %<m>
b = %<b>
```

with

```
m = %<slope>
b = %<intercept>
```

LineA now appears as follows.
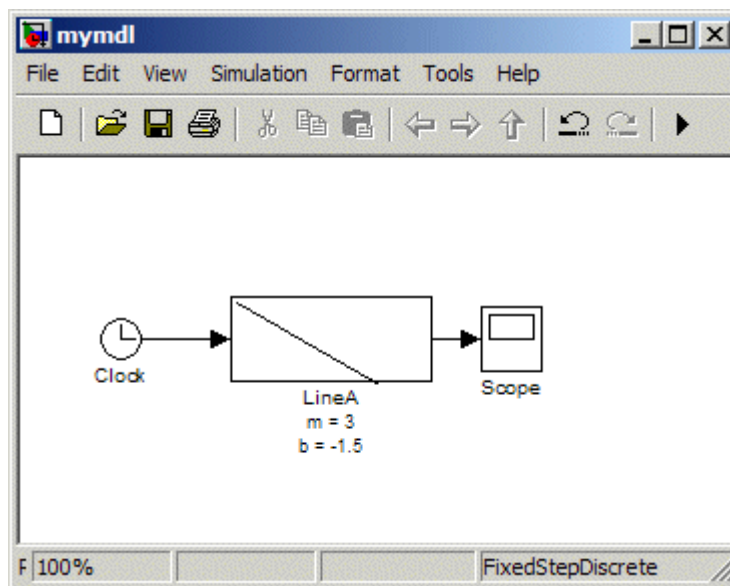
Note that LineA's annotation shows that, thanks to parameter aliasing, Simulink has correctly applied the parameter values stored for LineA in mymdl's model file to the block's renamed parameters.

Adding Libraries to the Library Browser

To cause your own top-level library (and its sublibraries) to appear in the Simulink® Library Browser:

1 Create a directory in the MATLAB® path for the top-level library and its sublibraries.

Each top-level 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 top-level libraries cannot exist in the same directory.

2 Create or copy the top-level library and its sublibraries into its directory.

The directory for each top-level library to be displayed in the Library Browser must contain a file named slblocks.m that describes the library. 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.

8

Working with Signals

Signal Basics

About Signals

The term *signal* refers to a time varying quantity that has values at all points in time. You can specify a wide range of signal attributes, including signal name, data type (e.g., 8-bit, 16-bit, or 32-bit integer), numeric type (real or complex), and dimensionality (one-dimensional, two-dimensional, or multidimensional array). Many blocks can accept or output signals of any data or numeric type and dimensionality. Others impose restrictions on the attributes of the signals they can handle.

Simulink® defines signals as the outputs of dynamic systems represented by blocks in a Simulink diagram and by the diagram itself. The lines in a block diagram represent mathematical relationships among the signals defined by the block diagram. For example, a line connecting the output of block A to the input of block B indicates that the signal output by B depends on the signal output by A.

On the block diagram, signals are represented with lines that have an arrowhead. The source of the signal corresponds to the block that writes to the signal during evaluation of its block methods (equations). The destinations of the signal are blocks that read the signal during the evaluation of the block's methods (equations).

Note It is tempting but misleading to think of Simulink signals as traveling along the lines that connect blocks the way electrical signals travel along a telephone wire. This analogy is misleading because it suggests that a block diagram represents physical connections between blocks, which is not the case. Simulink signals are mathematical, not physical, entities and the lines in a block diagram represent mathematical, not physical, relationships among blocks.

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 "About Composite Signals" on page 9-2 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 "Introducing the Signal and Scope Manager" on page 8-33 for more information.

Naming Signals

You can give any signal a name. The syntactic requirements for a signal name vary depending on how the name will be used. The three most common cases are:

- The signal is named so that it can be resolved to a `Simulink.Signal` object. (See Simulink.Signal.) The signal name must then be a legal MATLAB® identifier. Such an identifier starts with an alphabetic character, followed by at most 63 alphanumeric or underscore characters.

- The signal has a name so the signal can be identified and referenced by name in a data log. (See "Logging Signals" on page 8-49.) Such a signal name can contain space and newline characters. These can improve readability but sometimes require special handling techniques, as described in "Handling Spaces and Newlines in Logged Signal Names" on page 8-58.

- The signal name exists only to clarify the diagram, and has no computational significance. Such a signal name can contain anything and never needs special handling.

To avoid any doubt about whether a signal name will serve all present and future purposes, make every signal name a legal MATLAB identifier. Otherwise, unexpected requirements may require going back and changing signal names to follow a more restrictive syntax.

Assigning a Signal Name

To assign a name to a signal, double-click the signal. An edit box appears next to the signal near where you double-clicked. Enter the desired name, then click somewhere outside the edit box. The signal now has the specified name, and a label showing that name appears at the location where you entered it. For a named multibranched signal, you can put a duplicate label on any branch of the signal by double-clicking the branch.

Another way to name a signal is to right-click the signal, choose **Signal Properties** from the Context menu, enter a name in the **Signal Name** field, then click **OK** or **Apply**. A label showing the name then appears on every branch of the signal. See "Signal Properties Dialog Box" for more information.

You can also use the API to set the name parameter of the port or line that represents the signal:

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

Changing a Signal Name

To change the name of a signal, click to set the cursor in any label that shows the name, than change the text as needed; or edit the name in the **Signal Properties > Signal Name** field. All labels automatically update to reflect the change.

To change the location of a label that displays a signal name, drag it with the mouse. You cannot drag a label away from its signal, but only to a different location adjacent to the signal.

Deleting a Signal Name
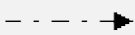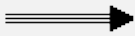
To delete a signal's name, leaving it nameless, delete all characters in the name, in any label on the signal or in the **Signal Properties > Signal Name** field. To delete a label without deleting the signal name, click near the edge of the label to select its surrounding box, then press Delete. The label disappears, but the signal name itself is unaffected.

Displaying Signal Values

As with creating signals, you can use either blocks or the Signal & Scope Manager to display the values of signals during a simulation. For example, you can use either the Scope block or the Signal & Scope Manager to graph time-varying signals on an oscilloscope-like display during simulation. See "Sinks" in the Simulink block reference for information on blocks that you can use to display signals in a model.

Signal Line Styles

Simulink uses a variety of line styles to display different types of signals in the model window. Assorted line styles help you to differentiate the signal types in Simulink diagrams. The signal types and their line styles are as follows:

Signal Type	Line Style	Description
Scalar and Nonscalar	——————▶	Simulink uses a thin, solid line to represent a diagram's scalar and nonscalar signals.
Nonscalar	━━━━━━▶	When the **Wide nonscalar lines** option is enabled, Simulink uses a thick, solid line to represent a diagram's nonscalar signals. See also "Using Muxes" on page 9-3.
Control	– · – · –▶	Simulink uses a thin, dash-dot line to represent a diagram's control signals.
Virtual Bus	═══════▶	Simulink uses a triple line with a solid core to represent a diagram's virtual signal buses. See "Using Buses" on page 9-5.
Nonvirtual Bus	┅┅┅┅┅▶	Simulink uses a triple line with a dotted core to represent a diagram's nonvirtual signal buses. See "Using Buses" on page 9-5.

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 8-76 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 8-13) and its **Show**

propagated signals property is on (see "Show propagated signals"), the label displays the names of the signals that make up the virtual signal.

Simulink creates a label for a signal when you assign it a name in the "Signal Properties Dialog Box". You can change the signal's name by editing its label on the block diagram. To edit the label, left-click the label. Simulink replaces the label with an edit field. Edit the name in the edit field, then click outside the label to apply the change.

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

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

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

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

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

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

To edit an existing signal label, select it:

• To replace the label, click the label, double-click or drag the cursor to select the entire label, then enter the new label.

- To insert characters, click between two characters to position the insertion point, then insert text.

- To replace characters, drag the mouse to select a range of text to replace, then enter the new text.

To delete all occurrences of a signal label, delete all the characters in the label. When you click outside the label, the labels are deleted. To delete a single occurrence of the label, hold down the **Shift** key while you select the label, then press the **Delete** or **Backspace** key.

To change the font of a signal label, select the signal, choose **Font** from the **Format** menu, then select a font from the **Set Font** dialog box.

Signal Data Types

Data type refers to the format used to represent signal values internally. The data type of Simulink signals is double by default. However, you can create signals of other data types. Simulink supports the same range of data types as MATLAB. See "Working with Data Types" on page 10-2 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 Fundamentals 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 8-21 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.
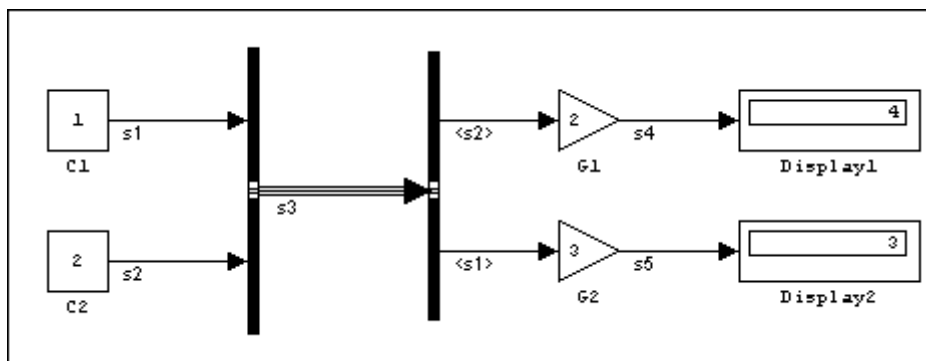
The signals driving Gain blocks G1 and G2 are virtual signals corresponding to signals s2 and s1, respectively. Simulink determines this automatically whenever you update or simulate the model.

Displaying the Nonvirtual Components of Virtual Signals

The **Show propagated signals** option (see "Signal Properties Dialog Box")
displays the nonvirtual signals represented by virtual signals in the labels
of the virtual signals.
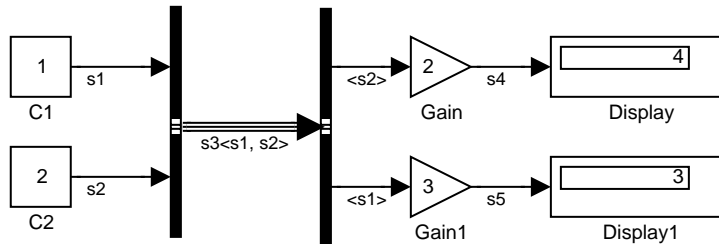
When you change the name of a nonvirtual signal, Simulink immediately
updates the labels of all virtual signals that represent the nonvirtual signal
and whose **Show propagated signals** is on, except if the path from the
nonvirtual signal to the virtual signal includes an unresolved reference to
a library block. In such cases, to avoid time-consuming library reference
resolutions while you are editing a block diagram, Simulink defers updating
the virtual signal's label until you update the model's block diagram either
directly (e.g., by pressing **Ctrl+D**) or by simulating the model.

Note Virtual signals can represent virtual as well as nonvirtual signals. For
example, you can use a Bus Creator block to combine multiple virtual and
nonvirtual signals into a single virtual signal. If during signal propagation,
Simulink determines that a component of a virtual signal is itself virtual,
Simulink uses signal propagation to determine the nonvirtual components of
the virtual component. This process continues until Simulink has determined
all nonvirtual components of a virtual signal.

To display the signal(s) represented by a virtual signal, 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).

Control Signals

A *control signal* is a signal used by one block to initiate execution of another block, e.g., a function-call or action subsystem. When you update or start simulation of a block diagram, Simulink uses a dash-dot pattern to redraw lines representing the diagram's control signals as illustrated in the following example.
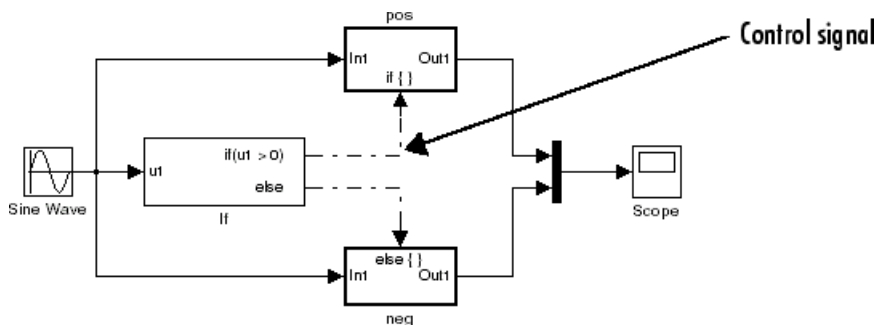
Signal Glossary

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 10-2 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 9-5.
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 8-70 for more information.
Vector	One-dimensional signal array.
Virtual signal	Signal that represents another signal or set of signals.
Width	Size of a vector signal.

Validating Signal Connections

Many Simulink® blocks have limitations on the types of signals they can accept. Before simulating a model, Simulink checks all blocks to ensure that they can accommodate the types of signals output by the ports to which they are connected. If any incompatibilities exist, Simulink reports an error and terminates the simulation.

To detect signal compatibility errors before running a simulation, choose **Update Diagram** from the Simulink **Edit** menu. Simulink reports any invalid connections found in the process of updating the diagram.

Displaying Signal Sources and Destinations

In this section...

About Signal Highlighting

You can highlight a signal and its source or destination block(s), then remove the highlighting once it has served its purpose. Signal highlighting crosses subsystem boundaries, allowing you to trace a signal across multiple subsystem levels. Highlighting does not cross the boundary into or out of a referenced model. If a signal is composite, all source or destination blocks are highlighted. (See Chapter 9, "Using Composite Signals".)

Highlighting Signal Sources

To display a signal's source block(s), choose **Highlight to Source** from the signal's context menu. Simulink® highlights:

- All branches of the signal anywhere in the model

- All virtual blocks through which the signal passes

- The nonvirtual block(s) that write the value of the signal
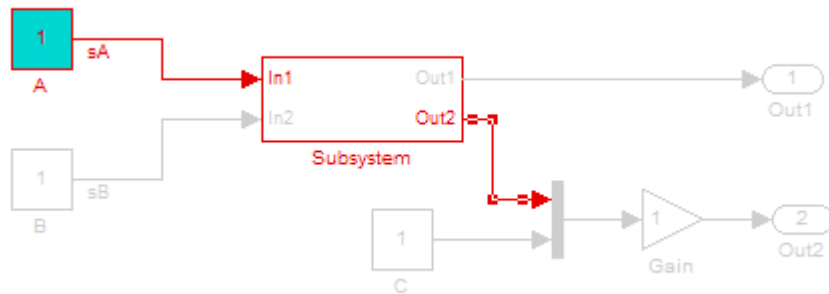
Highlighting Signal Destinations

To display a signal's destination blocks, choose **Highlight to Destination** from the signal's context menu. Simulink highlights:

- All branches of the signal anywhere in the model

- All virtual blocks through which the signal passes

- The nonvirtual block(s) that read the value of the signal
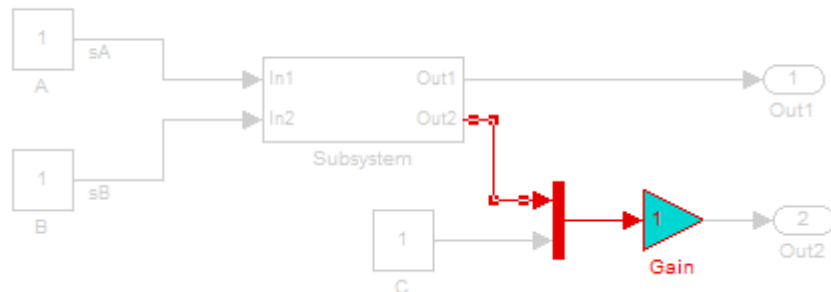
Removing Highlighting

To remove all highlighting, choose **Remove Highlighting** from the model's context menu, or choose **View > Remove Highlighting**.

Signal Highlighting and Library Blocks

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.

Determining Output Signal Dimensions

In this section...
"About Signal Dimensions" on page 8-21
"Determining the Output Dimensions of Source Blocks" on page 8-21
"Determining the Output Dimensions of Nonsource Blocks" on page 8-22
"Signal and Parameter Dimension Rules" on page 8-22
"Scalar Expansion of Inputs and Parameters" on page 8-24

About Signal Dimensions

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

Determining the Output Dimensions of Source Blocks

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

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

Constant Value	Interpret Vector Parameters as 1-D	Output
scalar	off	one-element array
scalar	on	one-element array
1-by-N matrix	off	1-by-N matrix
1-by-N matrix	on	N-element vector
N-by-1 matrix	off	N-by-1 matrix
N-by-1 matrix	on	N-element vector
M-by-N matrix	off	M-by-N matrix
M-by-N matrix	on	M-by-N matrix

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

Determining the Output Dimensions of Nonsource Blocks

If a block has inputs, the dimensions of its outputs are, after scalar expansion, the same as those of its inputs. (All inputs must have the same dimensions, as discussed in "Signal and Parameter Dimension Rules" on page 8-22).

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 8-24) 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 8-25) thus preserving the general rule.

- If an input is a vector, the corresponding parameter can be either an N-by-1 or a 1-by-N matrix. In this case, Simulink applies the N matrix elements to the corresponding elements of the input vector. This exception allows use of MATLAB® row or column vectors, which are actually 1-by-N or N-by-1 matrices, respectively, to specify parameters that apply to vector inputs.

Vector or Matrix Input Conversion Rules

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

- If a vector signal is connected to an input that requires a matrix, Simulink converts the vector to a one-row or one-column matrix.

- If a one-column or one-row matrix is connected to an input that requires a vector, Simulink converts the matrix to a vector.

- If the inputs to a block consist of a mixture of vectors and matrices and the matrix inputs all have one column or one row, Simulink converts the vectors to matrices having one column or one row, respectively.

> **Note** You can configure Simulink to display a warning or error message if a vector or matrix conversion occurs during a simulation. See "Vector/matrix block input conversion" for more information.

Scalar Expansion of Inputs and Parameters

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

Scalar Expansion of Inputs

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

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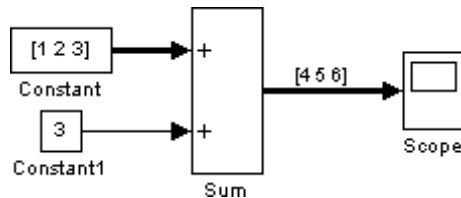
When a block's output is a function of a parameter and the parameter is nonscalar, Simulink expands a scalar input to match the dimensions of the parameter. For example, Simulink expands a scalar input to a Gain block to match the dimensions of a nonscalar gain parameter.

Scalar Expansion of Parameters

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

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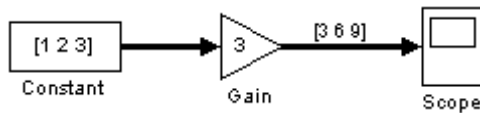
Checking Signal Ranges

In this section...
"About Signal Range Checking" on page 8-26
"Blocks That Allow Signal Range Specification" on page 8-26
"Specifying Ranges for Signals" on page 8-27
"Checking for Signal Range Errors" on page 8-28

About Signal Range Checking

Many Simulink® blocks allow you to specify a range of valid values for their output signals. Simulink provides a diagnostic that you can enable to detect when blocks generate signals that exceed their specified ranges during simulation. See the sections that follow for more information.

Blocks That Allow Signal Range Specification

The following blocks allow you to specify ranges for their output signals:

- Abs
- Constant
- Data Store Memory
- Data Type Conversion
- Difference
- Discrete Derivative
- Discrete-Time Integrator
- Gain
- Inport
- Interpolation Using Prelookup
- Lookup Table
- Lookup Table (2-D)

- Math Function

- MinMax

- Multiport Switch

- Outport

- Product, Divide, Product of Elements

- Relay

- Repeating Sequence Interpolated

- Repeating Sequence Stair

- Saturation

- Saturation Dynamic

- Signal Specification

- Sum, Add, Subtract, Sum of Elements

- Switch

See "Blocks — Alphabetical List" in the *Simulink Reference* for more information about these blocks and their parameters.

Specifying Ranges for Signals

In general, use the **Output minimum** and **Output maximum** parameters that appear on a block parameter dialog box to specify a range of valid values for the block output signal. Exceptions include the Data Store Memory, Inport, Outport, and Signal Specification blocks, for which you use their **Minimum** and **Maximum** parameters to specify a signal range. See "Blocks That Allow Signal Range Specification" on page 8-26 for a list of applicable blocks.

When specifying minimum and maximum values that constitute a range, enter only expressions that evaluate to a scalar, real number with `double` data type. The default value, `[]`, is equivalent to `-Inf` for the minimum value and `Inf` for the maximum value. The scalar values that you specify are subject to expansion, for example, when the block inputs are nonscalar or bus signals (see "Scalar Expansion of Inputs and Parameters" on page 8-24).

> **Note** You cannot specify the minimum or maximum value as NaN.

Specifying Ranges for Complex Numbers

When you specify an **Output minimum** and/or **Output maximum** for a signal that is a complex number, the specified minimum and maximum values apply separately to the real part and to the imaginary part of the complex number. If the value of either part of the number is less than the minimum, or greater than the maximum, the complex number is outside the specified range. No range checking occurs against any combination of the real and imaginary parts, such as (`sqrt(a^2+b^2)`)

Checking for Signal Range Errors

Simulink provides a diagnostic named **Simulation range checking**, which you can enable to detect when signals exceed their specified ranges during simulation. When enabled, Simulink compares the signal values that a block outputs with both the specified range (see "Specifying Ranges for Signals" on page 8-27) and the block data type. That is, Simulink performs the following check:

    DataTypeMin ≤ MinValue ≤ VALUE ≤ MaxValue ≤ DataTypeMax

where

- `DataTypeMin` is the minimum value representable by the block data type.

- `MinValue` is the minimum value the block should output, specified by, e.g., **Output minimum**.

- `VALUE` is the signal value that the block outputs.

- `MaxValue` is the maximum value the block should output, specified by, e.g., **Output maximum**.

- `DataTypeMax` is the maximum value representable by the block data type.

Note It is possible to overspecify how a block handles signals that exceed particular ranges. For example, you can specify values (other than the default values) for both signal range parameters and enable the **Saturate on integer overflow** parameter. In this case, Simulink displays a warning message that advises you to disable the **Saturate on integer overflow** parameter.

Enabling Simulation Range Checking

To enable the **Simulation range checking** diagnostic:

1 In your model window, select **Simulation > Configuration Parameters**.

Simulink displays the Configuration Parameters dialog box.

2 In the **Select** tree on the left side of the Configuration Parameters dialog box, click the **Diagnostics > Data Validity** category. On the right side under **Signals**, set the **Simulation range checking** diagnostic to error or warning.

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
Simulation range checking:	error		
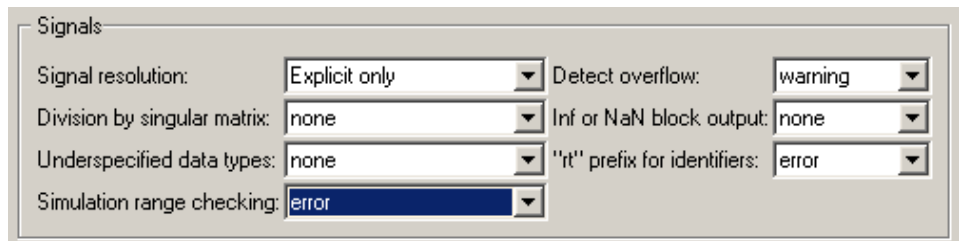
3 Click **OK** to apply your changes and close the Configuration Parameters dialog box.

See "Simulation range checking" in the *Simulink Graphical User Interface* documentation for more information.

Simulating Models with Simulation Range Checking

To check for signal range errors or warnings:

1 Enable the **Simulation range checking** diagnostic for your model (see "Enabling Simulation Range Checking" on page 8-29).

2 In your model window, select **Simulation > Start** to simulate the model.

Simulink simulates your model and performs signal range checking. If a signal exceeds its specified range when the **Simulation range checking** diagnostic specifies error, Simulink stops the simulation and displays an error message:
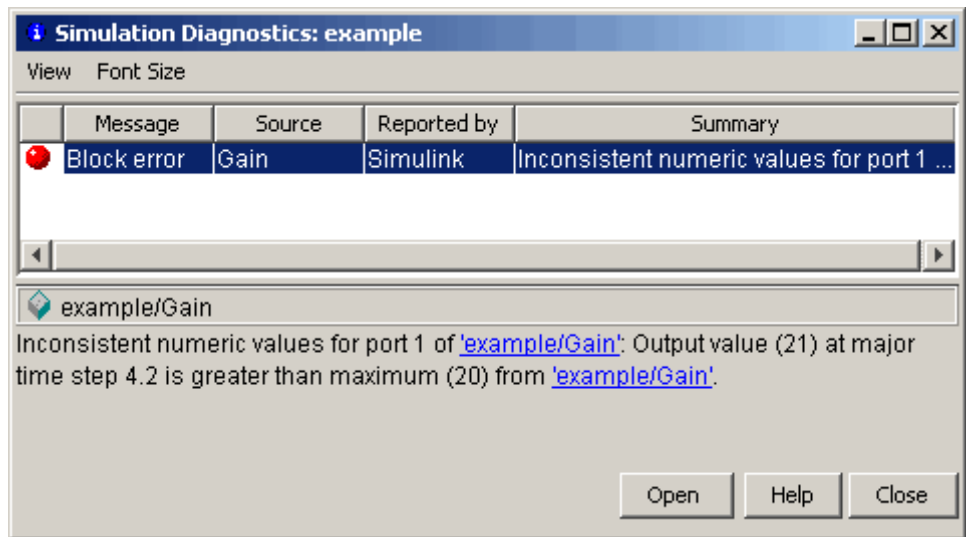
Otherwise, if a signal exceeds its specified range when the **Simulation range checking** diagnostic specifies warning, Simulink displays a warning message in the MATLAB® Command Window:

```
Warning: Inconsistent numeric values for port 1 of 'example/Gain': Out
(21) at major time step 4.2 is greater than maximum (20) from 'example
```

Each message identifies the block whose output signal exceeds its specified range, and the time step at which this violation occurs.

Signal Range Propagation for Virtual Blocks

Some virtual blocks (see "Virtual Blocks" on page 6-2) allow you to specify ranges for their output signals, for example, the Inport and Outport blocks. When the **Simulation range checking** diagnostic is enabled for a model that contains such blocks, the signal range of the virtual block propagates backward to the first instance of a nonvirtual block whose output signal it receives. If the nonvirtual block specifies different values for its own range, Simulink performs signal range checking with the *tightest* range possible. That is, Simulink checks the signal using the larger minimum value and the smaller maximum value.

For example, consider the following model:
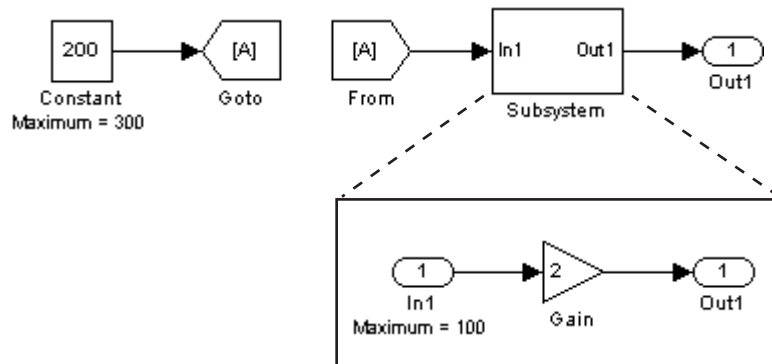
In this model, the Constant block specifies its **Output maximum** parameter as 300, and that of the Inport block is set to 100. Suppose you enable the **Simulation range checking** diagnostic and simulate the model. The Inport block back propagates its maximum value to the nonvirtual block that precedes it, i.e., the Constant block. Simulink then uses the smaller of the two maximum values to check the signal that the Constant block outputs. Because the Constant block outputs a signal whose value (200) exceeds the tightest range, Simulink displays the following error message:
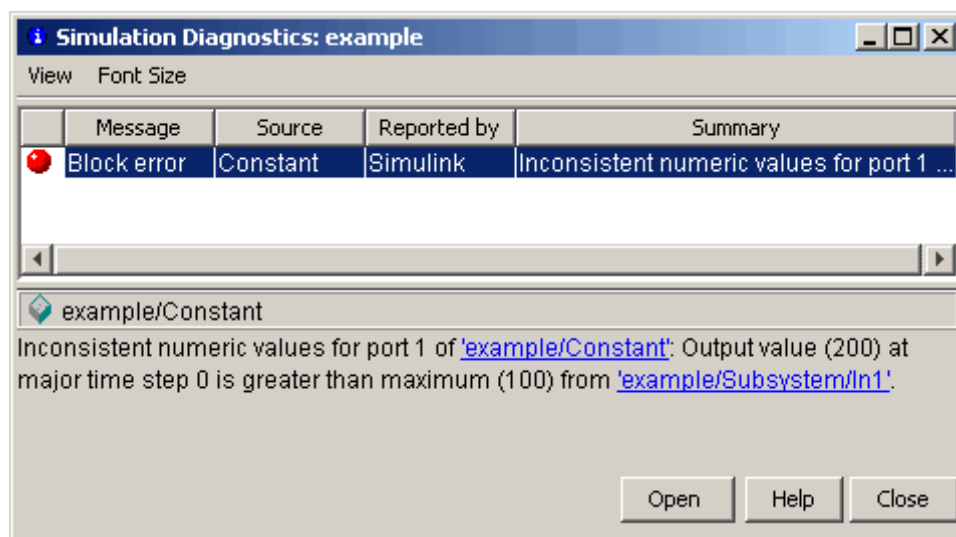
Introducing the Signal and Scope Manager

In this section...

What is the Signal & Scope Manager?

The Signal & Scope Manager is a user interface to the Signal Viewers and Generator objects. From the Signal and Scope Manager you manage all signal generators and viewers from a central place.

Note The Signal and Scope Manager requires that you have Java™ enabled when you start MATLAB®. This is the default.

What are Viewer and Generator Objects?

The small icons identifying a viewer or generator are called Viewer and Generator Objects. These objects are not the same as scope or signal blocks. Objects are managed by the Signal and Scope Manager, and are placed on signals. Blocks are dragged from the Library browser and are not managed by the Signal and Scope manager.
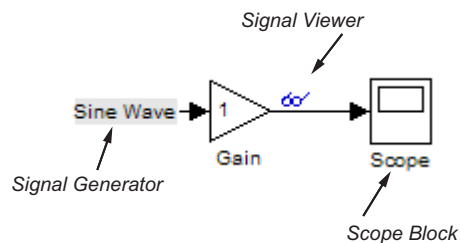
Displaying the Signal and Scope Manager User Interface

Access the Signal and Scope Manager from the model editor's **Tools** menu.

Alternatively, right click within your model and select **Signal & Scope Manager** from the context menu.

Understanding the Signal and Scope Manager User Interface
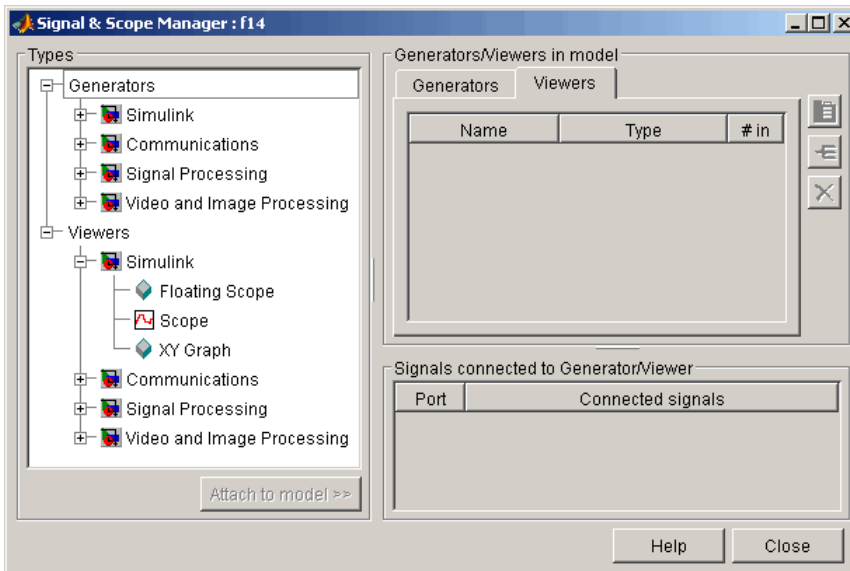
The Signal and Scope manager user interface is comprised of three panes:

- **Types**. Selects the viewer or generator to attach to your model. For more information, see "Types Pane" on page 8-35.
- **Generators/Viewers in model**. Selects signal sources and viewers for your model.

  For more information on sources, see "Generators Tab" on page 8-35.

  For more information on viewers, see "Viewers Tab" on page 8-36.

- **Signals connected to Generator/Viewer**. Manages the connections to the generators and viewers present in your model. For more information, see "Signals connected to Generator/Viewer Pane" on page 8-38.

Types Pane

The **Types** pane shows the generators and viewers associated with the products installed on your system. Expand a products node list to show all the generators and viewers available to you.

Note The Simulink® Scope displayed in the Signal and Scope Manager Types pane is not the same as the Simulink Scope Block. For an explanation of the differences, see "How Scope Blocks and Signal Viewers Differ" on page 15-3.

Generators Tab

The **Generators** tab displays a table listing the generators associated with your model.
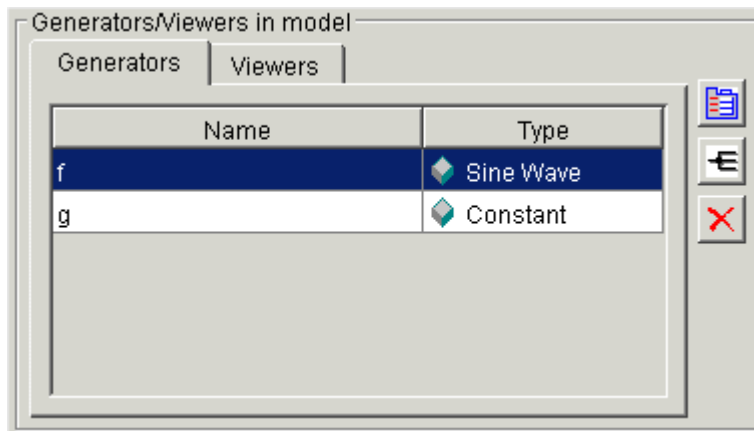
Each row corresponds to a generator. The columns specify each generator's name and type.

Viewers Tab

The **Viewers** tab displays a table listing the viewers present in your model.
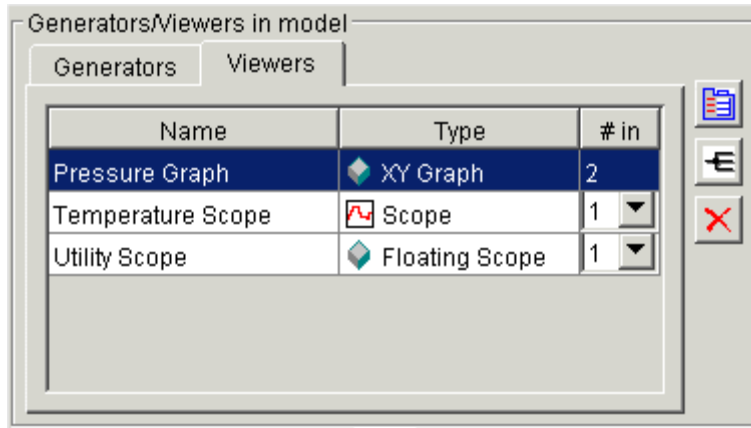
Each row corresponds to a viewer. The columns specify each viewer's name, type, and number of inputs. If a viewer accepts a variable number of inputs, the **#in** entry for the viewer contains a pull-down list that displays the range of inputs that the viewer can accept. To change the number of inputs accepted by the viewer, pull down the list and select the desired value.

Edit Buttons

Use these buttons to manage generators and viewers after you have selected them in the Generators or Viewers table:

Button	Description
	Opens the parameter dialog box for the selected generator or viewer.
	From the parameter dialog you view and change object parameters.
	See "Scope Viewer Parameters Dialog Box" on page 15-12 for more information.
	Opens the Signal Selector for the selected generator or viewer.
	You use the Signal Selector to connect and disconnect generators and viewers.
	See "The Signal Selector" on page 8-44 for information on the signal selector.
	Deletes the selected generator or viewer.

Signals connected to Generator/Viewer Pane

This table lists the signals connected to the generator or viewer selected in the Generators/Viewers control group of the Signal and Scope Manager.
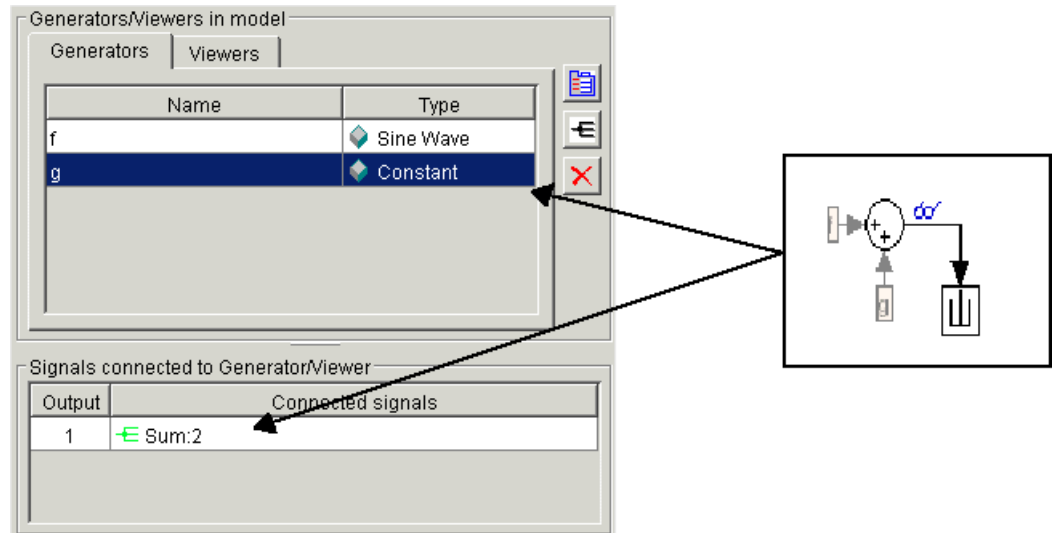
This graphic illustrates the table display when two generators are connected to a sum block. The Viewers display works in the same way.

Clicking on the name of a generator displays the connected signals. For instance, the constant is shown connected to the second input of the sum block.

Connection Menu

Selecting a connection in the **Signals connected to Generator/Viewer** table and pressing the right button on your mouse displays a context menu. From this context menu you can:

- Open the Properties dialog

- Highlight the connections in your block diagram

- Open the Signal Selector

Using the Signal and Scope Manager

In this section...
"Introduction" on page 8-39
"Attaching a New Viewer or Generator" on page 8-39
"Creating a Multiple Axes Viewer" on page 8-40
"Adding Additional Signals to an Existing Viewer" on page 8-41
"Viewing Test Point Data" on page 8-41
"Adding Custom Viewers and Generators" on page 8-42

Introduction

This section shows you how to use the Signal and Scope Manager to perform some basic Viewer and Generator object tasks.

If you are not familiar with the Signal and Scope manager, or Viewer or Generator objects, or if you do not know how to display the Signal and Scope manager, see "Introducing the Signal and Scope Manager" on page 8-33.

To learn how to use and adjust the viewers you have created, see "The Scope Viewer Toolbar" on page 15-11.

Attaching a New Viewer or Generator

To connect a new viewer or generator to a signal in the currently selected model:

1 Display the Signal and Scope manager.

2 Select a viewer or generator from the **Types** pane.

3 Click **Attach to Model**.

4 Click the **Signal Selector** button to display the Signal Selector dialog.

For more information, see "The Signal Selector" on page 8-44.

5 Select the signals to be displayed by this viewer, and close the dialog.

Tip To display a viewer that has been attached, double click on the viewer of interest in the Viewers pane.

Creating a Multiple Axes Viewer

To create a viewer with more than one axes:
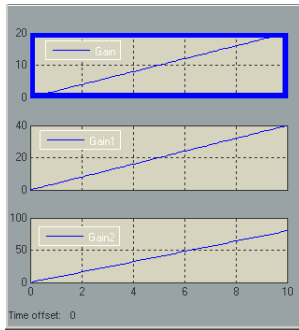
1 Display the Signal and Scope manager.

2 Select a viewer from the **Types** pane.

3 Click **Attach to Model**.

4 Click the #in pulldown, and select the total number of axes for the graph.

5 Navigate to the **Signals connected to Generator/Viewer** pane and select **Axes 1**.

6 Click the **Signal Selector** button to display the Signal Selector dialog.

For more information, see "The Signal Selector" on page 8-44.

7 Select the signals to add to this axis, and close the dialog.

8 Select the next axes, and repeat steps 6 and 7. Continue in this way until signals have been added to all axes.

Tip

- Click on the Scope Viewer icon to display the scope.

- Run the simulation after adding the new signals to make them visible.

Adding Additional Signals to an Existing Viewer

To add signals to a viewer you have already created:

1 Display the Signal and Scope manager.

2 Navigate to the **Viewers** pane, and select the scope to which you will add signals.

3 Click the **Signal Selector** button to display the Signal Selector dialog.

For more information, see "The Signal Selector" on page 8-44.

4 Select the signals to add to this viewer, and close the dialog.

Tip Run the simulation after adding the new signals to make them visible.

Viewing Test Point Data

You can use the Signal and Scope Manager to view any signal that is defined as a test point in a submodel. A test point is a signal that is guaranteed to be observable when using a signal viewer in a model.

For more information, see "Working with Test Points" on page 8-70 and Chapter 5, "Referencing a Model".

Adding Custom Viewers and Generators

You can add custom signal viewers or generators so that they appear in the Signal and Scope Manager. The following procedure assumes that you have a custom viewer named `newviewer` that you want to add.

Note If the viewer is a compound viewer, such as a subsystem with multiple blocks, make the top-level subsystem atomic.

---

1 Open a new Simulink® library.

   For example, open the Simulink browser and select **File > New > Library**.

2 Save the library.

   For example, save it as `newlib`.

3 In the MATLAB® Command Window, set the library type.

   For example, use this command to set the library type of `newviewer` to `viewer`,

   ```
   set_param('newlib','LibraryType','SSMgrViewerLibrary')
   ```

   To set library type for generators, use the type `'SSMgrGenLibrary'` as in this example:

   ```
   set_param('newlib','LibraryType','SSMgrGenLibrary')
   ```

4 Set the display name of the library, as in this example:.

   ```
   set_param('newlib','SSMgrDisplayString','My Custom Library')
   ```

5 Add the viewer or generator to the library.

6 Set the `iotype` of the viewer, as in this example:

```
set_param('newlib/newviewer','iotype','viewer')
```

7 Save the library `newlib`.

Select **File > Save**.

8 Using the MATLAB editor, create a file named `sl_customization.m`. In this file, enter a directive to incorporate the new library as a viewer library.

For example, to save `newlib` as a viewer library, add the following lines:

```
function sl_customization(cm)
cm.addSigScopeMgrViewerLibrary('newlib')
%end function
```

To add a library as a generator library, add a line like the following:

```
cm.addSigScopeMgrGeneratorLibrary('newlib')
```

9 Add a corresponding `cm.addSigScope` line for each viewer or generator library you want to add.

10 Save the `sl_customization.m` file on your MATLAB path. Edit this file to add new viewer or generator libraries.

11 To see the new custom libraries, restart MATLAB and start the Signal and Scope Manager.

The Signal Selector

About the Signal Selector

The Signal Selector allows you to connect a generator or viewer object (see "Introducing the Signal and Scope Manager" on page 8-33) or the Floating Scope to block inputs and outputs. It appears when you click the **Signal selection** button for a generator or viewer object in the Signal & Scope Manager or on the toolbar of the Floating Scope's window.

The Signal Selector that appears when you click the **Signal selection** button applies only to the currently selected generator or viewer object (or the Floating Scope). If you want to connect blocks to another generator or viewer object, you must select the object in the Signal & Scope Manager and launch another instance of the Signal Selector. The object used to launch a particular instance of the Signal Selector is called that instance's owner.
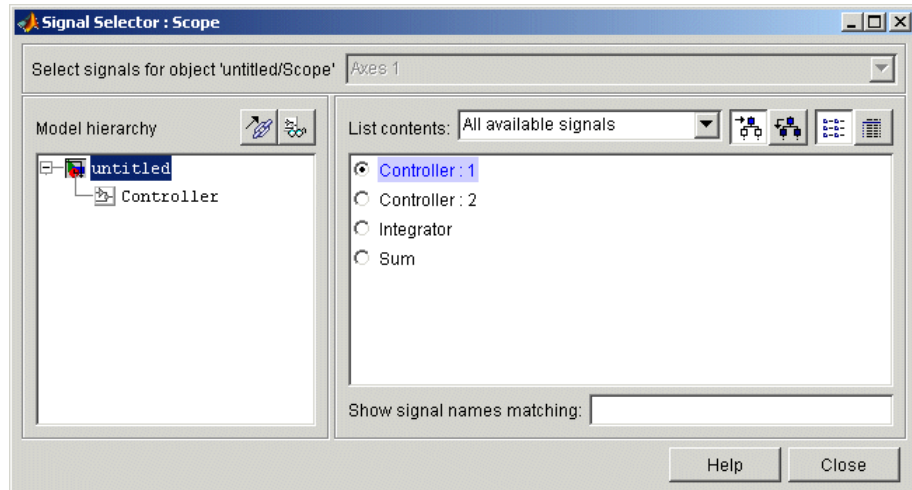
Port/Axis Selector

This list box allows you to select the owner output port (in the case of signal generators) or display axis (in the case of signal viewers) to which you want to connect blocks in your model.
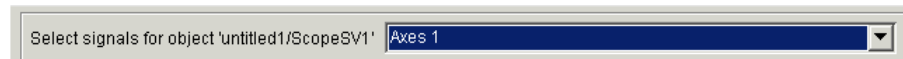
The list box is enabled only if the signal generator has multiple outputs or the signal viewer has multiple axes.

Model Hierarchy

This tree-structured list lets you select any subsystem in your model.
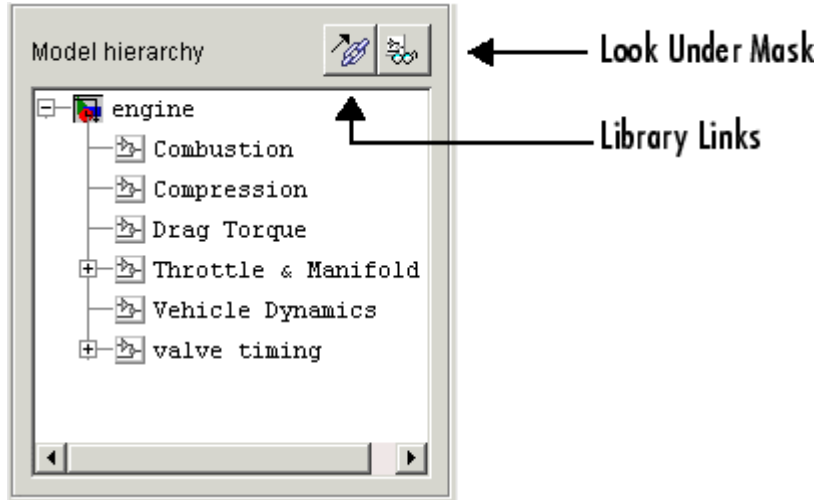
Selecting a subsystem causes the adjacent port list to display the ports available for connection in the selected subsystem. To display subsystems included as library links in your model, click the **Library Links** button at the top of the **Model hierarchy** control. To display subsystems contained by masked subsystems, click the **Look Under Masks** button at the top of the panel.

Inputs/Signals List

The contents of this panel displays input ports available for connection to the Signal Selector's owner if the owner is a signal generator or signals available for connection to the owner if the owner is a signal viewer.

If the Signal Selector's owner is a signal generator, the inputs/signals list by default lists each input port in the system selected in the model hierarchy tree that is either unconnected or connected to a signal generator.
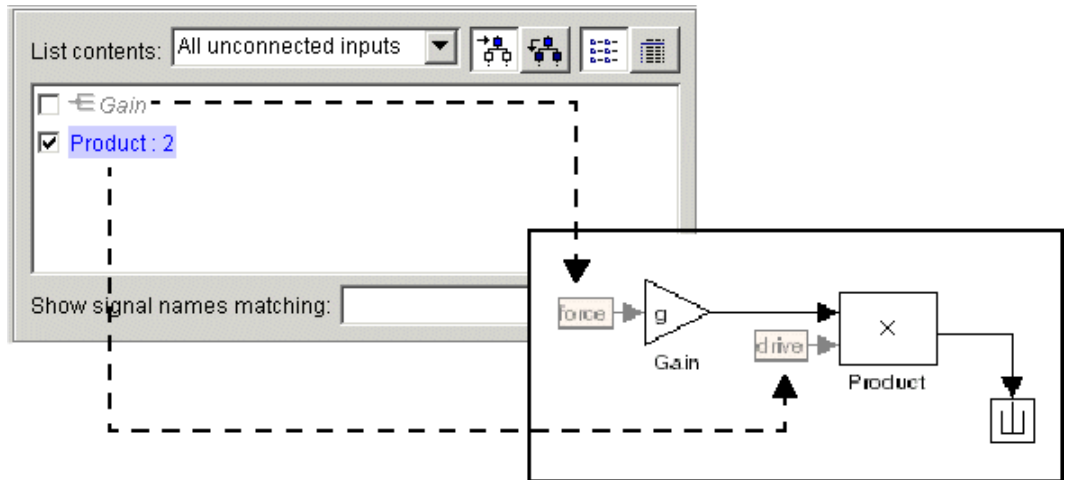
The label for each entry indicates the name of the block of which the port is an input. If the block has more than one input, the label indicates the number of the displayed port. A greyed label indicates that the port is connected to a signal generator other than the Signal Selectors' owner. Selecting 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

In this section...

About Signal Logging

Logging signals refers to the process of saving signal values to the MATLAB® workspace during simulation for later retrieval and postprocessing. Simulink® allows you to log a signal by

- Connecting the signal to a To Workspace block, Scope block, or viewer.

  This method allows you to document in the diagram itself the workspace variables used to store signal data. Results are visible during simulation. Be aware that Scopes store data and can be memory intensive.

- Connecting the signal to a root-level Outport block.

  This method reduces diagram clutter by eliminating the need to use Scope blocks to log signals. Data is only available when simulation is paused or completed.

- Setting the signal's signal logging properties.

  This method eliminates the need to add blocks. Data is only available when simulation is paused or completed.

All of these methods allow you to specify the names of the workspace variables used to save signal data and to limit the amount of data logged for a particular signal.

See *Simulink Reference* for the To Workspace and Outport blocks for information on using these blocks to log signal data. See the documentation of the `sim` command for some data logging capabilities that are available only for programmatic simulation.

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

Multidimensional signals are supported.

Bus hierarchy is supported. The hierarchy of a bus signal is preserved in the `logsout` object. See Chapter 9, "Using Composite Signals" for details about muxes and buses.

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").

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 8-70 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"). 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 ($\tilde{\uparrow}$) 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. To specify a log name for a signal, select Custom from the **Logging name** list on the signal's **Signal Properties** dialog box and enter the custom name in the adjacent text field.

If you do not specify a custom logging name, Simulink uses the signal name, or if there is no name, Simulink generates a default name that is composed of the block name and port number. For example, if the block name is MyBlock and the signal being logged is the first output of this block, Simulink generates the following name: SL_MyBlock1.

Limiting the Data Logged for a Signal

The **Data** panel of the **Signal Properties** dialog box lets you limit the amount of data logged for a signal. For example, you can specify the maximum amount of data to be logged for a signal or a decimation factor that causes Simulink to skip a specified number of time steps before logging a signal value. See "Data" for more information.

Logging Referenced Model Signals

You can log any signal that is defined as a test point in a referenced model. For information about test points, see "Designating a Signal as a Test Point" on page 8-70. For information about referenced models, see Chapter 5, "Referencing a Model".

To log test pointed signals in referenced models, select the Model block and then select **Log referenced signals** from the model editor's **Edit** menu or from the block's context menu.

The **Model Reference Signal Logging** dialog box appears.
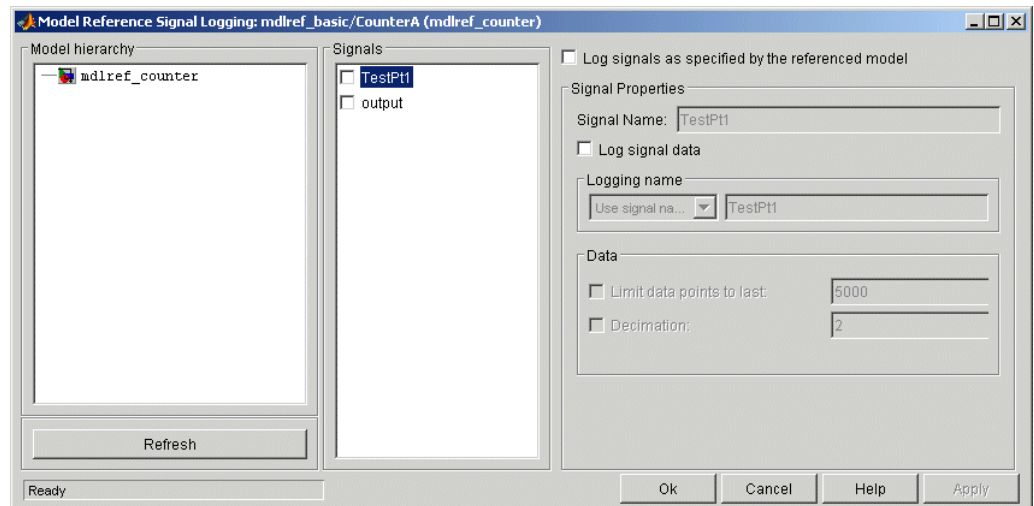
The dialog box contains the following panes and controls.

Model Hierarchy

This pane displays the contents of the referenced model as a tree control with expandable nodes. The top-level node represents the referenced model. Expanding this node displays the subsystems that the referenced model contains and any models that it itself references. Expanding a subsystem node displays the subsystems that it contains and the models that it references.

Refresh Button

Refreshes the dialog box to reflect changes in the model hierarchy.

Signals

This pane displays the test points of the model or subsystem selected in the **Model Hierarchy** pane (see "Working with Test Points" on page 8-70). Check the check box next to a test point's name to specify that it should be logged.

Log signals as specified by the referenced model

Checking this check box causes Simulink to log the signals that the referenced model specifies should be logged.

Signal Properties

This pane is enabled if **Log signals as specified by the referenced model** is not selected. In this case, the controls on this pane allow you to specify the signal logging properties of the signal selected in the **Signals** pane. The values that you specify override for this instance of the referenced model those specified by the model itself. The controls correspond to the controls of the same name on the **Signal Properties** dialog box. See "Signal Properties Dialog Box" for information on how to use them.

Viewing Logged Signal Data

To view logged signal data, either check 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 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") 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.

The signal logs for particular model elements are contained in the objects in the following table. The `Simulink.ModelDataLogs` object is the container for these objects.

Modeling Element	Signal Data Object
Signal in this model	`Simulink.Timeseries`
Model referenced by this model	`Simulink.ModelDataLogs`
Subsystem	`Simulink.SubsysDataLogs`
Bus, mux, vector concatenate	`Simulink.TsArray`
Scope	`Simulink.ScopeDataLogs`

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.
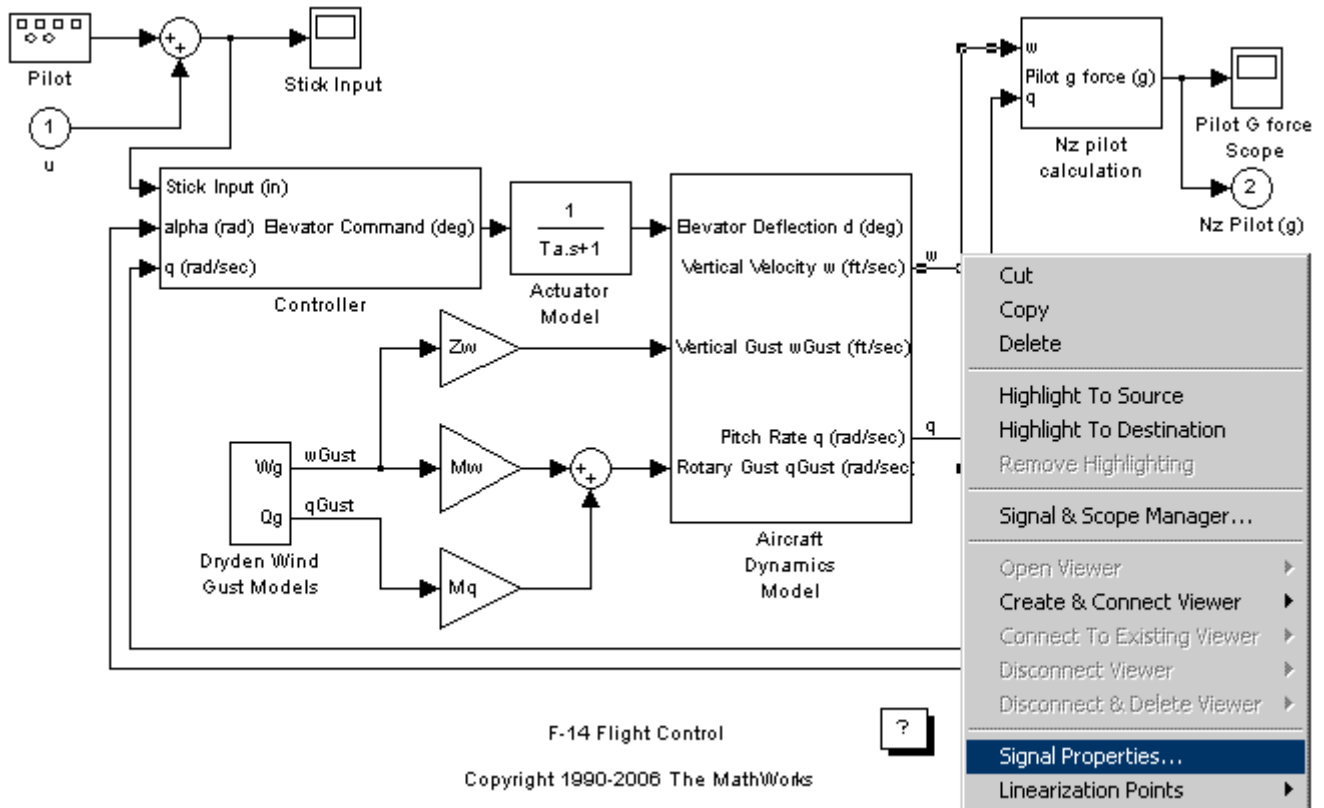
3 In the **Signal Properties** dialog box that opens, check the **Log signal data** option. Notice that the **Test point** option automatically becomes checked and the logging name initializes to the signal's name.
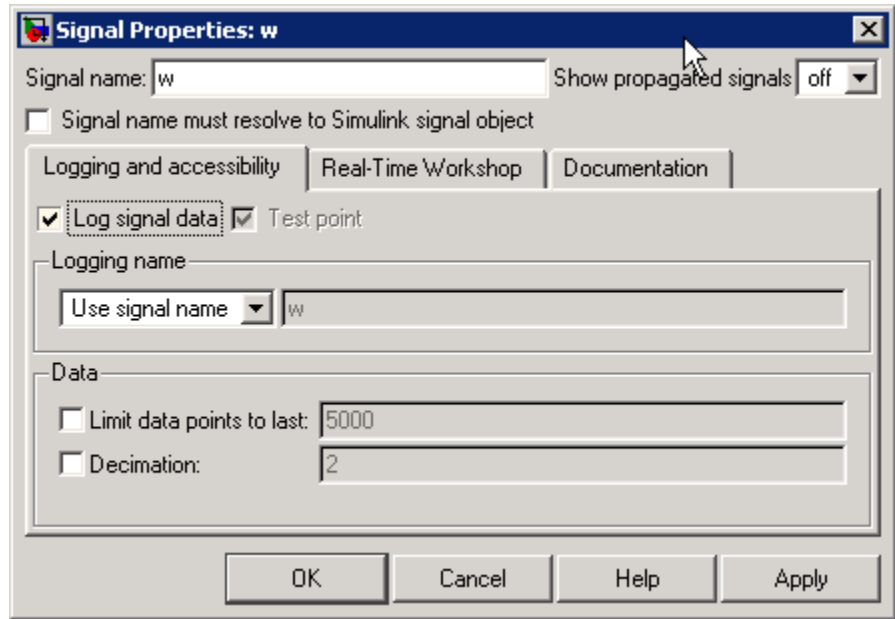
4 Click the **OK** button on the **Signal Properties** dialog box. The 'blue

antenna' icon $\overset{\circ}{\underset{|}{\top}}$ 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 in
the online MATLAB documentation).

Handling Spaces and Newlines in Logged Signal Names

Named signals that are used for signal logging but do not have an associated `Simulink.Signal` object can have names that include space and newline characters, as described in "Naming Signals" on page 8-4. Such names can improve the readability of a block diagram, but referencing them requires require special techniques when they appear in a data log. These techniques allow the MATLAB parser to process the names even though spaces and newlines are not legal in MATLAB identifiers.

Space and newline characters can also appear in a data log when an unnamed logged signal originates in a block whose name contains any spaces or newlines. The Simulink software constructs a name for the signal based on the block name, and the constructed name inherits the characters that the block name contains; these characters then appear in the data log. This situation often arises, because many blocks have spaces in their default names. One way to avoid it is to name the logged signal, thus avoiding the default name.

The following model includes a signal whose name contains a space, a signal whose name contains a newline, and an unnamed signal that originates in a block whose name contains a newline:
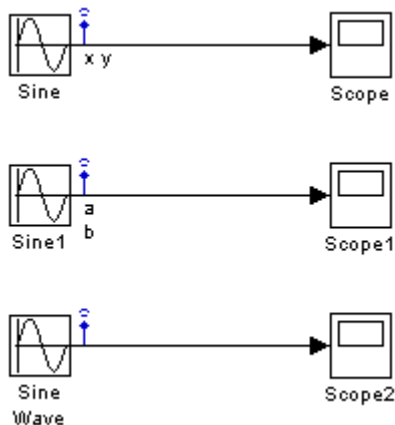
If you execute this model with data logging enabled in the Data Import/Export pane, accepting the default logging object name `logsout`, and then type

logsout in the MATLAB Command Window, MATLAB displays the following data log:

```
logsout =

Simulink.ModelDataLogs (model_name):
  Name                   Elements  Simulink Class

  ('x y')                    1       Timeseries
  ('a
b')                        1       Timeseries
  ('SL_Sine
Wave1')              1      Timeseries
```

You cannot access any of the Timeseries objects in this log using TAB name completion, or by typing the name to MATLAB, because the space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```
>> logsout.x y

??? logsout.x y
              |
Error: Unexpected MATLAB expression.
```

To reference a Timeseries object whose name contains a space, single-quote the element containing the space:

```
>> logsout.('x y')

          Name: 'x y'
     BlockPath: 'model_name/Sine'
     PortIndex: 1
    SignalName: 'x y'
    ParentName: 'x y'
      TimeInfo: [1x1 Simulink.TimeInfo]
          Time: [51x1 double]
          Data: [51x1 double]
```

To reference a Timeseries object whose name contains a newline, concatenate to construct the element containing the newline:

```
>> cr=sprintf('\n')
>> logsout.(['a' cr 'b'])
```

The same technique works when the newline derives from a block name used to generate a name for an unnamed signal:

```
>> logsout.(['SL_Sine' cr 'Wave1'])
```

For names with multiple spaces, newlines, or both, repeat and combine the two techniques as needed to specify the intended name to MATLAB. No analogous techniques exist for TAB name completion, which never works with names that contain space or newline characters.

Extracting Partial Data from a Running Simulation

Before a simulation ends, you can extract and write the currently logged signal data (from `Simulink.ModelDataLogs`) with the `set_param` `WriteDataLogs` command. The currently logged signal is the partial data logged between when the simulation started and when you request an extraction of the signal data. If you use this command during the simulation, Simulink writes the current logging variable values to the MATLAB workspace. If you use this command at the end of the simulation, Simulink writes the values from the last simulation to the MATLAB workspace.

To use this command, type the following at the MATLAB Command Window.

```
set_param(bdroot,'SimulationCommand','WriteDataLogs')
```

Initializing Signals and Discrete States

In this section...
"About Initialization" on page 8-62
"Using Block Parameters to Initialize Signals and Discrete States" on page 8-63
"Using Signal Objects to Initialize Signals and Discrete States" on page 8-63
"Using Signal Objects to Tune Initial Values" on page 8-64
"Example: Using a Signal Object to Initialize a Subsystem Output" on page 8-66
"Initialization Behavior Summary for Signal Objects" on page 8-67

About Initialization

Simulink® allows you to specify the initial values of signals and discrete states, i.e., the values of the signals and discrete states at 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.

Note When you specify a signal object for signal or discrete state initialization, or a variable as the value of a block parameter, the Simulink software resolves the name that you specify to an appropriate object or variable, as described in "Resolving Symbols" on page 3-69 and "Resolving Symbols" on page 3-69.

Using Block Parameters to Initialize Signals and Discrete States

For blocks that have an initial value or initial condition parameter, you can use that parameter to initialize a signal. For example, the following Block Parameters dialog box initializes the signal for a Unit Delay block with an initial condition of 0.5.
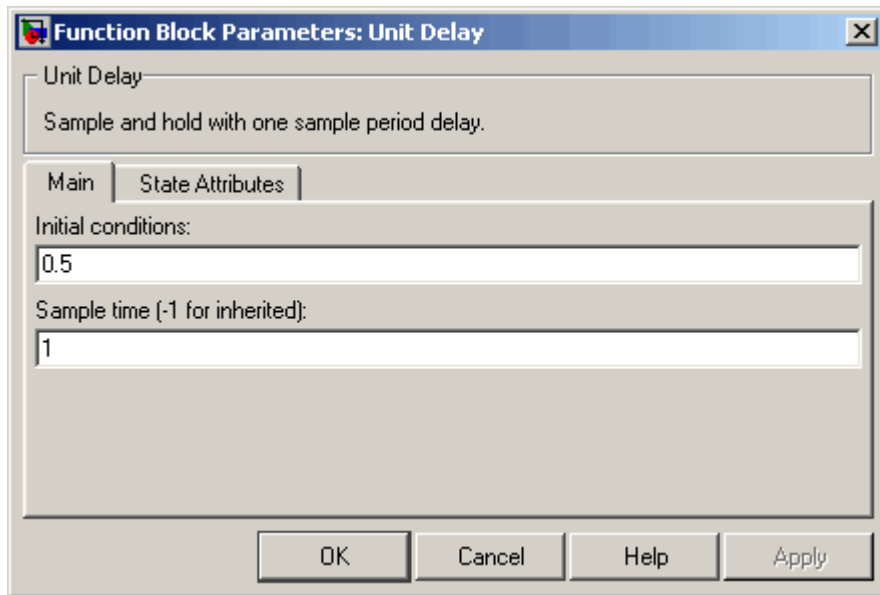
Using Signal Objects to Initialize Signals and Discrete States

To use a signal object to specify an initial value:

1 Create the signal object in the MATLAB® workspace, as explained in "Working with Data Objects" on page 10-27.

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 2-9 and "Changing the Values of Block Parameters During Simulation" on page 6-16.

Example: Using a Signal Object to Initialize a Subsystem Output

The following example shows a signal object specifying the initial output of an enabled subsystem.
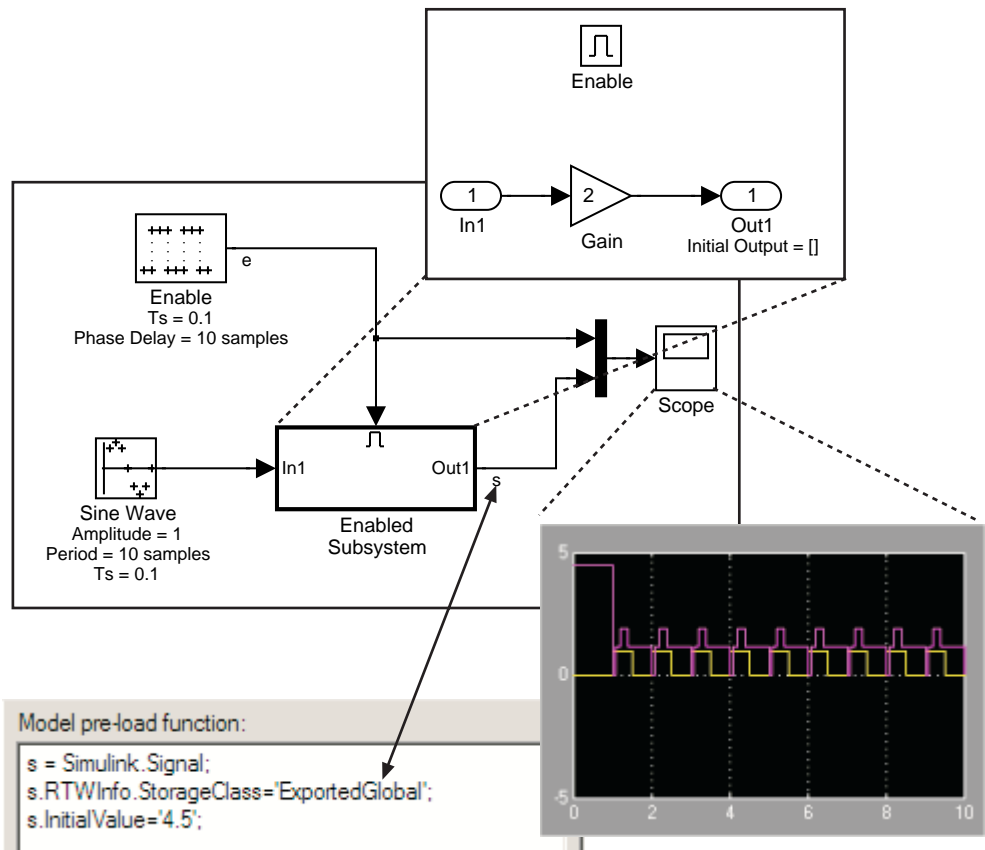
Signal s is initialized to 4.5. To avoid a consistency error, the initial value of the enabled subsystem's Outport block must be [ ] or 4.5.

If you need a signal object and its initial value setting to persist across Simulink sessions, see "Creating Persistent Data Objects" on page 10-37.

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 to specify values for the root inputs, the initial value is overwritten and may differ at each time step. Otherwise, the value remains constant.

Signal or Discrete State	Description	Behavior
X1	Unit Delay block — Block with a discrete state that has an initial condition	• 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 "Using 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

In this section...
"About Test Points" on page 8-70
"Designating a Signal as a Test Point" on page 8-70
"Displaying Test Point Indicators" on page 8-71

About Test Points

A *test point* is a signal that Simulink® guarantees to be observable when using a Floating Scope block in a model. Simulink allows you to designate any signal in a model as a test point.

Designating a signal as a test point exempts the signal from model optimizations, such as signal storage reuse (see "Signal storage reuse") and block reduction (see "Implement logic signals as boolean data (vs. double)"). These optimizations render signals inaccessible and hence unobservable during simulation.

Signals designated as test points will not have algebraic loops minimized, even if **Minimize algebraic loop occurrences** is selected (for more information about algebraic loops, see "Algebraic Loops" on page 2-31).

Test points are primarily intended for use when generating code from a model with Real-Time Workshop®. For more information about test points in the context of code generation, see "Declaring Test Points" in the Real-Time Workshop documentation.

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").

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.
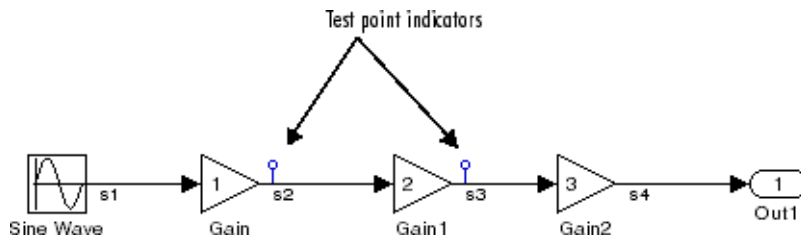
The appearance of the indicator changes slightly to indicate test points for which signal logging is enabled.
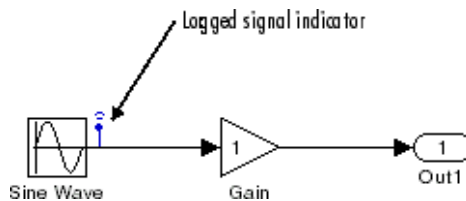
To turn display of test point indicators on or off, select **Port/Signal Displays > Test Point Indicators** from the Simulink **Format** menu.

Displaying Signal Properties

Port/Signal Displays Menu

A model window's **Format >Port/Signal Displays** menu offers the following options for displaying signal properties on the block diagram:

- Sample Time Colors (See "Displaying Sample Time Colors" on page 3-10)

- Linearization Indicators

- Port Data Types (See "Port Data Types" on page 8-73)

- Signal Dimensions (See "Signal Dimensions" on page 8-74)

- Storage Class

- Testpoint Indicators

- Signal Resolution Indicators (See "Signal Resolution Indicators" on page 8-75)

- Viewer Indicators

- Wide Nonscalar Lines (See "Wide Nonscalar Lines" on page 8-76)

Port Data Types

Displays the data type of a signal next to the output port that emits the signal.
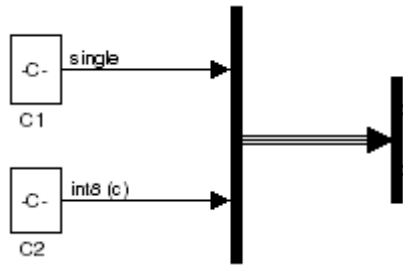
The notation (c) following the data type of a signal indicates that the signal is complex.

Signal Dimensions

Display the dimensions of nonscalar signals next to the line that carries the signal.
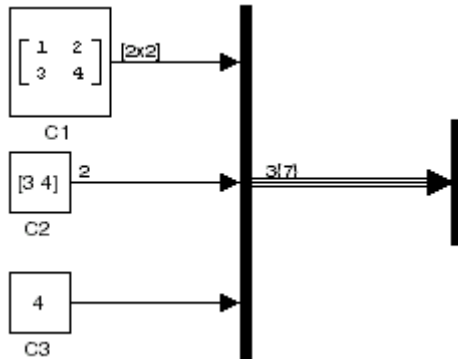
The format of the display depends on whether the line represents a single signal or a bus. If the line represents a single vector signal, Simulink® displays the width of the signal. If the line represents a single matrix signal, Simulink displays its dimensions as [N_1xN_2] where N_i is the size of the ith dimension of the signal. If the line represents a bus carrying signals of the same data type, Simulink displays N{M} where N is the number of signals carried by the bus and M is the total number of signal elements carried by the bus. If the bus carries signals of different data types, Simulink displays only the total number of signal elements {M}.

Signal Resolution Indicators

The Simulink Editor by default graphically indicates signals that must resolve to signal objects. For any labeled signal whose **Signal name must resolve to signal object** property is enabled, a signal resolution icon appears to the left of the signal name. The icon looks like this:

A signal resolution icon indicates only that a signal's **Signal name must resolve to signal object** property is enabled. The icon does not indicate whether the signal is actually resolved, and does not appear on a signal that is implicitly resolved without its **Signal name must resolve to signal object** property being enabled.

Where multiple labels exist, each label displays a signal resolution icon. No icon appears on an unlabeled branch. In the next figure, signal x2 must resolve to a signal object, so a signal resolution icon appears to the left of the signal name in each label:
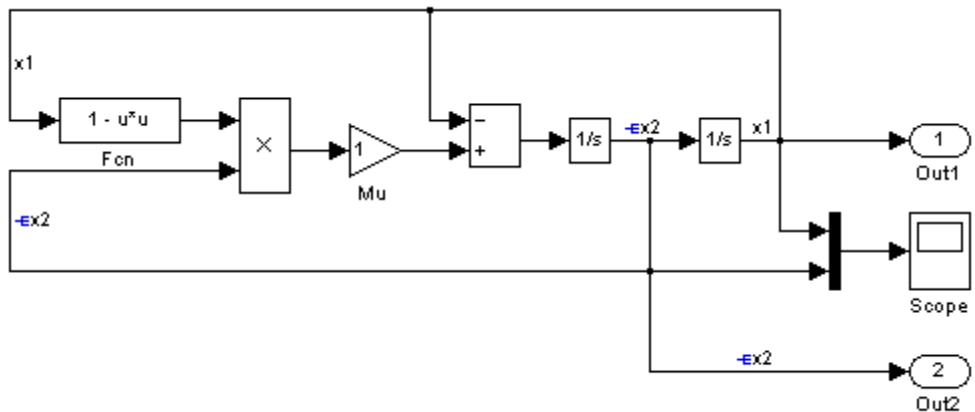
Copyright 2004 The MathWorks, Inc.

To suppress the display of signal resolution icons, in the model window deselect **Format > Port/Signal Displays > Signal Resolution Indicators**, which is selected by default. To restore signal resolution icons, reselect **Signal Resolution Indicators**. Individual signals cannot be set to show or

hide signal resolution indicators independently of the setting for the whole model. For additional information, see:

- "Resolving Symbols" on page 3-69
- "Initializing Signals and Discrete States" on page 8-62
- Simulink.Signal

Wide Nonscalar Lines

Draws lines that carry vector or matrix signals wider than lines that carry scalar signals.
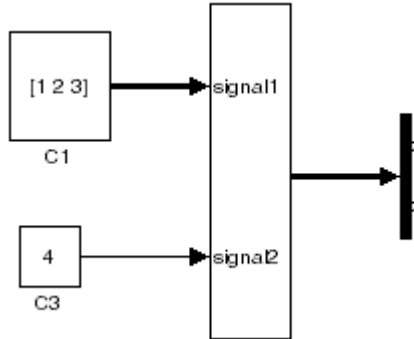
See Chapter 9, "Using Composite Signals" for more information about vector and matrix signals.

Working with Signal Groups

About the Signal Groups

The Signal Builder block allows you to create interchangeable groups of signal sources and quickly switch the groups into and out of a model. Signal groups can greatly facilitate testing a model, especially when used in conjunction with Simulink®'s Assertion blocks and Simulink Verification and Validation's Model Coverage Tool. For a description of the Model Coverage Tool, see the "Simulink Verification and Validation User's Guide" on The MathWorks™ Web site (www.mathworks.com).

Creating a Signal Group Set

To create an interchangeable set of signal groups:

1 Drag an instance of the Signal Builder block from the Simulink Sources library and drop it into your model.
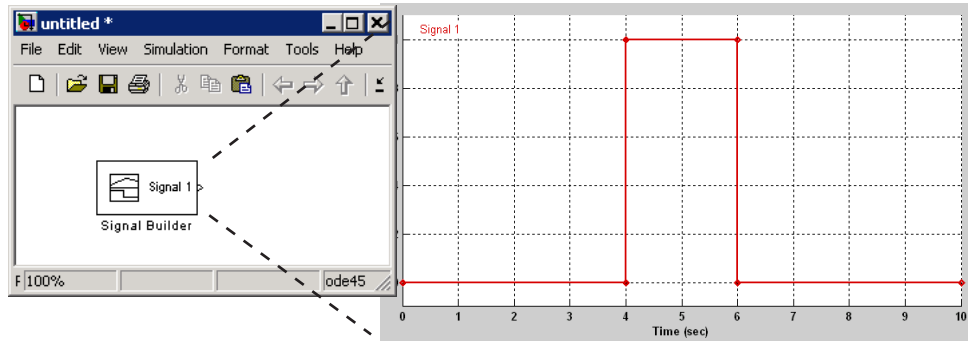
By default, the block represents a single signal group containing a single signal source that outputs a square wave pulse.

2 Use the block's signal editor (see "Signal Builder Dialog Box" on page 8-78) 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 8-92 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.
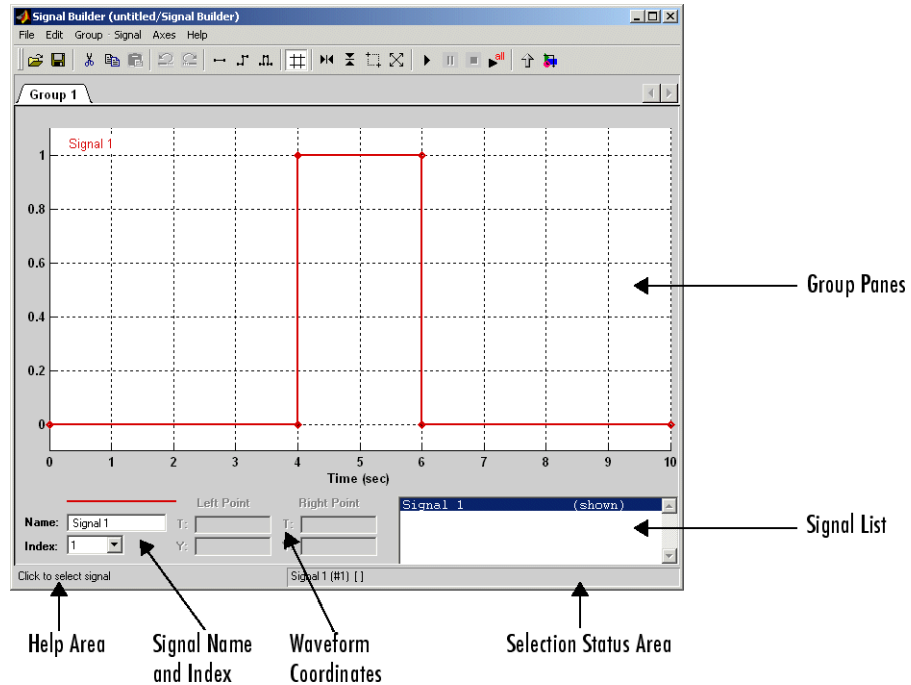
The **Signal Builder** dialog box allows you to create and modify signal groups represented by a Signal Builder block. The **Signal Builder** dialog box includes the following controls.

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 8-90). 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 8-81) 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 8-84).

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 8-83).

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 8-83).

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.
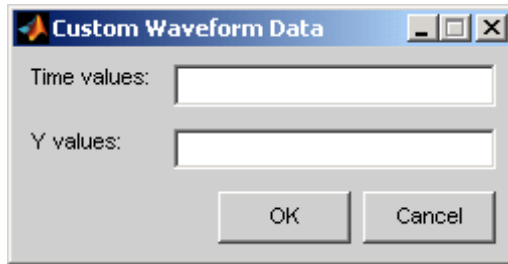
The dialog box allows you to specify a custom piecewise linear waveform to be added to the groups defined by the Signal Builder block. Enter the custom waveform's time coordinates in the **Time values** field and the corresponding signal amplitudes in the **Y values** field. The entries in either field 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 8-80).

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.
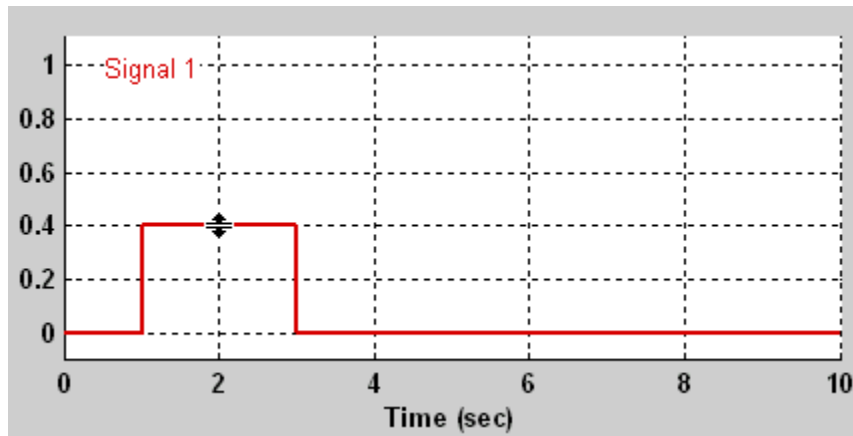
The Signal Builder displays the waveform's points to indicate that the waveform is selected.
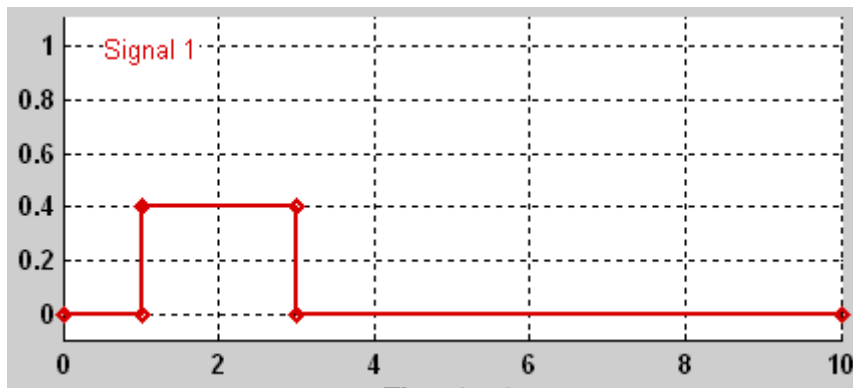
To deselect a waveform, left-click any point on the waveform graph that is not on the waveform itself or press the **Esc** key.

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.
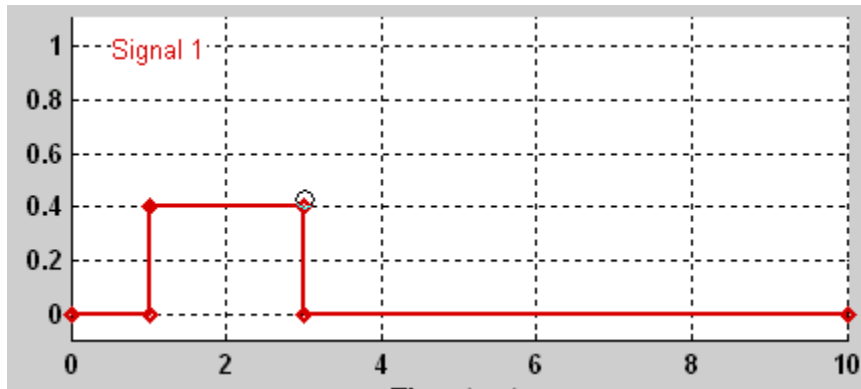
Left-click the point with the mouse. The Signal Builder draws a circle around the point to indicate that it is selected.
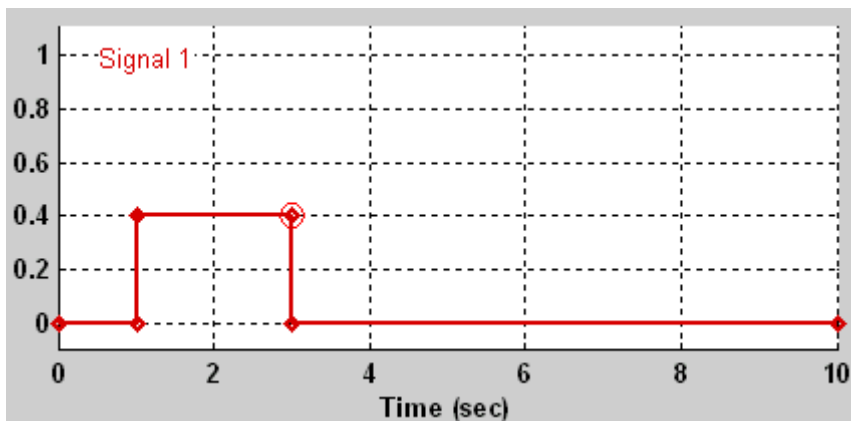
To deselect the point, press the **Esc** key.

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.
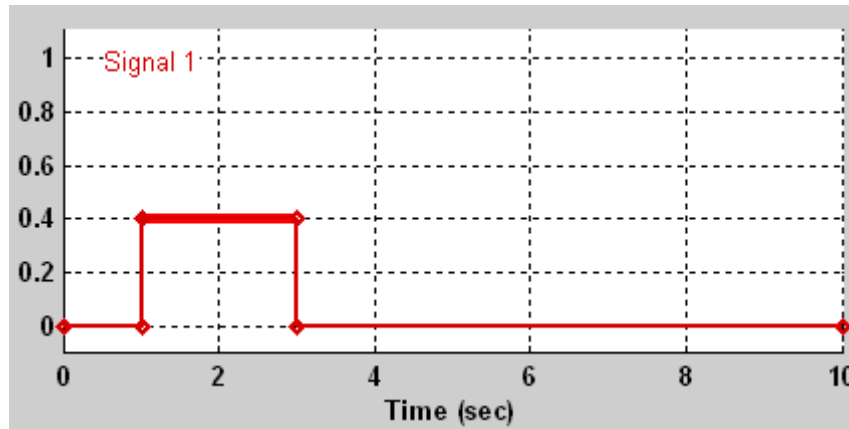
To deselect the segment, press the **Esc** key.

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 8-88) 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.
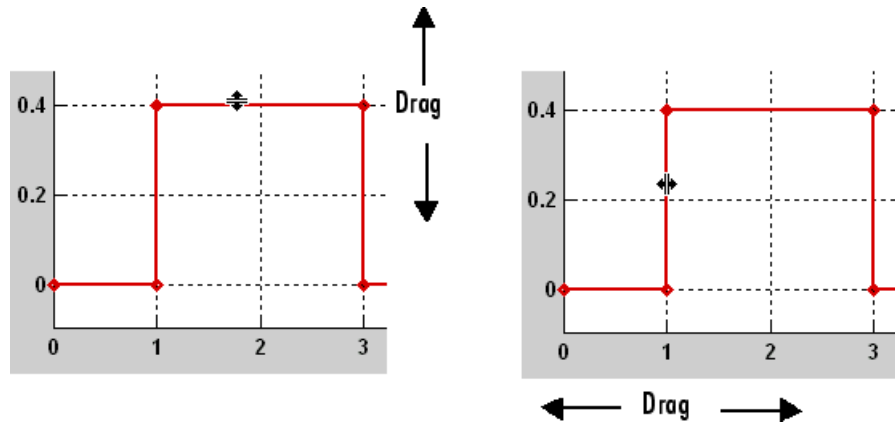
Press the left mouse button and drag the segment in the direction indicated to the desired location. You can also use the arrow keys on your keyboard to move the selected line segment.

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 8-90).

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 8-93 for more information).

Changing a Signal Builder's Time Range

To change the time range, select **Change Time Range** from the Signal Builder's **Axes** menu. A dialog box appears.
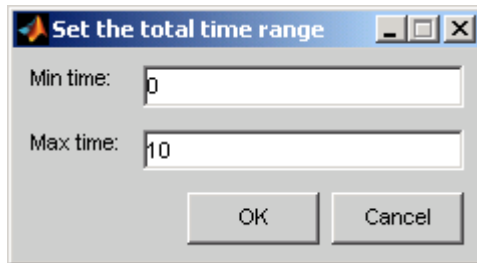
Edit the **Min time** and **Max time** fields as necessary to reflect the beginning and ending times of the new time range, respectively. Click **OK**.

Exporting Signal Group Data

To export the data that define a Signal Builder block's signal groups to the MATLAB workspace, select **Export to Workspace** from the block's **File** menu. A dialog box appears.
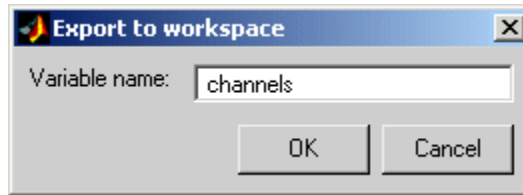
The Signal Builder exports the data by default to a workspace variable named channels. To export to a differently named variable, enter the variable's name in the **Variable name** field. Click **OK**. The Signal Builder exports the data to the workspace as the value of the specified variable.

The exported data is an array of structures. The structure's xData and yData fields contain the coordinate points defining signals in the currently selected signal group. You can access the coordinate values defining signals associated with other signal groups from the structure's allXData and allYData fields.

Printing, Exporting, and Copying Waveforms

The **Signal Builder** dialog box allows you to print, export, and copy the waveforms visible in the active signal group.

To print the waveforms to a printer, select **Print** from the block's **File** menu.

You can also export the waveforms to other destinations by using the **Export** option from the block's **File** menu. From this submenu, select one of the following destinations:

- **To File** — Converts the current view to a graphics file.

  Select the format of the graphics file from the **Save as type** drop-down list on the resulting **Export** dialog box.

- **To Figure** — Converts the current view to a MATLAB figure window.

To copy the waveforms to the system clipboard for pasting into other applications, select **Copy Figure To Clipboard** from the block's **Edit** menu.

Simulating with Signal Groups

You can use standard simulation commands to run models containing Signal Builder blocks or you can use the Signal Builder's **Run all** command (see "Running All Signal Groups" on page 8-92).

Activating a Signal Group

During a simulation, a Signal Builder block always outputs the active signal group. The active signal group is the group selected in the **Signal Builder** dialog box for that block, if the dialog box is open, otherwise the group that was selected when the dialog box was last closed. To activate a group, open the group's **Signal Builder** dialog box and select the group.

Running Different Signal Groups in Succession

The Signal Builder's toolbar includes the standard Simulink buttons for running a simulation. This facilitates running several different signal groups in succession. For example, you can open the dialog box, select a group, run a simulation, select another group, run a simulation, etc., all from the Signal Builder's dialog box.

Running All Signal Groups

To run all the signal groups defined by a Signal Builder block, open the block's dialog box and click the **Run all** button

from the Signal Builder's toolbar. The **Run all** command runs a series of simulations, one for each signal group defined by the block. If you installed Simulink Verification and Validation on your system and are using the Model Coverage Tool, the **Run all** command configures the tool to collect and save coverage data for each simulation in the MATLAB workspace and display a report of the combined coverage results at the end of the last simulation. This allows you to quickly determine how well a set of signal groups tests your model.

> **Note** To stop a series of simulations started by the **Run all** command, enter **Ctrl+C** at the MATLAB command line.

Simulation Options Dialog Box

The **Simulation Options** dialog box allows you to specify simulation options pertaining to the Signal Builder. To display the dialog box, select **Simulation Options** from the Signal Builder's **File** menu. The dialog box appears.
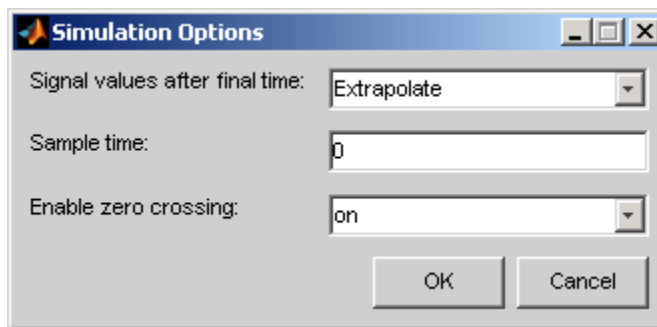
The dialog box allows you to specify the following options.

Signal values after final time

The setting of this control determines the output of the Signal Builder block if a simulation runs longer than the period defined by the block. The options are

- Hold final value

Selecting this option causes the Signal Builder block to output the last defined value of each signal in the currently active group for the remainder of the simulation.
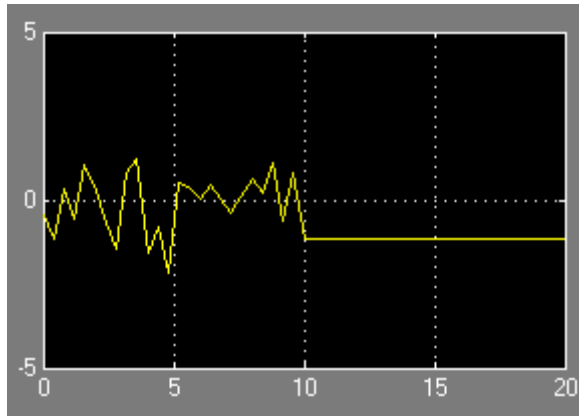
- Extrapolate

  Selecting this option causes the Signal Builder block to output values extrapolated from the last defined value of each signal in the currently active group for the remainder of the simulation.
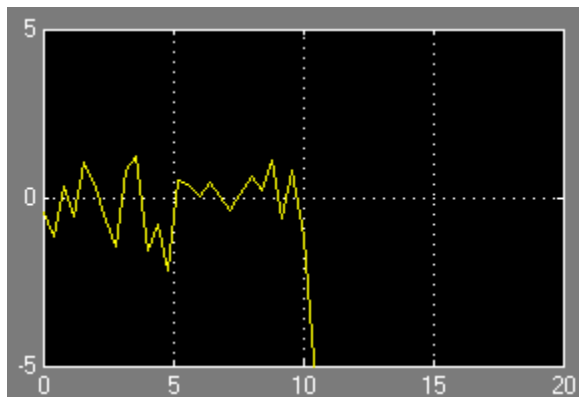
- Set to zero

Selecting this option causes the Signal Builder block to output zero for the remainder of the simulation.
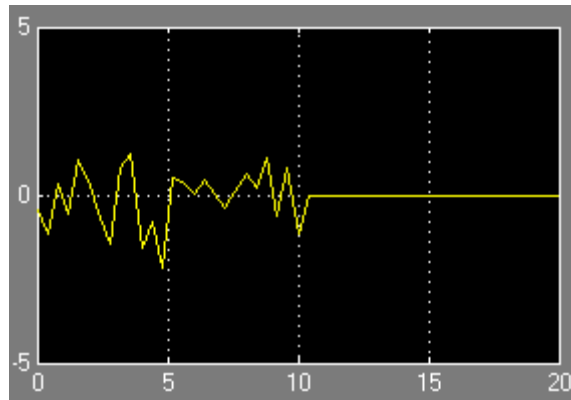
Sample time

Determines whether the Signal Builder block outputs a continuous (the default) or a discrete signal. If you want the block to output a continuous signal, enter 0 in this field. For example, the following display shows the output of a Signal Builder block set to output a continuous Gaussian waveform over a period of 10 seconds.
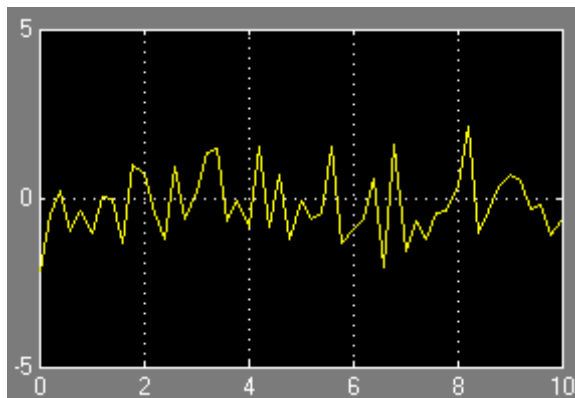
If you want the block to output a discrete signal, enter the sample time of the signal in this field. The following example shows the output of a Signal Builder block set to emit a discrete Gaussian waveform having a 0.5 second sample time.
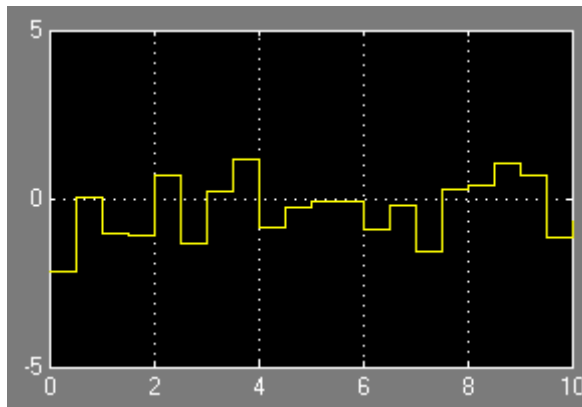
Enable zero crossing

Specifies whether the Signal Builder block detects zero-crossing events (enabled by default). For more information, see "Zero-Crossing Detection" on page 2-20.

9

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 8-3.

Two types of composite signals are provided by the Simulink® software: 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 9-30 for details.

In some cases, muxes and buses can be intermixed; implicit conversion occurs when needed. However, as a rule you should not intermix muxes and buses. The practice may become unsupported in the future, and should not be used in new applications. Diagnostics are provided 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 the Simulink® software documentation, indexed vectors are also called "vectors" and "wide signals", and both "mux" and "vector" appear in 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.
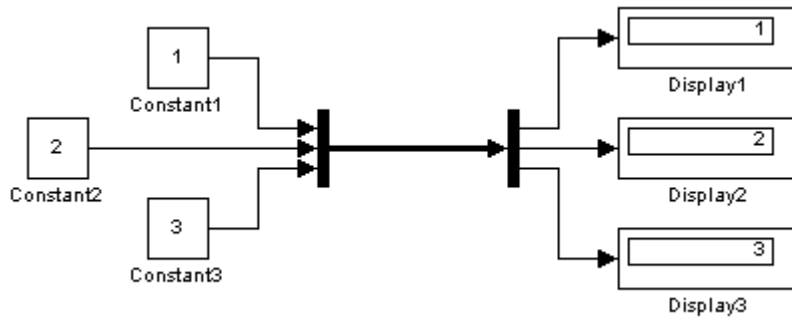
The Mux and Demux blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the mux signal, is wide because the model has been built with **Format > Port/Signal Displays > Wide Nonscalar Lines** enabled in the model menu. See "Displaying Signal Properties" on page 8-73 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:
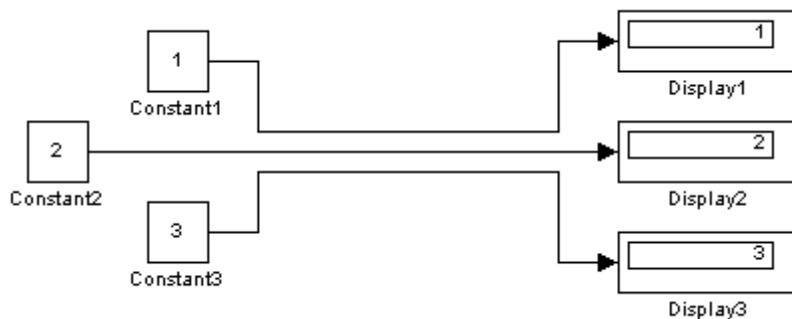
Signals input to a Mux block can be virtual composite signals, but the resulting mux is flat, not hierarchical. The signals in the mux appear in the order in

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 9-5 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 9-37 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 9-37 for details.

Using Buses

In this section...
"Introduction" on page 9-5
"Creating and Accessing a Bus" on page 9-6
"Nesting Buses" on page 9-7
"Bus-Capable Blocks" on page 9-8
"Circular Bus Definitions" on page 9-8

Introduction

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 and Nonvirtual Buses" on page 9-30 for details. The Signal Routing library provides three blocks that you can use for implementing buses:

Bus Creator
>   Create a bus that contains specified elements

Bus Assignment
>   Replace specified bus elements.

Bus Selector
>   Select elements from a bus.

Each of these blocks is virtual or nonvirtual depending on whether the bus it processes is virtual or nonvirtual. The Simulink® software chooses the block type, and changes it automatically if the bus type changes.

Creating and Accessing a Bus

To create and access 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.
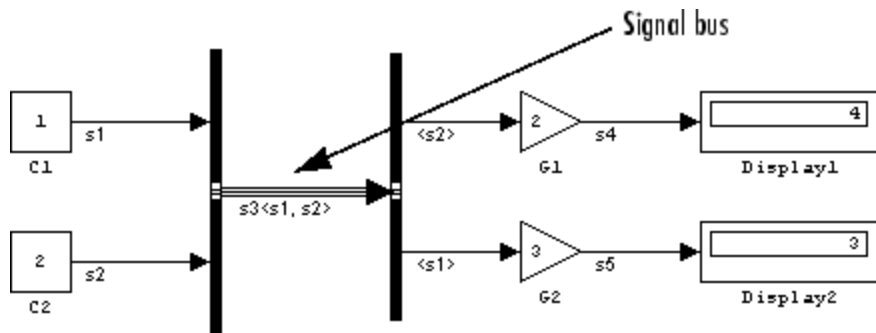
The Bus Creator and Bus Selector blocks are the left and right vertical bars, respectively. Consistent with the goal of reducing visual complexity, neither block displays a name. The line connecting the blocks, representing the bus signal, is tripled because the model has been built, and the middle

line is solid because the bus is virtual. The line would be dashed if the bus were nonvirtual:

Virtual Bus	⟹
Nonvirtual Bus	⇢

See "Signal Line Styles" on page 8-7 for more about the graphical appearance of signals. You can also display other signal characteristics graphically, as described under "Displaying Signal Properties" on page 8-73. For more information about creating and accessing buses, see the reference documentation for the Bus Creator, Bus Selector, and Bus Assignment blocks.

Nesting Buses

Buses can be nested to any depth. The Simulink software automatically handles most of the complexities involved. For example, the next figure shows six signals nested into two buses, which are nested into one, followed by separation into two buses and then into six separate signals:
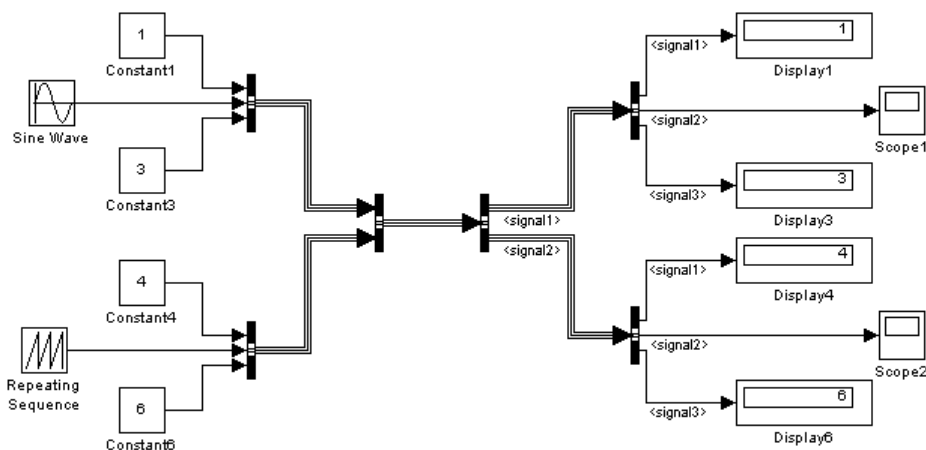
The six signals retain their separate identities just as if no bus creation and selection occurred, as shown by the Display and Scope blocks.
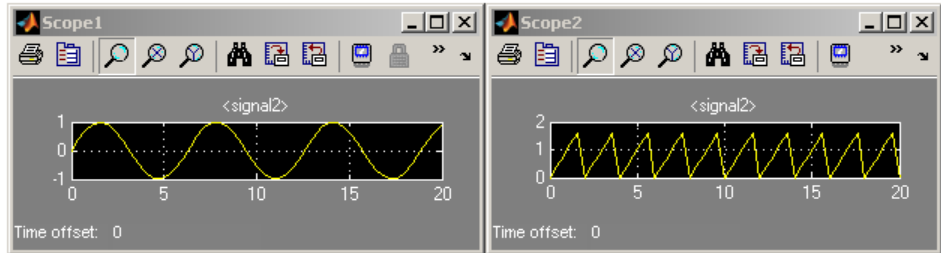
Specifying nonvirtual buses, like those in the previous figure, requires only cloning blocks, setting parameters, and connecting signals. Bus Creator and Bus Selector blocks have two ports by default. See the Bus Creator and Bus Selector block documentation for information about how to specify buses of different widths.

Bus-Capable Blocks

A *bus-capable block* is a block through which both virtual and nonvirtual buses can propagate. All virtual blocks are bus-capable. 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.

Circular Bus Definitions

The ability to include a bus as an element of another bus creates the possibility of a loop of Bus Creator blocks, Bus Selector blocks, and bus-capable blocks

that inadvertently includes a bus as an element of itself. The resulting circular definition cannot be resolved and therefore causes an error.

The error message that appears specifies the location at which the Simulink software determined that the circular structure exists. The error is not really at any one location: the structure as a whole is in error. Nonetheless, the location cited in the error message can be useful for beginning to trace the definition cycle, the structure of which may not be obvious on visual inspection.

1 Begin by selecting a signal line associated with the location cited in the error message.

2 Choose **Highlight to Source** or **Highlight to Destination** from the signal's Context menu. (See "Displaying Signal Sources and Destinations" on page 8-18 for more information.)

3 Continue choosing signals and highlighting their sources and destinations until the cycle becomes clear.

4 Restructure the model as needed to eliminate the circular bus definition.

Because the problem is a circular definition rather than a circular computation, the cycle cannot be broken by inserting additional blocks, in the way that an algebraic loop can be broken by inserting a Unit Delay block. No alternative exists but to restructure the model to eliminate the circular bus definition.

Using the Bus Editor

Introduction

The Simulink® Bus Editor is a tool similar to the Model Explorer, but is customized for use with bus objects. You can use the Simulink Bus Editor to:

- Create new bus objects and elements
- Navigate, change, and nest bus objects
- Import existing bus objects from an M-file or MAT-file
- Export bus objects to an M-file or MAT-file

For a description of buses and their use, see "Using Buses" on page 9-5. For a description of bus objects and their use, see .

Base Workspace Bus Objects

All bus objects exist in the MATLAB® base workspace. Bus Editor actions take effect in the base workspace immediately, and can be used by Simulink models as soon as each action is complete. The Bus Editor does not have a workspace of its own: it acts only on the base workspace. Bus Editor actions

do not directly affect bus object definitions in saved M-files or MAT-files. To save changed bus object definitions, export them from the base workspace into M-files or MAT-files, as described in "Exporting Bus Objects" on page 9-26.

Opening the Bus Editor

You can open the Bus Editor in any of these ways:

- Select **Bus Editor** from the model editor's **Tools** menu.

- Click the **Launch Bus Editor** button on a bus object's dialog box in the Model Explorer.

- Enter buseditor at the command line of the MATLAB software.

After you have performed any of these actions, the Bus Editor appears. If no bus objects exist, the Bus Editor looks like this:
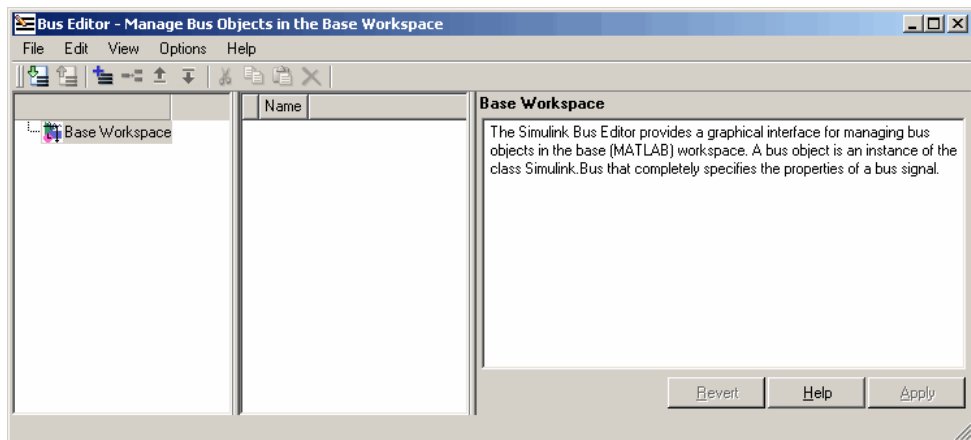
The Bus Editor provides menu choices that you can use to execute all Bus Editor commands. The editor also provides toolbar icons and keyboard shortcuts for all commonly used commands, including the standard MATLAB GUI shortcuts for Cut, Copy, Paste, and Delete. The Toolbar Tip for each icon describes the command, and the menu entry for each command shows any shortcut. The icons for commands that are specific to the Bus Editor are as follows:

Command	Icon	Description
Import		Import the contents of an M-file or MAT-file into the base workspace.
Export		Export all bus objects and elements to an M-file or MAT-file.
Create		Create a new bus object in the base workspace.
Insert		Add a bus element below the currently selected bus entity.
Move Up		Move the selected element up in the list of a bus object's elements.
Move Down		Move the selected element down in the list of a bus object's elements.

For brevity, this section refers only to menu commands. You can use toolbar icons and keyboard shortcuts instead wherever that is convenient.

Displaying Bus Objects

The Bus Editor is similar to the Model Explorer (which can display bus objects but cannot edit them) and uses the same three panes to display bus objects:

- **Hierarchy pane** (left) — Displays the bus objects defined in the base workspace

- **Contents pane** (center) — Displays the elements of the bus object selected in the Hierarchy pane

- **Dialog pane** (right) — Displays for editing the current selection in the Contents or Hierarchy pane

Hierarchy Pane

If no bus objects exist in the base workspace, the Hierarchy pane shows only **Base Workspace**, which is the root of the hierarchy of bus objects. The Bus Editor than looks as shown in the previous figure. As you create or import bus objects, they appear in the Hierarchy Pane as nodes subordinate to **Base Workspace**. The bus objects appear in alphabetical order. The next figure shows the Bus Editor with two bus objects, control and main, defined in the base workspace:
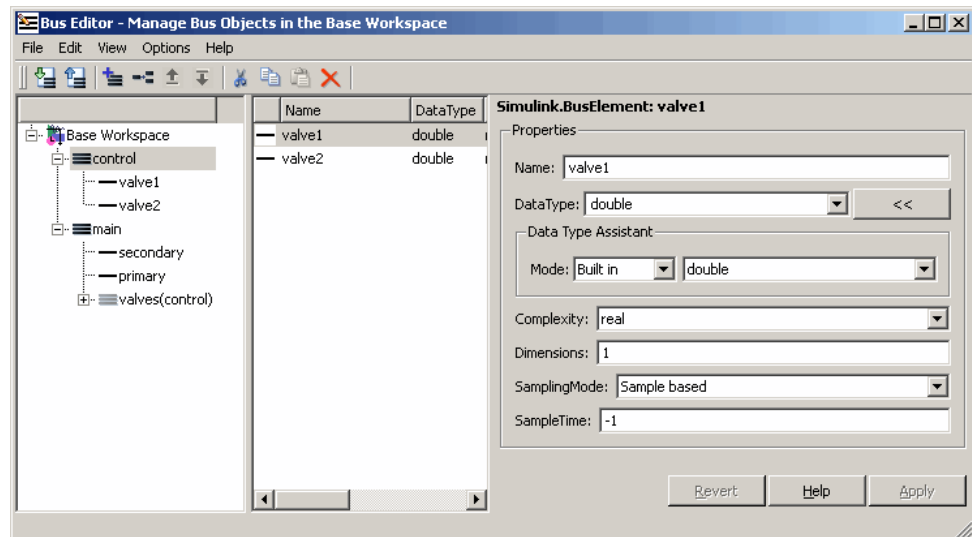
The Hierarchy pane displays each bus object as an expandable node. The root of the node displays the name of the bus object, and (if the bus contains any elements) a button for expanding and collapsing the node. Expanding a bus node displays named subnodes that represent the bus's top-level elements. In the preceding figure, both bus objects are fully expanded , and control is selected.

Contents Pane

Selecting any top-level bus object in the Hierarchy Pane displays the object's elements in the Contents pane. In the previous figure, the elements of bus object control, valve1 and valve2, appear. Each element's properties appear to the right of the element's name. These properties are editable, and you can

edit the properties of multiple elements in one operation, as described in "Editing in the Contents Pane" on page 9-22.

Dialog Pane

When a bus object is selected in the Hierarchy pane, or a bus object or element is selected in the Contents pane, the properties of the selected item appear in the Dialog pane. In the previous figure, valve1 is selected in the Contents pane, so the Dialog pane shows its properties. These properties are editable, and changes can be reverted or applied using the buttons below the Dialog pane, as described in "Editing in the Dialog Pane" on page 9-24.

Creating Bus Objects

To use the Bus Editor to create a new bus object in the base workspace:

1 Choose **File > Add Bus**.

A new bus object with a default name and properties is created immediately in the base workspace. The object appears in the Hierarchy pane, and its default properties appear in the Dialog pane:
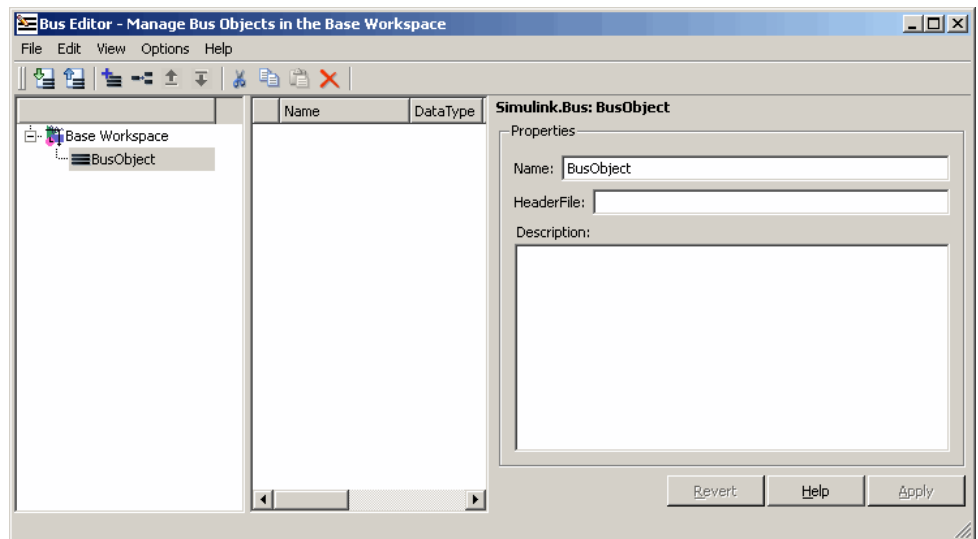
2 To specify the bus object name and other properties, in the Dialog pane:

    a Specify the **Name** of the of the new bus object (or you can retain the default name). The name must be unique in the base workspace.

    b Optionally, specify a C **Header file** that defines a user-defined type corresponding to this bus object. This header file has no effect on Simulink simulation; it is used only by Real-Time Workshop® software to generate code.

    c Optionally, specify a **Description** that provides information about the bus object to human readers. This description has no effect on Simulink simulation; it exists only for human convenience.

3 Click **Apply**.

The properties of the bus object on the base workspace change as specified. If you rename BusObject to main, the Bus Editor looks like this:
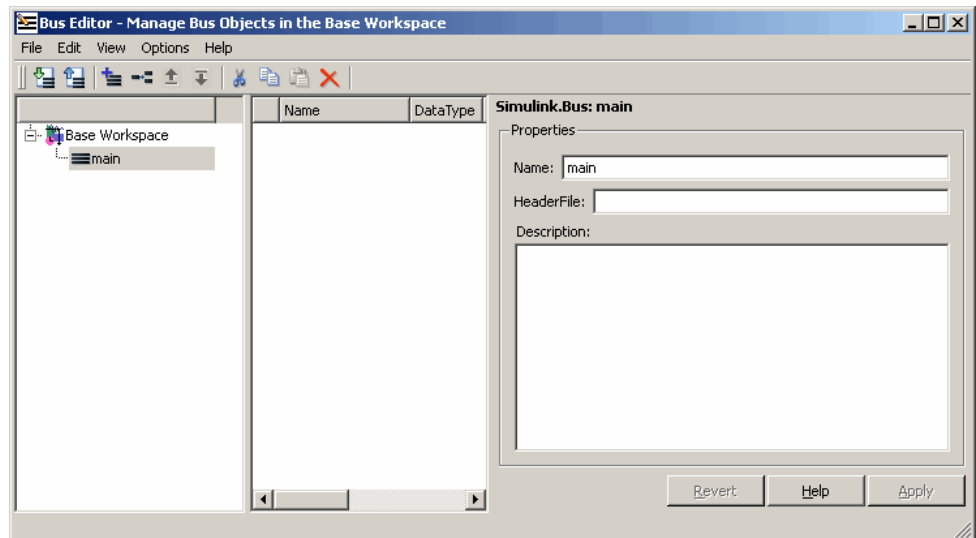
You can use **Add Bus** at any time to create a new bus object in the base workspace, then set the name and properties of the object as needed. You can intermix creating bus objects and specifying their properties in any order. The hierarchy pane reorders as needed to display all bus objects in alphabetical

order. If you add an additional bus object named `control`, the Bus Editor looks like this:
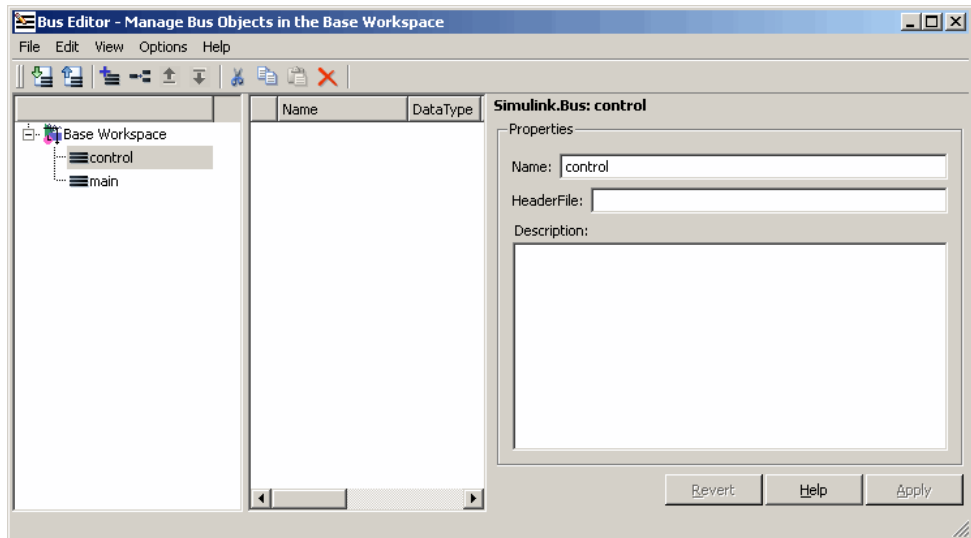
Creating Bus Elements

Every bus element belongs to a specific bus. To create a new bus element:

1 In the Hierarchy pane, select the entity below which to create the new element. The entity can be a bus or a bus element. The new element will belong to the selected bus object, or to the bus object that contains the selected element. The previous figure shows the `control` bus object selected.

2 Choose **File > Add/Insert Bus Element**.

A new bus element with a default name and properties is created immediately in the applicable bus object. The object appears in the Hierarchy pane immediately below the previously selected entity, and its default properties appear in the Dialog pane:
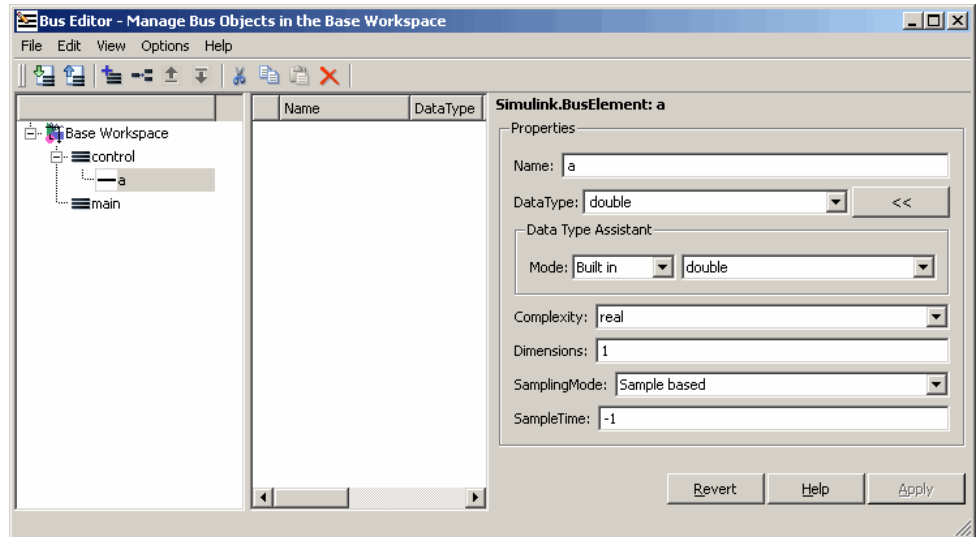
3 To specify the bus element name and other properties, in the Dialog pane:

   a Specify the **Name** of the of the new bus element (or you can retain the default name). The name must be unique among the elements of the bus object.

   b Specify the other properties of the element. These must match exactly the properties of the corresponding signal within the bus, and can be anything that a legal signal might have. The Data Type Assistant appears in the Dialog pane to help specify the element's data type. You can specify any available data type, including a user-defined data type.

4 Click **Apply**.

The properties of the bus element of the bus object in the base workspace change as specified. If you rename the new element `a` to `valve1`, the Bus Editor looks like this:
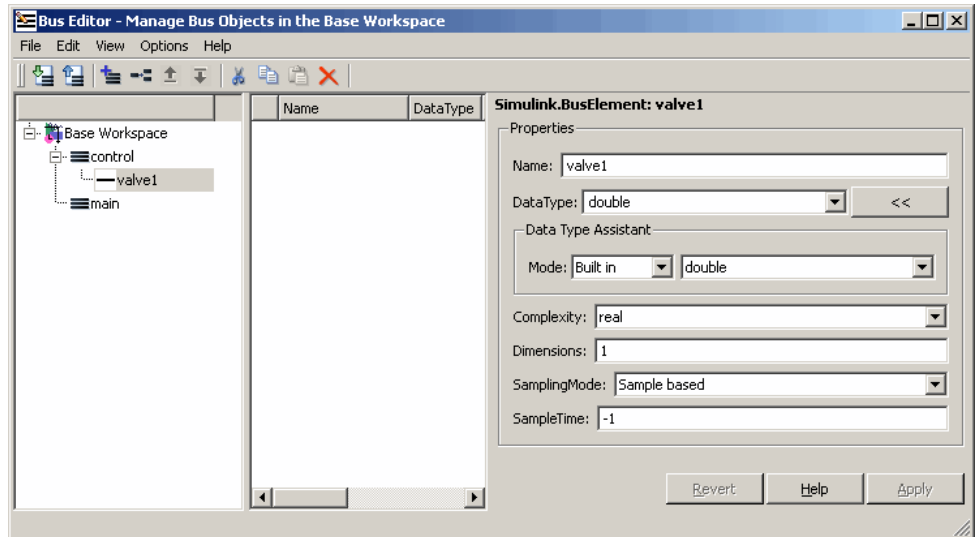
You can use **Add/Insert Bus Element** at any time to create a new bus element in any bus object. You can intermix creating bus objects and specifying their properties in any order. The order of the other bus elements in the bus object does not change when a new element is added. If you add element `valve2` to `control`, and `secondary` and `primary` to `main`, the Bus Editor looks like this:
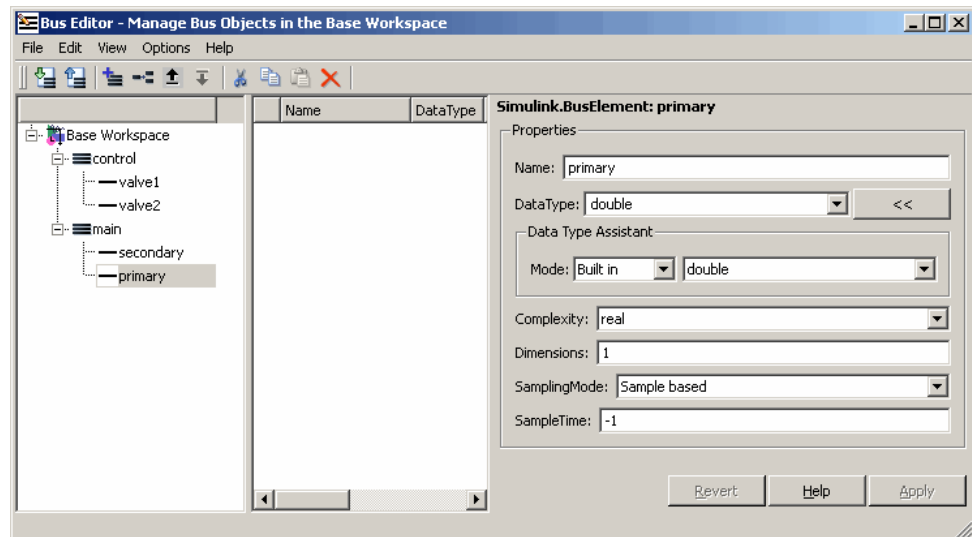
Nesting Bus Definitions

As described in "Nesting Buses" on page 9-7, any signal in a bus can be another bus, which can in turn contain subordinate buses, and so on to any depth. Describing nested buses with bus objects requires nesting the bus definitions that the objects provide.

Every bus object inherently defines a data type whose properties are those specified by the object. To nest one bus definition in another, you assign to an element of one bus object a data type that is defined by another bus object. The element then represents a nested bus whose structure is given by the bus object that provides its data type.

A data type defined by a bus object is called a *bus type*. Nesting buses by assigning bus types to elements, rather than by subordinating the bus objects that define the types, allows the same bus definition to be used conveniently in multiple contexts without unwanted interactions. To specify that an element of a bus object represents a nested bus definition:

1 Create a bus element to represent the nested bus definition, in the appropriate position under the containing bus object, and give the element the desired name. (You can also use an existing element.)

2 Use the Dialog pane to set the data type of the element to the name of a bus object. The Data Type Assistant shows the names of all available bus types. (You can also specify a nonexistent bus type and define the object later.)

In the preceding figure, if you add to bus object `main` a third element named `valves`, set the data type of `valves` to be `control` (the name of the other defined bus object) and expand the new element `valves`, the Bus Editor looks like this:
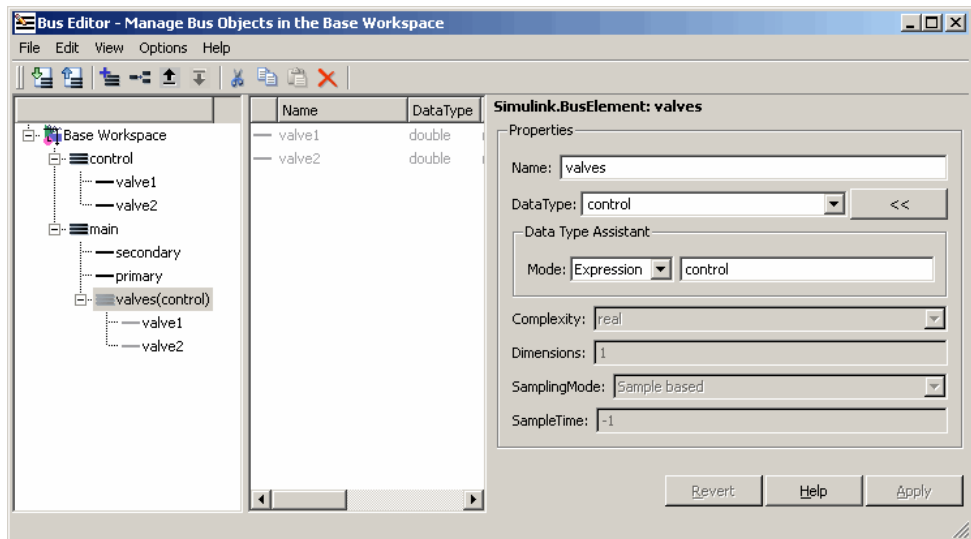
The bus object `main` shown in the Bus Editor now defines the same structure used by the bus signal `main` in the next figure:
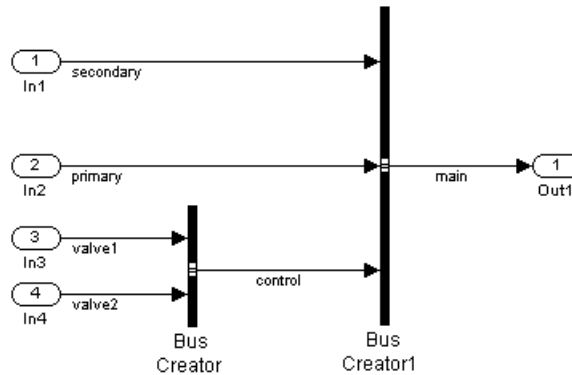
The distinction between a bus object and the bus type that it defines can be useful for initially understanding how nested bus objects work and how the Bus Editor handles them. In other contexts, the distinction is mostly an implementation detail, and describing bus objects themselves as being nested is more convenient. The rest of this chapter follows that convention.

You can nest a bus object in as many different bus objects as desired, and as many times in the same bus object as desired. You can nest bus objects to any depth, but you cannot define a circular structure by directly or indirectly nesting a bus object within itself. If you try to define a circular structure, the Bus Editor posts a Notice and sets the data type of the element that would have created the cycle to double. Click **OK** to dismiss the Notice and continue using the editor.

You can use the Hierarchy pane to explore nested bus objects by expanding the objects, but you cannot change any property of a bus object anywhere that it appears in nested form. To change the properties of a nested bus object, you must change the source object, which is accessible at the top level in the Hierarchy pane.

You can jump from a nested bus object to the source object by selecting the nested object and choosing **Go to '*element*'** from its Context menu.

Changing Bus Entities

You can use the Bus Editor to change and delete existing bus objects and elements at any time. All three panes allow you to change the entities that they display. Changes that create, reorder, or delete entities take effect immediately in the base workspace. Changes to properties take effect when you Apply them, or can be Canceled, leaving the properties unchanged. The Bus Editor does not provide an Undo capability.

The Bus Editor provides comprehensive GUI capabilities for changing bus entities. You can Cut, Copy, and Paste within and between panes in any way that has a legal result. The Hierarchy and Dialog panes provide a Context menu for the current selection. Pasting a Copied entity always creates a copy, as distinct from a pointer to the original. The Bus Editor automatically changes names when needed to avoid duplication. The rest of this section describes capabilities that are specific to one of the Bus Editor panes.

Editing in the Hierarchy pane

As you use the Bus Editor, the Hierarchy pane automatically reorders the bus objects it displays to maintain alphabetical order. This behavior cannot be changed. However, the elements under a bus object can appear in any order. To change that order, cut and paste elements as needed, or move elements up and down as follows:

1 Select the element to be moved.

2 Choose **Edit > Move Element Up** or **Edit > Move Element Down**.

You cannot Paste one bus object under another to create a nested bus object specification. To specify a nested bus, you must change the data type of a bus element to be the type of an existing bus object, as described in "Nesting Buses" on page 9-7.

Editing in the Contents Pane

Selecting any top-level bus object in the Hierarchy Pane displays the object's elements in the Contents pane. Each element's properties appear to the right of the element's name, and can be edited. To change a property displayed in the Contents pane, click the value, enter a new value, then press Return.

Choose **View > Dialog View** to hide the Dialog pane to provide more room to display properties in the Contents pane. Choose the command again to redisplay the Dialog pane.

You can use the mouse and keyboard to select multiple elements in the Contents pane. The selected entities need not be contiguous. You can then perform any operation that you could on a single entity selected in the pane, including operations performed with the Context menu. Clicking and editing a value in any selected element changes that value in them all.

The next figure shows the Bus Editor with **Dialog View** enabled, two elements selected in the Contents pane, and the **DataType** property selected for editing in the second element:
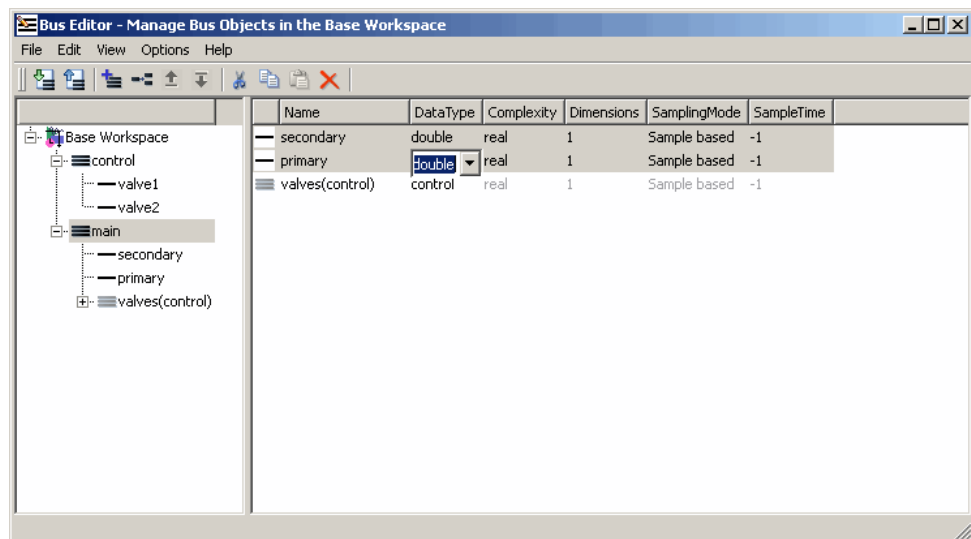
If you change the value of **DataType** to `single` and press Return, the value changes for both elements. The effect is the same no matter which element you edit in a multiple selection:
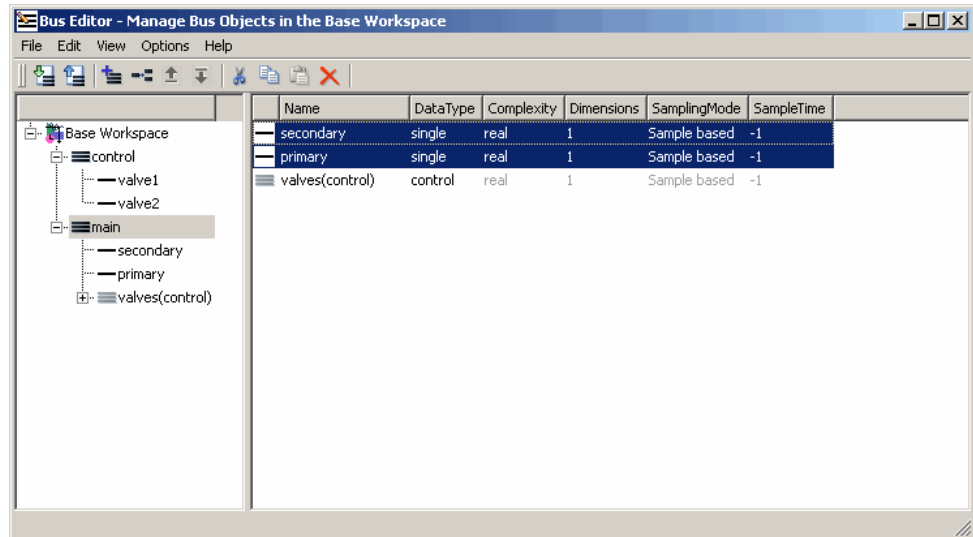
Editing in the Dialog Pane

When a bus object is selected in the Hierarchy pane, or a bus object or element is selected in the Contents pane, the properties of the selected item appear in the Dialog pane. In the next figure, valve1 is selected in the Contents pane, so the Dialog pane shows its properties:
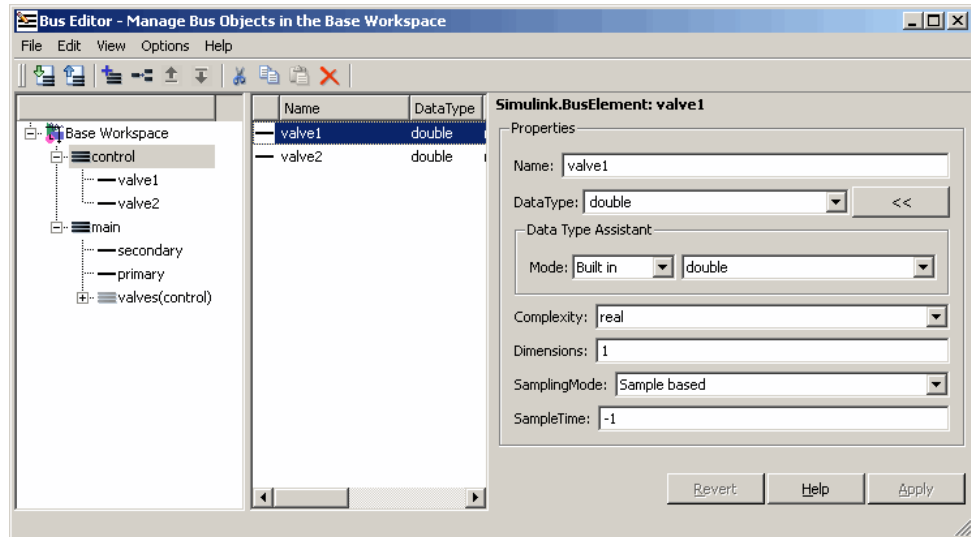
The properties shown in the Dialog pane are editable, and the pane includes the Data Type Assistant. Click **Apply** to save changes, or **Revert** to cancel them and restore the values that existed before any unapplied changes. You can edit only one element at a time in the Dialog pane. If multiple entities are selected in the Contents pane, all fields in the Dialog pane are grayed out:
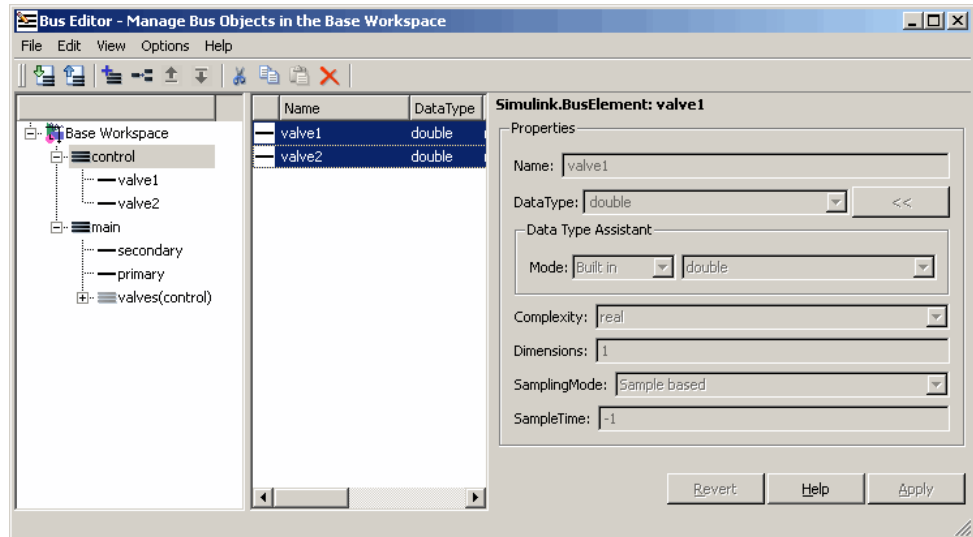
If you use the Dialog pane to change any property of a bus entity, then navigate elsewhere without clicking either **Apply** or **Revert**, a query box appears by default. The query box asks whether to apply changes, ignore changes, or continue as if the navigation had not been tried. You can suppress this query for future operations by checking **Never ask me again** in the box, or by selecting **Options > Auto Apply/Ignore Dialog Changes**.

If you suppress the query, and thereafter navigate away from a change without clicking **Apply** or **Revert**, the Bus Editor automatically applies or discards changes, depending on which action you most recently chose in the box. You can re-enable the query box for future operations by deselecting **Options > Auto Apply/Ignore Dialog Changes**.

Exporting Bus Objects

Like all base workspace objects, bus objects are not saved with a model that uses them, but exist separately in an M-file or MAT-file. You can use the Bus Editor to export some or all bus objects to either type of file.

- If you export bus objects to an M-file, the Bus Editor asks whether to store them in object format or cell format (the default). Specify the desired format.

- If exporting would overwrite an existing M-file or MAT-file, a confirmation dialog box appears. Confirm the export or cancel it and try a different filename.

To export all bus objects from the base workspace to a file:

1 Choose **File > Export to File**.

  The Export dialog box appears.

2 Specify the desired name and format of the export file.

3 Click **Save**.

  All bus objects, and nothing else, are exported to the specified file in the specified format.

To export only selected bus objects from the base workspace to a file:

1 Select a bus object in the Hierarchy pane, or one or more bus objects in the Contents pane.

2 Right-click to display the Context menu.

3 Choose **Export to File** to export only the selected bus objects, or **Export with Related Bus Objects to File** to also export any nested bus objects used by the selected objects.

4 Use the Export dialog box to export the selected bus object(s).

Clicking the **Export** icon in the toolbar is equivalent to choosing **File > Export**, which exports all bus objects whether or not any are selected.

Importing Bus Objects

You can use the Bus Editor to import the definitions in an M-file or MAT-file to the base workspace. Importing an M-file or MAT-file imports the complete contents of the file, not just any bus objects that it contains. If you import a file not exported by the Bus Editor, be careful that it does not contain unwanted definitions previously exported from the base workspace or created programmatically.

To import bus objects from a file to the base workspace:

1 Choose **File > Import into Base Workspace**.

2 Use the Open File dialog box to navigate to and import the desired file.

Before importing the file, the Bus Editor posts a warning that importing the file will overwrite any variable in the base workspace that has the same name as an entity in the file. Click **Yes** or **No** as appropriate.

Closing the Bus Editor

To close the Bus Editor, choose **File > Close**. Closing the Bus Editor neither saves nor discards changes to bus objects, which remain unaffected in the base workspace. However, if you also close MATLAB without saving changes to bus objects, the changes will be lost. To save bus objects without saving other base workspace contents, use the techniques described in "Exporting Bus Objects" on page 9-26. You can also save bus objects using any MATLAB technique that saves the contents of the base workspace, but the resulting file will contain everything in the base workspace, not just bus objects.

You can configure the Bus Editor so that closing it posts a reminder to save bus objects. To enable the reminder, select **Options > Always Warn Before Closing**. When this option is selected and you try to close the Bus Editor, a reminder appears that asks whether the editor should save bus objects before closing. Click **Yes** to save bus objects and close, **No** to close without saving bus objects, or **Cancel** to dismiss the reminder and continue in the Bus Editor. You can disable the reminder by deselecting **Options > Always Warn Before Closing**.

Using the Bus Object API

The capabilities of the Bus Editor are also available programatically. Some of these capabilities, like importing and exporting M-files and MAT-files, are not specific to bus objects, and are described elsewhere in the MATLAB® and Simulink® documentation. The Simulink software provides the following API functions for use specifically with bus objects (class Simulink.Bus) and bus elements (class Simulink.BusElement):

`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`
>   Export specified bus objects or all bus objects from the base workspace to an M-file in a specified format

In addition, when you use `Simulink.SubSystem.convertToModelReference` to convert an atomic subsystem to a referenced model, you can save any bus objects created during the conversion to an M-file.

Virtual and Nonvirtual Buses

In this section...
"Introduction" on page 9-30
"Creating Nonvirtual Buses" on page 9-31

Introduction

A bus signal can be *virtual*, meaning that it is just a graphical convenience that has no functional effect, or *nonvirtual*, meaning that the signal occupies its own storage. During simulation, a block connected to a virtual bus reads inputs and writes outputs by accessing the memory allocated to the component signals. These signals are typically noncontiguous, and no intermediate memory exists. Simulation results and generated code are exactly the same as if the bus did not exist, which functionally it does not.

By contrast, a block connected to a nonvirtual bus reads inputs and writes outputs by accessing copies of the component signals. The copies are maintained in a contiguous area of memory allocated to the bus. Such a bus is represented by a structure in generated code, which can be helpful when tracing the correspondence between the model and the code.

Compared with nonvirtual buses, virtual buses reduce memory requirements because they do not require a separate contiguous storage block, and execute faster because they do not require copying data to and from that block.

Virtual buses are the default except where nonvirtual buses are explicitly required. A nonvirtual subsystem needs bus objects on all bus inports and outports, as described in 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 Nonvirtual Subsystems" on page 9-33 and "Connecting Buses to Root Level Inports" on page 9-33. Not all blocks can accept buses. See "Bus-Capable Blocks" on page 9-8 for more about which blocks can handle which types of buses. You can use a Signal Conversion block to convert a nonvirtual bus to a virtual bus, and vice versa.

When updating a diagram prior to simulation or code generation, the Simulink® software automatically upgrades virtual to nonvirtual buses where the upgrade prevents an error and does not cause any errors. The upgrade consists of inserting hidden Signal Conversion blocks into the model where needed. You can eliminate the need for the upgrade by specifying a nonvirtual bus in the block where the bus originates, or by manually inserting appropriate conversion blocks. The latter is generally unnecessary, but can be useful to clarify the model.

Creating Nonvirtual Buses

Bus signals do not specify whether they are virtual or nonvirtual; they inherit that specification from the block in which they originate. Every block that creates or requires a nonvirtual bus must have an associated bus object. Those blocks are:

- Bus Creator

- Inport

- Outport

To specify that a bus is nonvirtual:

1 Associate the block with a bus object, as described in .

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 Attributes > Output as nonvirtual bus**.

- **Outport:** Select **Signal Attributes > Output as nonvirtual bus in parent model**.

4 Click **OK** or **Apply**.

Note All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any bus operation that would result in a nonvirtual bus that violates this requirement generates an error.

All buses and signals input to a Bus Creator block that outputs a nonvirtual bus must therefore have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus, to allow the signal or bus to be included in a nonvirtual bus.

Connecting Buses to Nonvirtual Subsystems

In this section...
"Connecting Buses to Root Level Inports" on page 9-33
"Connecting Buses to Nonvirtual Inports" on page 9-33
"Connecting Multi-Rate Buses to Referenced Models" on page 9-35

Connecting Buses to Root Level Inports

If you want a root level Inport of a model to be able to produce 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 produces. See "Data Object Classes" and the `Simulink.Bus` class in the Simulink® software documentation for more information.

Connecting Buses to Nonvirtual 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 the Simulink software to halt the simulation and display an error message. You can avoid this problem, without changing the semantics of your model, by inserting a Signal Conversion block between the Inport block and the Outport block to which it was originally connected.

Consider, for example, the following model:
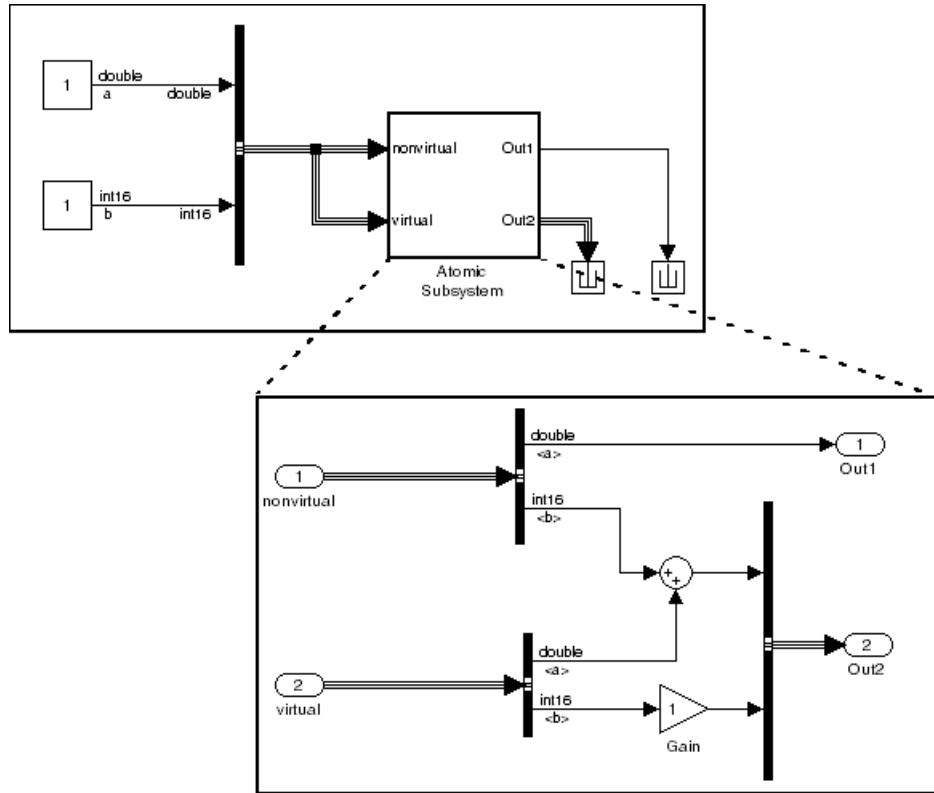
In this model, the Inport labeled `nonvirtual` is nonvirtual because it resides in an atomic subsystem and one of its components (labeled `a`) is directly connected to one of the subsystem's outputs. Further, the bus connected to the subsystem's inputs has components of differing data types. As a result, this model cannot be simulated.

Inserting a Signal Conversion block with the `bus copy` option selected breaks the direct connection to the subsystem's output and thereby enables the Simulink software to simulate the model.
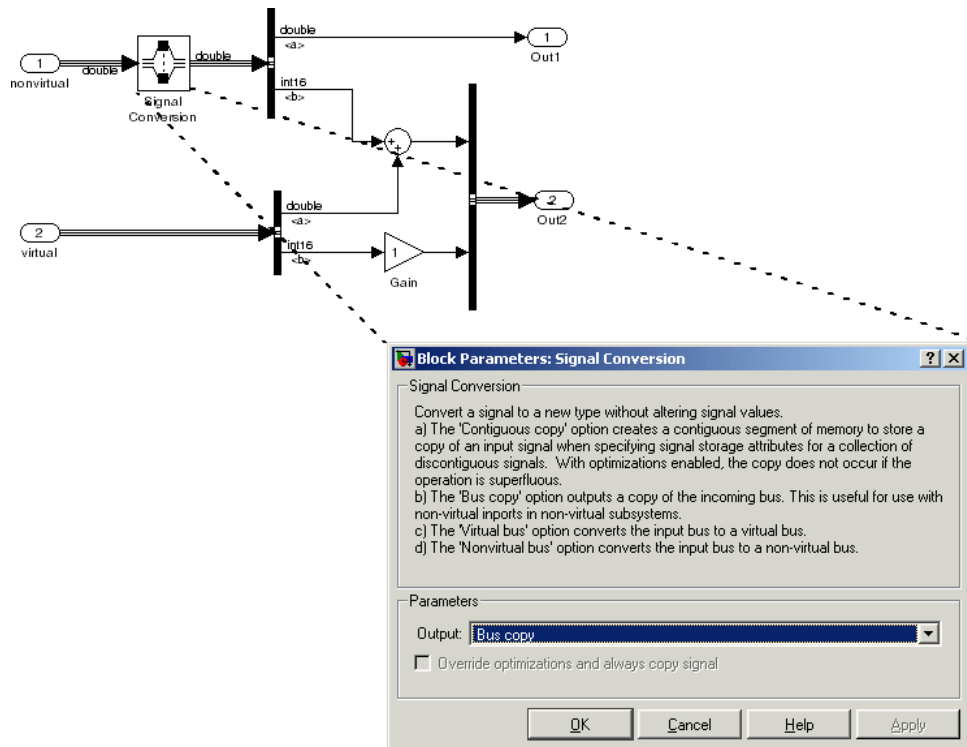
Connecting Multi-Rate Buses to Referenced Models

In a model that uses a fixed-rate solver, referenced models can input only single-rate buses. However, you can input the signals in a multi-rate bus to a referenced model by inserting blocks into the parent and referenced model as follows:

1 **In the parent model:** Insert a Rate Transition block to convert the multi-rate bus to a single-rate bus. The Rate Transition block must specify a rate in its **Block Parameters > Output port sample time** field unless one of the following is true:

- The **Configuration Parameters > Solver** pane specifies a rate:

  - **Periodic sample time constraint** is `Specified`

  - **Sample time properties** contains the specified rate.

- The Inport that accepts the bus in the referenced model specifies a rate in its **Block Properties > Signal Attributes > Sample time** field.

2 In the referenced model: Use a Bus Selector block to pick out signals of interest, and use Rate Transition blocks to convert the signals to the desired rates.

Intermixing Composite Signal Types

In this section...

Differences Between Muxes and Buses

Muxes are implemented as indexed vectors, while buses are implemented as structures. Buses can do anything muxes can do and more. See "Using Muxes" on page 9-3 and "Using Buses" on page 9-5 for summaries of the two capabilities and their similarities and differences.

For convenience, muxes and virtual buses can by default 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. 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. You should not mix muxes and virtual buses in new applications, and you should upgrade existing applications to avoid such mixtures. Configuration Parameters diagnostics, Model Advisor checks, and other tools are provided 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 9-7 for information about including one bus within another.

Using Diagnostics for Mixed Composite Signals

Two controls are provided on the **Diagnostics > Connectivity** pane to detect mixed composite signals:
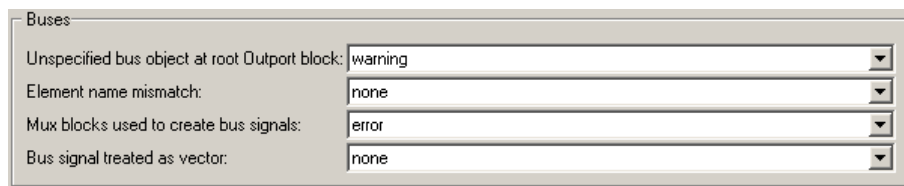
- **Mux blocks used to create bus signals**

- **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 the Model Advisor" on page 3-80. For complete information about the **Connectivity** pane, see "Connectivity Diagnostics Overview".

Mux blocks used to create bus signals

To detect and correct muxes that are used as buses:

1 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Mux blocks used to create bus signals** to warning or error.

2 Set **Configuration Parameters > Diagnostics > Connectivity > Buses > Bus signal treated as vector** to none.

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.
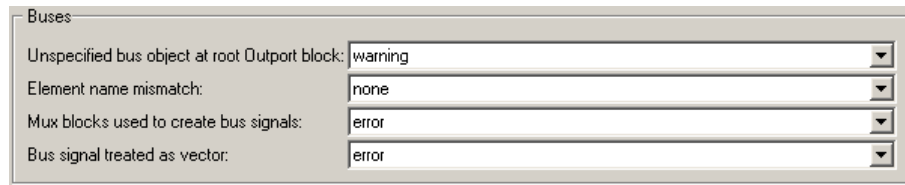
For complete information about this option, see the reference documentation for "Mux blocks used to create bus signals".

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 9-42.

For complete information about this option, see the reference documentation for "Mux blocks used to create bus signals".

Note **Bus signal treated as vector** is disabled unless **Mux blocks used to create bus signals** is set to **error**. Setting **Bus signal treated as vector** to **error** has no effect unless you have previously corrected all cases of Mux blocks used to create buses.

Equivalent Parameter Values

Due to the requirement that **Mux blocks used to create bus signals** be error before **Bus signal treated as vector** is enabled, one parameter, StrictBusMsg, can specify all permutations of the two controls. The parameter can have one of five values. The following table shows these values and the equivalent GUI control settings:

Value of `StrictBusMsg` (API)	Mux blocks used to create bus signals (GUI)	Bus signal treated as vector (GUI)
None	none	none
Warning	warning	none
ErrorLevel1	error	none
WarnOnBusTreatedAsVector	error	warning
ErrorOnBusTreatedAsVector	error	error

Using the Model Advisor for 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 9-42 to correct any errors discovered by the check.

Instructions for using the Model Advisor appear in "Consulting the Model Advisor" on page 3-80.

Correcting Buses Used as Muxes

When you discover a bus signal used as a mux, one answer is to reorganize the model by replacing blocks so that the mixture no longer occurs. Where that is undesirable or unfeasible, the Simulink® software provides two capabilities to address the problem:

- The Bus to Vector block (Signal Attributes library), which you can insert into any bus used implicitly as a mux to explicitly convert the bus to a mux (vector).

- The Simulink.BlockDiagram.addBusToVector function, which automatically inserts Bus to Vector blocks wherever needed.

For example, this figure shows a model that uses a bus as a mux by inputting the bus to a Gain block.
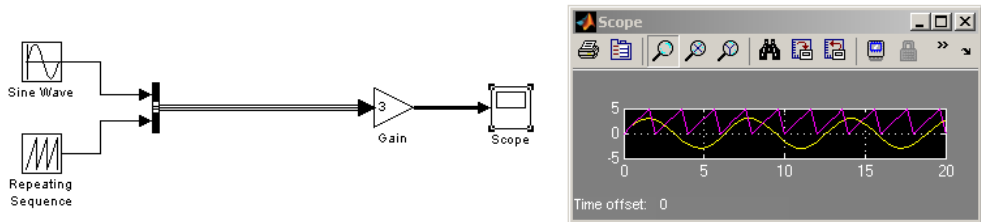
This figure shows the same model, rebuilt after inserting a Bus to Vector block into the bus.
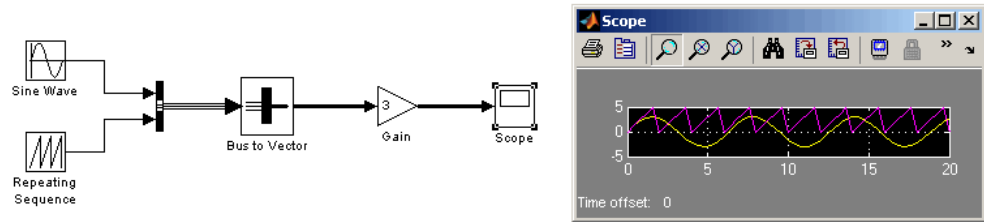
Note that the results of simulation are the same in either case. The Bus to Vector block is virtual, and never affects simulation results, code generation, or performance. For more information, see the reference documentation for the Bus to Vector block and the `Simulink.BlockDiagram.addBusToVector` function.

Bus to Vector Block Backward Compatibility

If you use **Save As** to save a model in a version of the Simulink product before R2007a (V6.6), the following is done:

- Set the `StrictBusMsg` parameter to `error` if its value is `WarnOnBusTreatedAsVector` or `ErrorOnBusTreatedAsVector`.

- Replace each Bus to Vector block in the model with a null subsystem that outputs nothing.

The resulting model specifies strong type checking for Mux blocks used to create buses. Before you can use the model, you must reconnect or otherwise correct each signal that contained a Bus to Vector block but is now interrupted by a null subsystem.

Avoiding 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.

Working with Data

Working with Data Types

About Data Types

The term *data type* refers to the way in which a computer represents numbers in memory. A data type determines the amount of storage allocated to a number, the method used to encode the number's value as a pattern of binary digits, and the operations available for manipulating the type. Most computers provide a choice of data types for representing numbers, each with specific advantages in the areas of precision, dynamic range, performance, and memory usage. To optimize performance, you can specify the data types of variables used in the MATLAB® technical computing environment. The Simulink® software builds on this capability by allowing you to specify the data types of Simulink signals and block parameters.

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

you can dramatically increase performance and decrease the size of the code generated from the model.

Simulink performs extensive checking before and during a simulation to ensure that your model is *typesafe*, that is, that code generated from the model will not overflow or underflow and thus produce incorrect results. Simulink models that use the default data type (double) are inherently typesafe. Thus, if you never plan to generate code from your model or use a nondefault data type in your models, you can skip the remainder of this section.

On the other hand, if you plan to generate code from your models and use nondefault data types, read the remainder of this section carefully, especially the section on data type rules (see "Data Typing Rules" on page 10-25). In that way, you can avoid introducing data type errors that prevent your model from running to completion or simulating at all.

Data Types Supported by Simulink®

Simulink supports all built-in MATLAB data types except int64 and uint64. The term *built-in data type* refers to data types defined by MATLAB itself as opposed to data types defined by MATLAB users. Unless otherwise specified, the term data type in the Simulink documentation refers to built-in data types. The following table lists the built-in MATLAB data types supported by Simulink.

Name	Description
double	Double-precision floating point
single	Single-precision floating point
int8	Signed 8-bit integer
uint8	Unsigned 8-bit integer
int16	Signed 16-bit integer
uint16	Unsigned 16-bit integer
int32	Signed 32-bit integer
uint32	Unsigned 32-bit integer

Besides the built-in types, Simulink defines a `boolean` (1 or 0) type, instances of which are represented internally by `uint8` values.

Many Simulink blocks also support fixed-point data types. See "Blocks — Alphabetical List" in the online documentation for information on the data types supported by specific blocks for parameter and input and output values. If the documentation for a block does not specify a data type, the block inputs or outputs only data of type `double`.

To view a table that summarizes the data types supported by the blocks in the Simulink block libraries, execute the following command at the MATLAB command line:

```
showblockdatatypetable
```

Fixed-Point Data

The Simulink software allows you to create models that use fixed-point numbers to represent signals and parameter values. Use of fixed-point data can reduce the memory requirements and increase the speed of code generated from a model.

To execute a model that uses fixed-point numbers, you must have the Simulink® Fixed Point™ product installed on your system. Specifically, you must have the product to:

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types

- Run a model containing fixed-point data types

- Generate code from a model containing fixed-point data types

- Log the minimum and maximum values produced by a simulation

- Automatically scale the output of a model using the autoscaling tool

If the Simulink Fixed Point product is not installed on your system, you can execute a fixed-point model as a floating-point model by enabling automatic conversion of fixed-point data to floating-point data during simulation. See "Overriding Fixed-Point Specifications" on page 10-5 for details.

If you do not have the Simulink Fixed Point product installed and do not enable automatic conversion of fixed-point to floating-point data, an error occurs if you try to execute a fixed-point model.

Note You do not need the Simulink Fixed Point product to edit a model containing fixed-point blocks, or to use the Data Type Assistant to specify fixed-point data types, as described in "Specifying a Fixed-Point Data Type" on page 10-18 .

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Simulink Fixed Point product. However, even if you do not have the Simulink Fixed Point product, you can use the tool's data type override mode to simulate a model that specifies fixed-point data types.

Data type override mode allows you to share fixed-point models with people in your company who do not have the Simulink Fixed Point product. In data type override mode, the fixed-point values are replaced with floating-point values when executing the model. Such replacement does not affect Fixed-Point Toolbox™ `fi` objects used as fixed-point parameters in your model. However, you can prevent the checkout of a Fixed-Point Toolbox license by setting the `fipref DataTypeOverride` property to `TrueDoubles` (see the Fixed-Point Toolbox documentation).

To simulate a model in data type override mode:

1 From the Simulink **Tools** menu, select **Fixed-Point > Fixed-Point Tool**.

   The Fixed-Point Tool appears.

2 Set the **Logging mode** parameter to `Force off`.

3 Set the **Data type override** parameter to either `True doubles` or `True singles`.

Block Support for Data and Numeric Signal Types

All Simulink blocks accept signals of type double by default. Some blocks
prefer boolean input and others support multiple data types on their inputs.
See Simulink Blocks in *Simulink Reference* for information on the data types
supported by specific blocks for parameter and input and output values. If
the documentation for a block does not specify a data type, the block inputs
or outputs only data of type double.

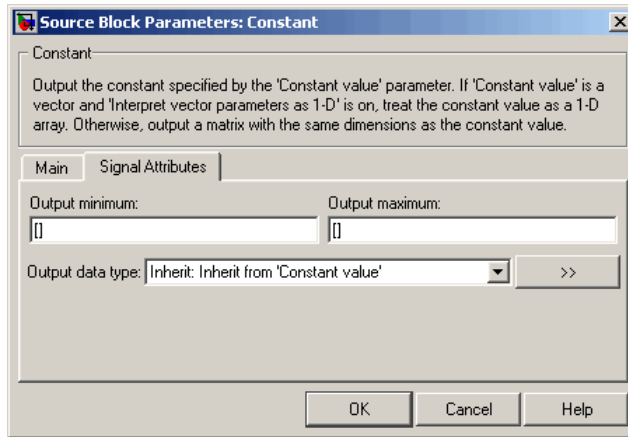## Creating Signals of a Specific Data Type

You can introduce a signal of a specific data type into a model in any of the
following ways:

- Load signal data of the desired type from the MATLAB workspace into your
  model via a root-level Inport block or a From Workspace block.

- Create a Constant block in your model and set its parameter to the desired
  type.

- Use a Data Type Conversion block to convert a signal to the desired data
  type.

Specifying Block Output Data Types

Simulink blocks determine the data type of their outputs by default. Many
blocks allow you to override the default type and explicitly specify an output
data type, using a block parameter that is typically named **Output data
type**. For example, the **Output data type** parameter appears on the **Signal
Attributes** pane of the Constant block dialog box.

Source Block Parameters: Constant

Constant

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

Main   Signal Attributes

Output minimum: Output maximum:

[]   []

Output data type: Inherit: Inherit from 'Constant value'   >>

OK   Cancel   Help

See the following topics for more information:

For Information About...	See...
Valid data type values that you can specify	"Entering Valid Data Type Values" on page 10-8
An assistant that helps you specify valid data type values	"Using the Data Type Assistant" on page 10-14
Specifying valid data type values for multiple blocks simultaneously	"Using the Model Explorer for Batch Editing" on page 10-11

Entering Valid Data Type Values

In general, you can specify the output data type as any of the following:

- A rule that inherits a data type (see "Data Type Inheritance Rules" on page 10-9)

- The name of a built-in data type (see "Built-In Data Types" on page 10-10)

- An expression that evaluates to a data type (see "Data Type Expressions" on page 10-10)

Valid data type values vary among blocks. You can use the pull-down menu associated with a block data type parameter to view the data types that a particular block supports. For example, the **Data type** pull-down menu on the Data Store Memory block dialog box lists the data types that it supports, as shown here.
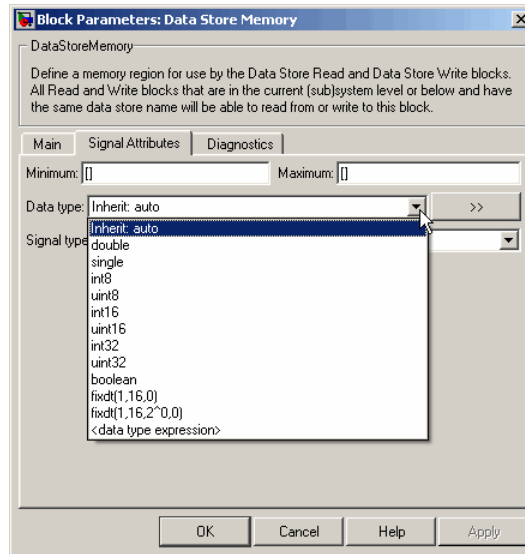
For more information about the data types that a specific block supports, see the documentation for the block in the *Simulink Reference*.

Data Type Inheritance Rules. Blocks can inherit data types from a variety of sources, including signals to which they are connected and particular block parameters. You can specify the value of a data type parameter as a rule that determines how the output signal inherits its data type. To view the inheritance rules that a block supports, use the data type pull-down menu on the block dialog box. The following table lists typical rules that you can select.

Inheritance Rule	Description
`Inherit:  Inherit via back propagation`	Simulink automatically determines the output data type of the block during data type propagation (see "Data Type Propagation" on page 10-24). In this case, the block uses the data type of a downstream block or signal object.
`Inherit:  Same as input`	The block uses the data type of its sole input signal for its output signal.

Inheritance Rule	Description
`Inherit:  Same as first input`	The block uses the data type of its first input signal for its output signal.
`Inherit:  Same as second input`	The block uses the data type of its second input signal for its output signal.
`Inherit:  Inherit via internal rule`	The block uses an internal rule to determine its output data type. The internal rule chooses a data type that optimizes the accuracy and precision of the block output signal.

Built-In Data Types. You can specify the value of a data type parameter as the name of a built-in data type, for example, `single` or `boolean`. To view the built-in data types that a block supports, use the data type pull-down menu on the block dialog box. See "Data Types Supported by Simulink®" on page 10-3 for a list of all built-in data types that are supported.

Data Type Expressions. You can specify the value of a data type parameter as an expression that evaluates to a numeric data type object. Simply enter the expression in the data type field on the block dialog box. In general, enter one of the following expressions:

- **`fixdt` Command**

  Specify the value of a data type parameter as a command that invokes the `fixdt` function. This function allows you to create a `Simulink.NumericType` object that describes a fixed-point or floating-point data type. See the documentation for the `fixdt` function in the *Simulink Reference* for more information.

- **Data Type Object Name**

  Specify the value of a data type parameter as the name of a data object that represents a data type. Simulink data objects that you instantiate from classes, such as `Simulink.NumericType` and `Simulink.AliasType`, simplify the task of making model-wide changes to output data types and

allow you to use custom aliases for data types. See "Working with Data Objects" on page 10-27 for more information about Simulink data objects.

Using the Model Explorer for Batch Editing

Using the Model Explorer (see "The Model Explorer" on page 13-2), you can assign the same output data type to multiple blocks simultaneously. For example, the sldemo_f14 model that comes with the Simulink product contains numerous Gain blocks. Suppose you want to specify the **Output data type** parameter of all the Gain blocks in the model as single. You can achieve this task as follows:

1 Use the Model Explorer search bar (see "Search Bar" on page 13-17) to identify all blocks in the sldemo_f14 model of type Gain.

The Model Explorer **Contents** pane lists all Gain blocks in the model.
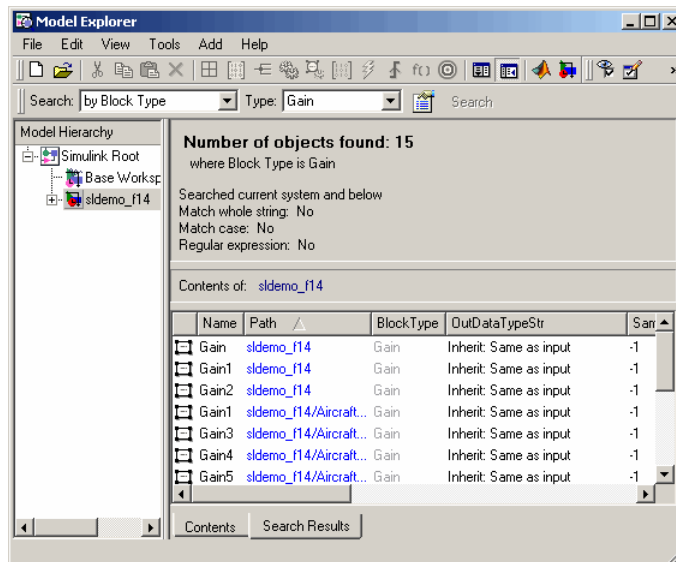
2 In the Model Explorer **Contents** pane, select all the Gain blocks whose **Output data type** parameter you want to specify.

Model Explorer highlights the rows corresponding to your selections.
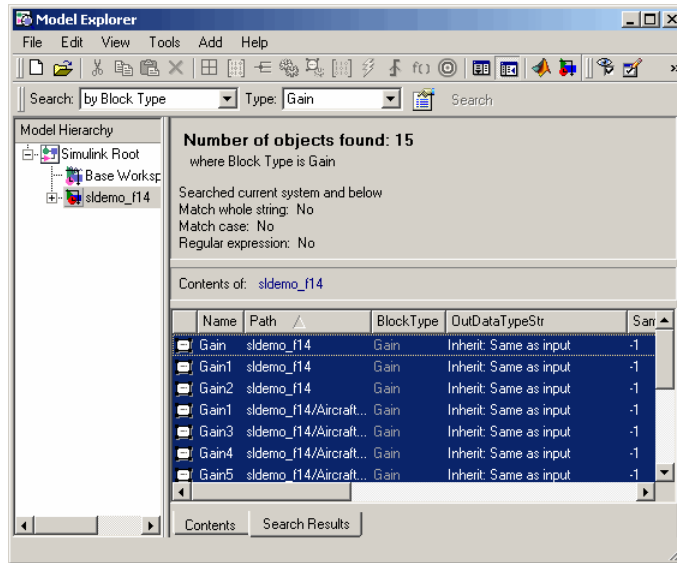
3 In the Model Explorer **Contents** pane, click the data type associated with one of the selected Gain blocks.

Model Explorer displays a pull-down menu with valid data type options.
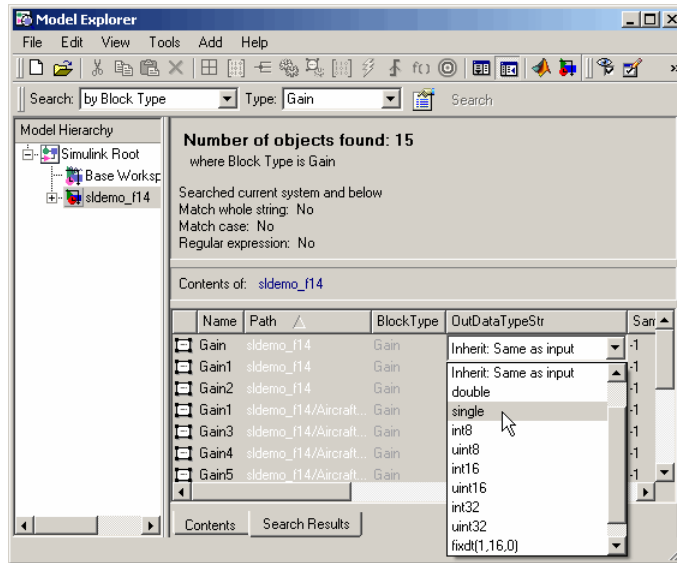
4 In the pull-down menu, enter or select the desired data type, for example, single.

Model Explorer specifies the data type of all selected items as single.
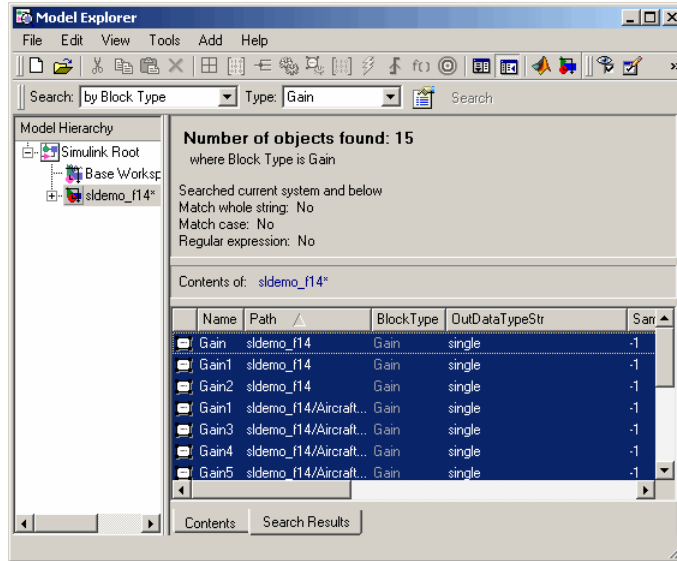
Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool that simplifies the task of specifying data types for blocks and data objects. The assistant appears on block and object dialog boxes, adjacent to parameters that provide data type control, such as the **Output data type** parameter. For example, it appears on the **Signal Attributes** pane of the Constant block dialog box shown here.
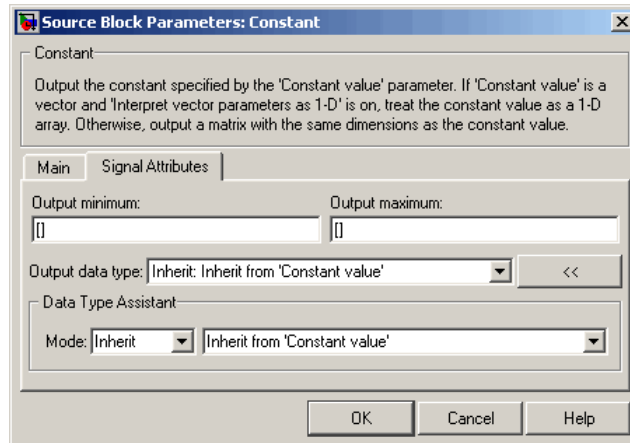
You can selectively show or hide the **Data Type Assistant** by clicking the applicable button:

- Click the **Show data type assistant** button [>>] to display the assistant.

- Click the **Hide data type assistant** button [<<] to hide a visible assistant.

Use the **Data Type Assistant** to specify a data type as follows:

1 In the **Mode** field, select the category of data type that you want to specify. In general, the options include the following:

Mode	Description
Inherit	Inheritance rules for data types
Built in	Built-in data types
Fixed point	Fixed-point data types
Expression	Expressions that evaluate to data types
Bus Object	Simulink.Bus object name, valid only for the Embedded MATLAB™ Function block and the Stateflow® Chart block

The assistant changes dynamically to display different options that correspond to the selected mode. For example, setting **Mode** to Expression causes the Constant block dialog box to appear as follows.
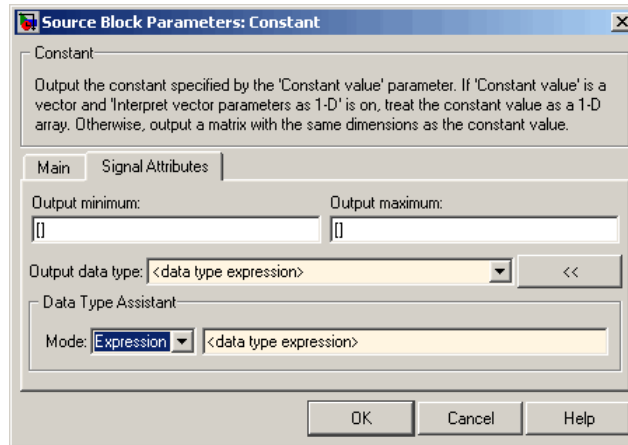

2 In the field that is to the right of the **Mode** field, select or enter a data type.

For example, suppose that you designate the variable myDataType as an alias for a single data type. You create an instance of the Simulink.AliasType class and set its BaseType property by entering the following commands:

```
myDataType = Simulink.AliasType
myDataType.BaseType = 'single'
```

You can use this data type object to specify the output data type of a Constant block. Enter the data type alias name, myDataType, as the value of the expression in the assistant.
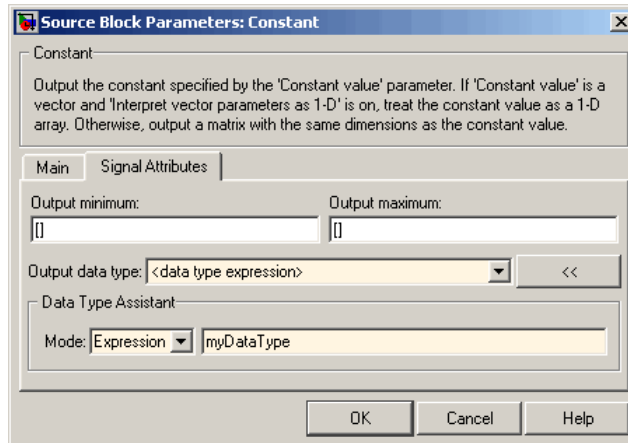
3 Click the **OK** or **Apply** button to apply your changes.

The assistant uses the data type that you specified to populate the associated data type parameter in the block or object dialog box. In the following example, the **Output data type** parameter of the Constant block specifies the same expression that you entered using the assistant.
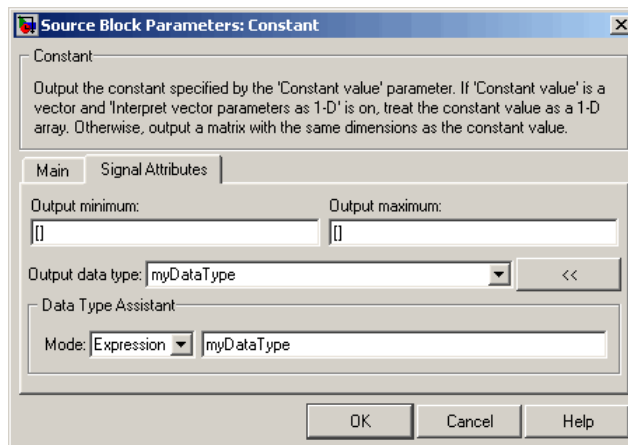

For more information about the data types that you can specify using the **Data Type Assistant**, see "Entering Valid Data Type Values" on page 10-8. See "Specifying Fixed-Point Data Types with the Data Type Assistant" in the

Simulink Fixed Point User's Guide for details about specifying fixed-point data types.

Specifying a Fixed-Point Data Type

When the Data Type Assistant **Mode** is Fixed point, the Data Type Assistant displays fields for specifying information about your fixed-point data type. For a detailed discussion about fixed-point data, see "Fixed-Point Concepts" in the *Simulink Fixed Point User's Guide*. For example, the next figure shows the Block Parameters dialog box for a Gain block, with the **Signal Attributes** tab selected and a fixed-point data type specified.
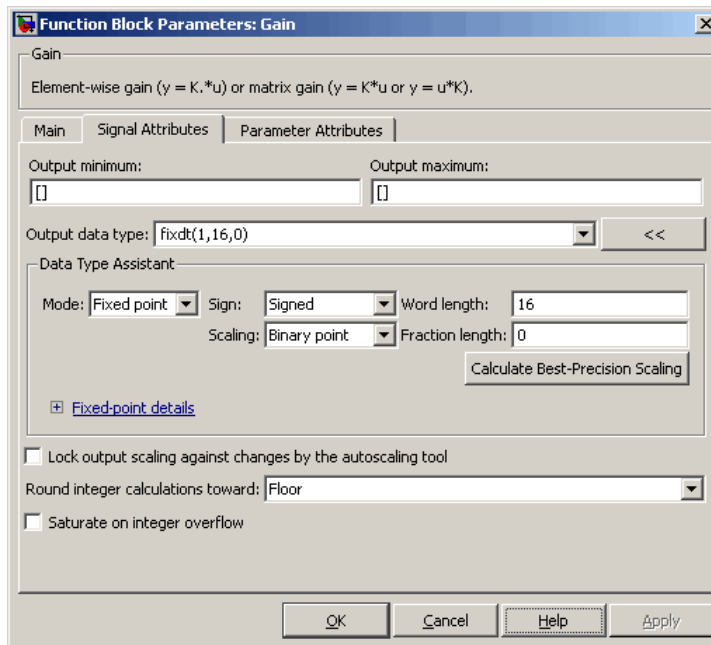
If the **Scaling** is Slope and bias rather than Binary point, the Data Type Assistant displays a **Slope** field and a **Bias** field rather than a **Fraction length** field:
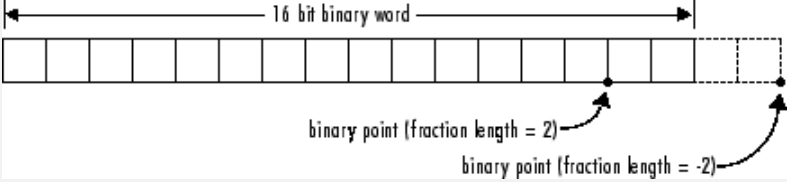
You can use the Data Type Assistant to set these fixed-point properties:

Sign. Specify whether you want the fixed-point data to be Signed or Unsigned. Signed data can represent positive and negative values, but unsigned data represents positive values only. The default setting is Signed.

Word length. Specify the bit size of the word that will hold the quantized integer. Large word sizes represent large values with greater precision than small word sizes. Word length can be any integer between 0 and 32. The default bit size is 16.

Scaling. Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. The default method is Binary point scaling. You can select one of two scaling modes:

Scaling Mode	Description
Binary point	If you select this mode, the Data Type Assistant displays the **Fraction Length** field, which specifies the binary point location.
	Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:
	The default binary point is 0.
Slope and bias	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias**.
	Slope can be any positive real number, and the default slope is 1.0. Bias can be any real number, and the default bias is 0.0. You can enter slope and bias as expressions that contain parameters you define in the MATLAB workspace.

Note Use binary-point scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point scaling are performed with simple bit shifts and eliminate expensive code implementations, which are required for separate slope and bias values.

For more information about fixed-point scaling, see "Scaling" in the *Simulink Fixed Point User's Guide*.

Calculate Best-Precision Scaling. Click this button to calculate best-precision values for both `Binary point` and `Slope and bias` scaling, based on the specified minimum and maximum values. The Simulink software displays the scaling values in the **Fraction Length** field or the **Slope** and **Bias** fields. For more information, see "Constant Scaling for Best Precision" in the *Simulink Fixed Point User's Guide*.

Lock output scaling against changes by the autoscaling tool. Check this box to prevent a Simulink model from replacing the current fixed-point type with a type that the autoscaling tool chooses. See "Automatic Scaling" in the *Simulink Fixed Point User's Guide* for instructions on autoscaling fixed-point data.

Showing Fixed-Point Details. When you specify a fixed-point data type, you can use the **Fixed-point details** subpane to see information about the fixed-point data type that is currently displayed in the Data Type Assistant. To see the subpane, click the expander next to **Fixed-point details** in the Data Type Assistant. The **Fixed-point details** subpane appears at the bottom of the Data Type Assistant:

The rows labeled `Output minimum` and `Output maximum` show the same values that appear in the corresponding **Output minimum** and **Output maximum** fields above the Data Type Assistant. The names of these fields may differ from those shown. For example, a fixed-point block parameter would show **Parameter minimum** and **Parameter maximum**, and the

corresponding **Fixed-point details** rows would be labeled accordingly. See "Checking Signal Ranges" on page 8-26 and "Checking Parameter Values" on page 6-12 for more information.

The rows labeled `Representable minimum`, `Representable maximum`, and `Precision` always appear. These rows show the minimum value, maximum value, and precision that can be represented by the fixed-point data type currently displayed in the Data Type Assistant. See "Fixed-Point Concepts" in the *Simulink Fixed Point User's Guide* for information about these three quantities.

The values displayed by the **Fixed-point details** subpane *do not* automatically update if you click **Calculate Best-Precision Scaling**, or change the range limits, the values that define the fixed-point data type, or anything elsewhere in the model. To update the values shown in the **Fixed-point details** subpane, click **Refresh Details**. The Data Type Assistant then updates or recalculates all values and displays the results.

Clicking **Refresh Details** does not change anything in the model, it only changes the display. Click **OK** or **Apply** to put the displayed values into effect. If the value of a field cannot be known without first compiling the model, the **Fixed-point details** subpane shows the value as `Unknown`.

If any errors occur when you click **Refresh Details**, the **Fixed-point details** subpane shows an error flag on the left of the applicable row, and a description of the error on the right. For example, the next figure shows two errors:

The row labeled Output minimum shows the error Cannot evaluate because evaluating the expression MySymbol, specified in the **Output minimum** field, did not return an appropriate numeric value. When an expression does not evaluate successfully, the **Fixed-point details** subpane displays the unevaluated expression (truncating to 10 characters if necessary to save space) in place of the unavailable value.

To correct the error in this case, you would need to define MySymbol in an accessible workspace to provide an appropriate numeric value. After you clicked **Refresh Details**, the value of MySymbol would appear in place of its unevaluated text, and the error indicator and error description would disappear.

To correct the error shown for Output maximum, you would need to decrease **Output maximum**, increase **Word length**, or decrease **Fraction length** (or some combination of these changes) sufficiently to allow the fixed-point data type to represent the maximum value that it could have.

Other values relevant to a particular block can also appear in the **Fixed-point details** subpane. For example, on a Discrete-Time Integrator block's **Signal Attributes** tab, the subpane could look like this:
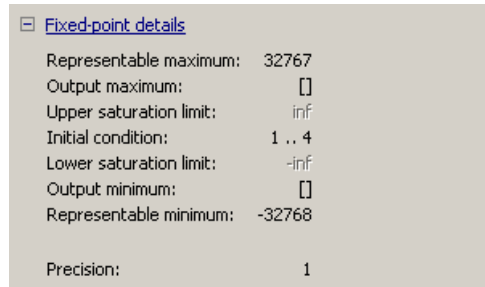
Note that the values displayed for Upper saturation limit and Lower saturation limit are greyed out. This appearance indicates that the corresponding parameters are not currently used by the block. The greyed-out values can be ignored.

Note also that Initial condition displays the value 1..4. The actual value is a vector or matrix whose smallest element is 1 and largest element is 4. To conserve space, the **Fixed-point details** subpane shows only the smallest and largest element of a vector or matrix. An ellipsis (..) replaces the omitted values. The underlying definition of the vector or matrix is unaffected.

Displaying Port Data Types

To display the data types of ports in your model, select **Port Data Types** from the Simulink **Format** menu. The port data type display is not updated when you change the data type of a diagram element. To refresh the display, press **Ctrl+D**.

Data Type Propagation

Whenever you start a simulation, enable display of port data types, or refresh the port data type display, the Simulink software performs a processing step called data type propagation. This step involves determining the types of signals whose type is not otherwise specified and checking the types of signals and input ports to ensure that they do not conflict. If type conflicts arise, an error dialog is displayed that specifies the signal and port whose data types conflict. The signal path that creates the type conflict is also highlighted.

Note You can insert typecasting (data type conversion) blocks in your model to resolve type conflicts. See "Typecasting Signals" on page 10-26 for more information.

Data Typing Rules

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

- Signal data types generally do not affect parameter data types, and vice versa.

  A significant exception to this rule is the Constant block, whose output data type is determined by the data type of its parameter.

- If the output of a block is a function of an input and a parameter, and the input and parameter differ in type, the Simulink software converts the parameter to the input type before computing the output.

- In general, a block outputs the data type that appears at its inputs.

  Significant exceptions include Constant blocks and Data Type Conversion blocks, whose output data types are determined by block parameters.

- Virtual blocks accept signals of any type on their inputs.

  Examples of virtual blocks include Mux and Demux blocks and unconditionally executed subsystems.

- The elements of a signal array connected to a port of a nonvirtual block must be of the same data type.

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

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

- Solver blocks accept only `double` signals.

- Connecting a non-`double` signal to a block disables zero-crossing detection for that block.

Typecasting Signals

An error is displayed whenever it detects that a signal is connected to a block that does not accept the signal's data type. If you want to create such a connection, you must explicitly typecast (convert) the signal to a type that the block does accept. You can use the Data Type Conversion block to perform such conversions.

Working with Data Objects

In this section...

About Data Object Classes

You can create entities called data objects that specify values, data types, tunability, value ranges, and other key attributes of block outputs and parameters. You can create various types of data objects and assign them to workspace variables. You can use the variables in Simulink® dialog boxes to specify parameter and signal attributes. This allows you to make model-wide changes to parameter and signal specifications simply by changing the values of a few variables. With Simulink objects you can parameterize the specification of a model's data attributes. For information on working with specific kinds of data objects, see "Data Object Classes".

Note This section uses the term *data* to refer generically to signals and parameters.

The Simulink software uses objects called data classes to define the properties of specific types of data objects. The classes also define functions, called

methods, for creating and manipulating instances of particular types of objects. A set of built-in classes are provided for specifying specific types of attributes (see "Data Object Classes" for information on these built-in classes). Some of The MathWorks™ products based on Simulink, such as the Real-Time Workshop® product, also provide classes for specifying data attributes specific to their applications. See the documentation for those products for information on the classes they provide. You can also create subclasses of some of these built-in classes to specify attributes specific to your applications (see "Subclassing Simulink® Data Classes" on page 10-41).

Memory structures called *packages* are used to store the code and data that implement data classes. The classes provided by the Simulink software reside in the Simulink package. Classes provided by products based on Simulink reside in packages provided by those products. You can create your own packages for storing the classes that you define.

Class Naming Convention

Simulink uses dot notation to name classes:

```
PACKAGE.CLASS
```

where CLASS is the name of the class and PACKAGE is the name of the package to which the class belongs, for example, `Simulink.Parameter`. This notation allows you to create and reference identically named classes that belong to different packages. In this notation, the name of the package is said to qualify the name of the class.

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

About Data Object Methods

Data classes define functions, called methods, for creating and manipulating the objects that they define. A class may define any of the following kinds of methods.

Dynamic Methods

A dynamic method is a method whose identity depends on its name and the class of an object specified implicitly or explicitly as its first argument. You can use either function or dot notation to specify this object, which must be an instance of the class that defines the method or an instance of a subclass of the class that defines the method. For example, suppose class A defines a method called `setName` that assigns a name to an instance of A. Further, suppose the MATLAB® workspace contains an instance of A assigned to the variable `obj`. Then, you can use either of the following statements to assign the name `'foo'` to `obj`:

```
obj.setName('foo');
setName(obj, 'foo');
```

A class may define a set of methods having the same name as a method defined by one of its super classes. In this case, the method defined by the subclass overrides the behavior of the method defined by the parent class. The Simulink software determines which method to invoke at runtime from the class of the object that you specify as its first or implicit argument. Hence, the term dynamic method.

Note Most Simulink data object methods are dynamic methods. Unless the documentation for a method specifies otherwise, you can assume that a method is a dynamic method.

Static Methods

A static method is a method whose identity depends only on its name and hence cannot change at runtime. To invoke a static method, use its fully qualified name, which includes the name of the class that defines it followed by the name of the method itself. For example, `Simulink.ModelAdvisor` class defines a static method named `getModelAdvisor`. The fully qualified name of this static method is `Simulink.ModelAdvisor.getModelAdvisor`. The following example illustrates invocation of a static method.

```
ma = Simulink.ModelAdvisor.getModelAdvisor('vdp');
```

Constructors

Every data class defines a method for creating instances of that class. The name of the method is the same as the name of the class. For example, the name of the `Simulink.Parameter` class's constructor is `Simulink.Parameter`. The constructors defined by Simulink data classes take no arguments.

The value returned by a constructor depends on whether its class is a handle class or a value class. The constructor for a handle class returns a handle to the instance that it creates if the class of the instance is a handle class; otherwise, it returns the instance itself (see "Handle Versus Value Classes" on page 10-34).

Using the Model Explorer to Create Data Objects

You can use the Model Explorer (see "The Model Explorer" on page 13-2) as well as MATLAB commands to create data objects. To use the Model Explorer,

1 Select the workspace in which you want to create the object in the Model Explorer's **Model Hierarchy** pane.
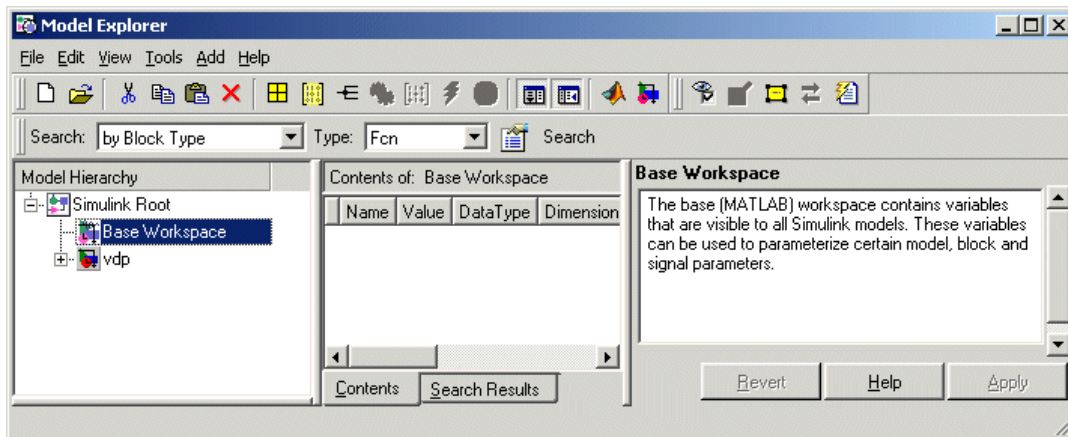
Only `Simulink.Parameter` and `Simulink.Signal` objects for which the storage class is set to `Auto` can reside in a model workspace. You must create all other Simulink data objects in the base MATLAB workspace to ensure the objects are unique within the global Simulink context and accessible to all models.


Note Subclasses of `Simulink.Parameter` and `Simulink.Signal` classes, such as `mpt.Parameter` and `mpt.Signal` objects (Real-Time Workshop® Embedded Coder™ license required), can reside in a model workspace only if their storage class is set to `Auto`.

---

2 Select the type of the object that you want to create (for example, **Simulink Parameter** or **Simulink Signal**) from the Model Explorer's **Add** menu or from its toolbar.The Simulink software creates the object, assigns it to a variable in the selected workspace, and displays its properties in the Model Explorer's **Contents** and **Dialog** panes.
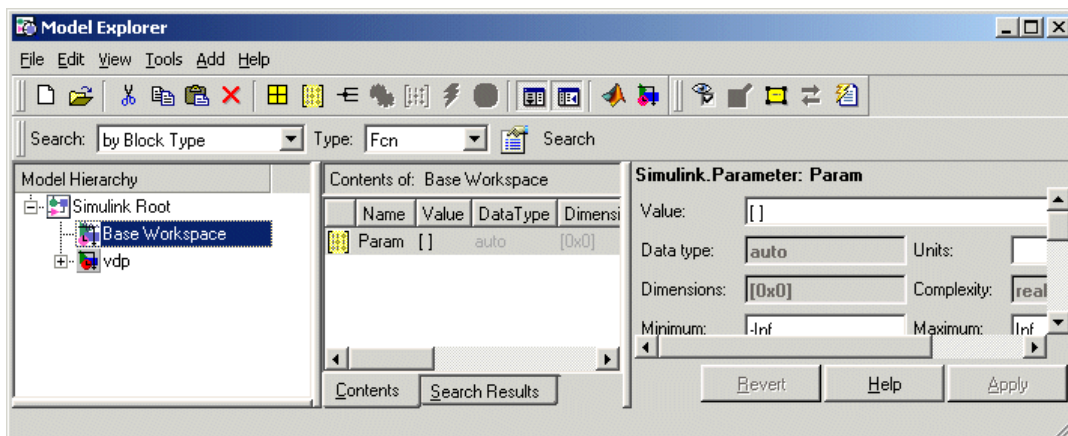
If the type of object you want to create does not appear on the **Add** menu, select **Find Custom** from the menu. The MATLAB path is searched for all data object classes derived from Simulink class on the MATLAB path, including types that you have created, and displays the result in a dialog box.
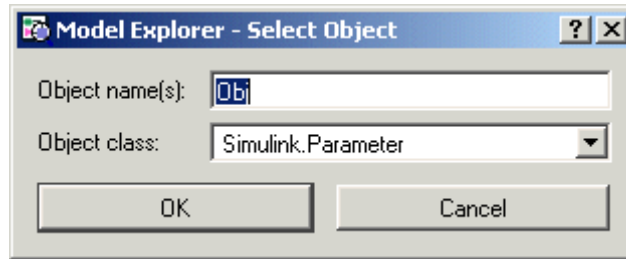
3 Select the type of object (or objects) that you want to create from the **Object class** list and enter the names of the workspace variables to which you want the objects to be assigned in the **Object name(s)** field. Simulink creates the specified objects and displays them in the Model Explorer's **Contents** pane.

About Object Properties

Object properties are variables associated with an object that specify properties of the entity that the object represents, for example, the size of a data type. The object's class defines the names, value types, default values, and valid value ranges of the object's properties.
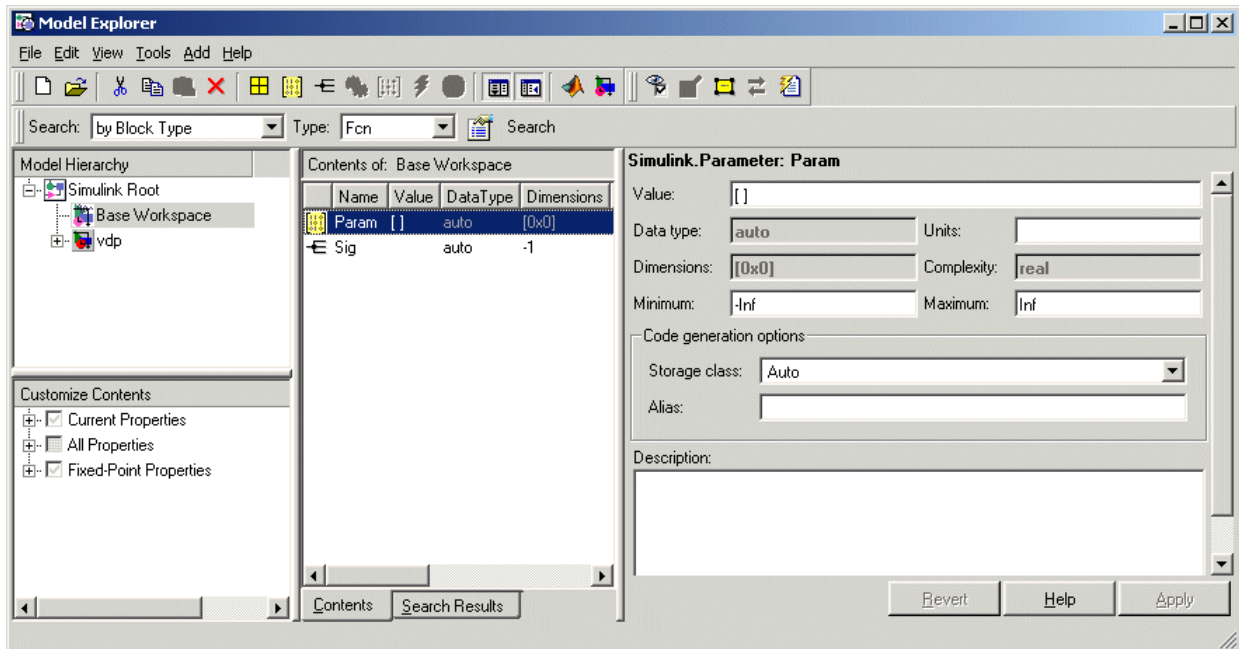
Changing Object Properties

You can use either the Model Explorer (see "Using the Model Explorer to Change an Object's Properties" on page 10-32) or MATLAB commands to change a data object's properties (see "Using MATLAB® Commands to Change Workspace Data" on page 3-63).

Using the Model Explorer to Change an Object's Properties

To use the Model Explorer to change an object's properties, select the workspace that contains the object in the Model Explorer's **Model Hierarchy** pane. Then select the object in the Model Explorer's **Contents** pane.

The Model Explorer displays the object's property dialog box in its **Dialog** pane (if the pane is visible).

You can configure the Model Explorer to display some or all of the object's properties in the **Contents** pane (see "Customizing the Contents Pane" on page 13-9). 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 10-34), 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 10-30) 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: [O O]
```

Copying Handle Classes

Use the copy method of a handle class to create copies of instances of that class. For example, Simulink.ConfigSet is a handle class that represents model configuration sets. The following code creates a copy of the current model's active configuration set and attaches it to the model as an alternate configuration geared to model development.

```
activeConfig = getActiveConfigSet(gcs);
develConfig = activeConfig.copy;
develConfig.Name = 'develConfig';
attachConfigSet(gcs, develConfig);
```

Comparing Data Objects

Simulink data objects provide a method, named isContentEqual, that determines whether object property values are equal. This method compares the property values of one object with those belonging to another object and returns true (1) if all of the values are the same or false (0) otherwise. For example, the following code instantiates two signal objects (A and B) and specifies values for particular properties.

```
A = Simulink.Signal;
B = Simulink.Signal;
A.DataType = 'int8';
B.DataType = 'int8';
A.InitialValue = '1.5';
B.InitialValue = '1.5';
```

Afterward, use the isContentEqual method to verify that the object properties of A and B are equal.

```
>> result = A.isContentEqual(B)

result =

     1
```

Saving and Loading Data Objects

You can use the save command to save data objects in a MAT-file and the load command to restore them to the MATLAB workspace in the same or a later session. Definitions of the classes of saved objects must exist on the

MATLAB path for them to be restored. If the class of a saved object acquires new properties after the object is saved, Simulink adds the new properties to the restored version of the object. If the class loses properties after the object is saved, only the properties that remain are restored.

Using Data Objects in Simulink® Models

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

Creating Persistent Data Objects

To create parameter and signal objects that persist across Simulink sessions, first write a script that creates the objects or create the objects with the Simulink **Data Class Designer** (see "Subclassing Simulink® Data Classes" on page 10-41) or at the command line and save them in a MAT-file (see "Saving and Loading Data Objects" on page 10-36). Then use either the script or a load command as the `PreLoadFcn` callback routine for the model that uses the objects. For example, suppose you save the data objects in a file named `data_objects.mat` and the model to which they apply is open and active. Then, entering the following command

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

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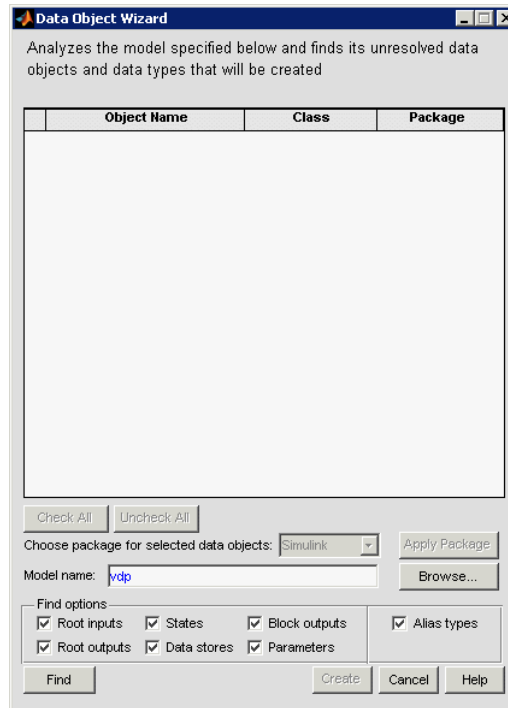
Data Object Wizard

The Data Object Wizard allows you to determine quickly which model data are not associated with data objects and to create and associate data objects with the data.

To use the wizard to create data objects:

1 Select **Tools > Data Object Wizard** from the Model Editor's tool bar.

The Data Object Wizard appears.

2 Enter, if necessary, the name of the model you want to search in the wizard's **Model name** field.

By default the wizard displays the name of the model from which you opened the wizard. You can enter the name of another model in this field. If the model is not open, the wizard opens the model.

3 In **Find options**, uncheck any of the data object types that you want the search to ignore.

The search options include:

Option	Description
Root inputs	Named signals from root-level input ports

Option	Description
Root outputs	Named signals from root-level output ports
States	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 3-74)
Block outputs	Named signals emitted by non-root-level blocks.
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 "Sharing Simulink Parameters with Stateflow Charts" 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 theReal-Time Workshop Embedded Coder documentation for more information.

4 Click the wizard's **Find** button.

The wizard displays the search results in the data objects table.
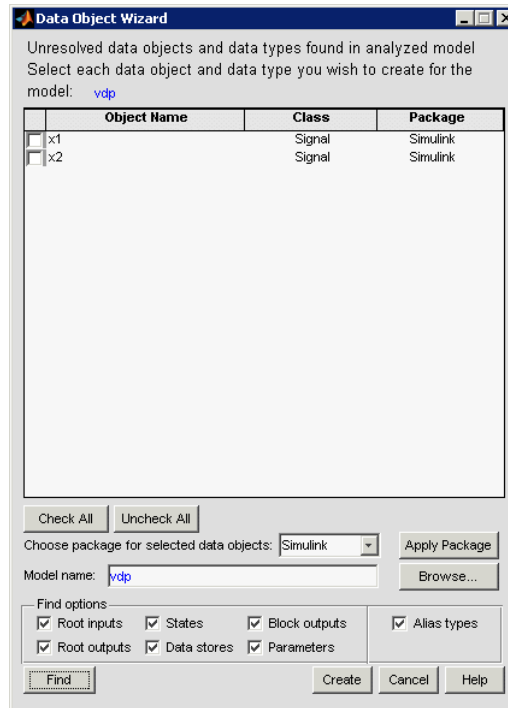
5 Check the data for which you want the wizard to create data objects.

6 If you want the wizard to use data object classes from a package other than the Simulink standard class package to create the data objects, select the package from the **Choose package for selected data objects** list and then select **Apply Package** to confirm your choice.

7 Click **Create**.

The wizard creates data objects of the appropriate class for the data selected in the search results table.

Note Use the Model Explorer to view and edit the created data objects.

Subclassing Simulink® Data Classes

In this section...

About the Data Class Designer

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 use the **Data Class Designer** to change the definitions of classes that it created, for example, to add or remove properties. See "Working with Data Structures" for information on storage classes. You can use the **Data Class Designer** to create custom storage classes. See "Custom Storage Classes" in the Real-Time Workshop® Embedded Coder™ documentation for details. The **Data Class Designer** is not 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.
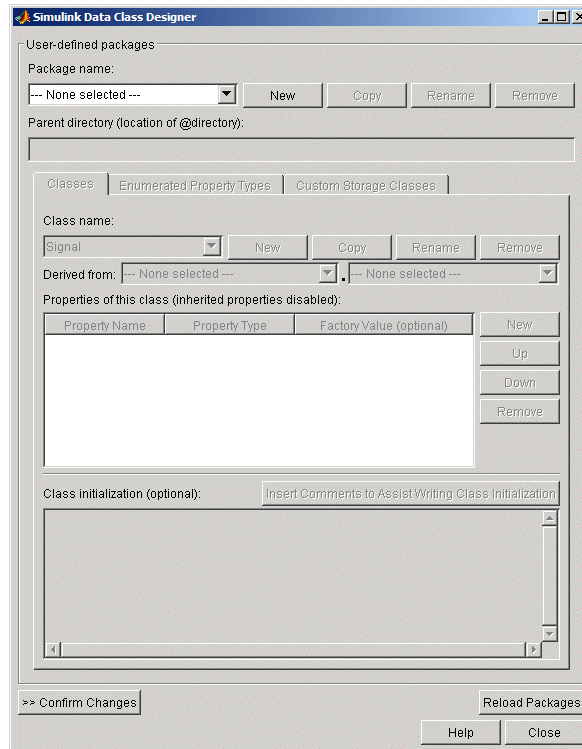
2 Select the name of the package in which you want to create the class from the **Package name** list.

Do not create a class in any of the Simulink built-in packages, i.e., packages in *matlabroot*/toolbox/simulink, or any directory under *matlabroot*. See "Creating a Class Package" on page 10-52 for information on creating your own class packages.

3 Click the **New** button on the **Classes** pane of the **Data Class Designer** dialog box.

4 Enter the name of the new class in the **Class name** field on the **Classes** pane.

Note The name of the new class must be unique in the package to which the new class belongs. Class names are case sensitive. For example, Signal and signal is considered to be names of different classes.

---

5 Press **Enter** or click **OK** on the **Classes** pane to create the specified class in memory.

6 Select a parent class for the new class (see "Specifying a Parent for a Class" on page 10-45).

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 10-46).

8 If necessary, create initialization code for the new class (see "Creating Initialization Code" on page 10-51).

9 Click **Confirm Changes**.

Simulink displays the **Confirm Changes** pane.
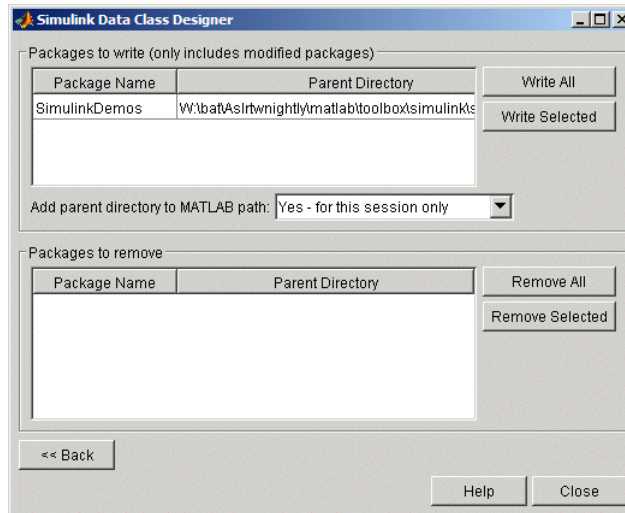
10 Click **Write All** or select the package containing the new class definition and click **Write Selected** to save the new class definition.

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**. The Simulink software creates a copy of the class under a slightly different name. Edit the name, if desired, click **Confirm Changes**, and click **Write All** or, after selecting the appropriate package, **Write Selected** to save the new class.

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.

> **Note** You cannot modify an instantiated Simulink data class. If you try
> to modify a class that has already been instantiated during the current
> MATLAB® session, an error message informs you that you need to restart
> MATLAB if you want to modify this package.

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**. The class is removed from the
in-memory definition of the class. Save the package that formerly contained
the class.

Specifying a Parent for a Class

To specify a parent for a class:

1 Select the name of the class from the **Class name** field on the **Classes**
pane.

2 Select the package name of the parent class from the left-hand **Derived
from** list box.
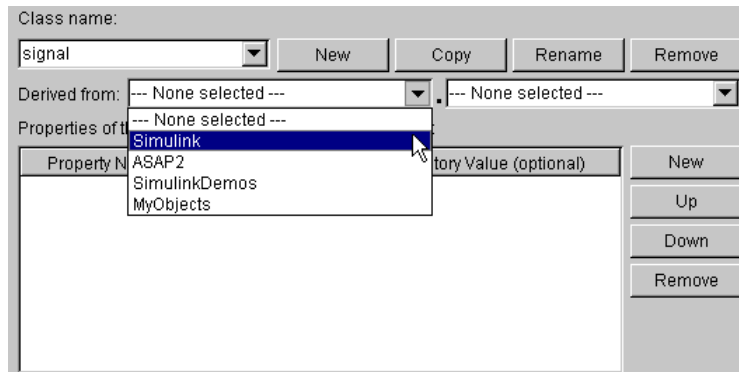
3 Select the parent class from the right-hand **Derived from** list.
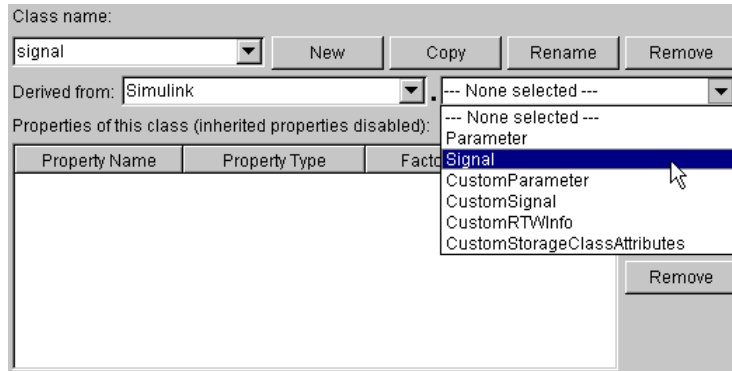
The properties of the selected class derived from the parent class are displayed in the **Properties of this class** field.

The inherited properties are grayed to indicate that they cannot be redefined by the child class.

4 Save the package containing the class.

Defining Class Properties

To add a property to a class:

1 Select the name of the class from the **Class name** field on the **Classes** pane.

> **Note** You cannot modify an instantiated Simulink data class. If you try to modify a class that has already been instantiated during the current MATLAB session, an error message informs you that you need to restart MATLAB if you want to modify this package.

2 Click the **New** button next to the **Properties of this class** field on the **Classes** pane.

A property is created with a default name and value and displays the property in the **Properties of this class** field.
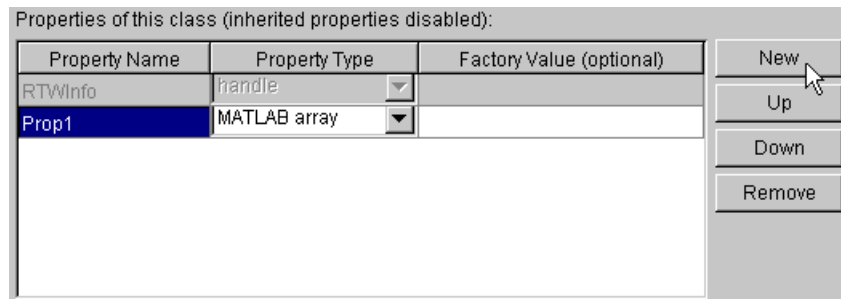
3 Enter a name for the new property in the **Property Name** column.

> **Note** The property name must be unique to the class. Unlike class names, property names are not case sensitive. For example, Value and value are treated 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 10-48).

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 10-51 for more information).

The following rules apply to entering factory values for properties:

- Do not use quotation marks when entering the value of a string property. The value that you enter is treated as a literal string.

- The value of a MATLAB array property can be any expression that evaluates to an array, cell array, structure, or object. Enter the expression exactly as you would enter the value on the command line, for example, [0 1; 1 0]. The expression that you enter is evaluated to check its validity. A warning is displayed if evaluating the expression results in an error. Regardless of whether an evaluation error occurs, Simulink stores the expression as the factory value of the property. This is because an expression that is invalid at define time might be valid at run-time.

- You can enter any expression that evaluates to a numeric value as the value of a double or int32 property. The expression is evaluated and the result stored as the property's factory value.

6 Save the package containing the class with new or changed properties.

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**.
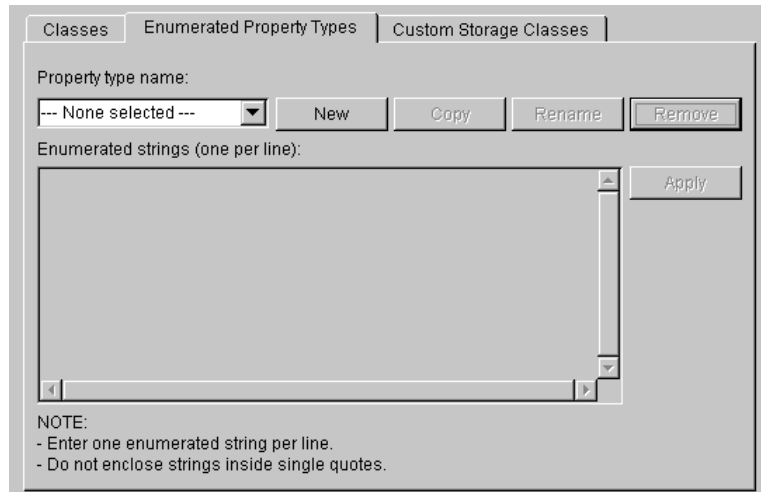
2 Click the **New** button next to the **Property type name** field.

An enumerated type is created with a default name.

**3** Change the default name in the **Property type name** field to the desired name for the property.

The currently selected package defines an enumerated property type and the type can be referenced only in the package that defines it. However, the name of the enumerated property type must be globally unique. There cannot be any other built-in or user-defined enumerated property with the same name. An error is displayed if you enter the name of an existing built-in or user-defined enumerated property for the new property.

4 Click the **OK** button.

The new property is created in memory and the **Enumerated strings** field on the **Enumerated Property Types** pane is enabled .

5 Enter the permissible values for the 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`.
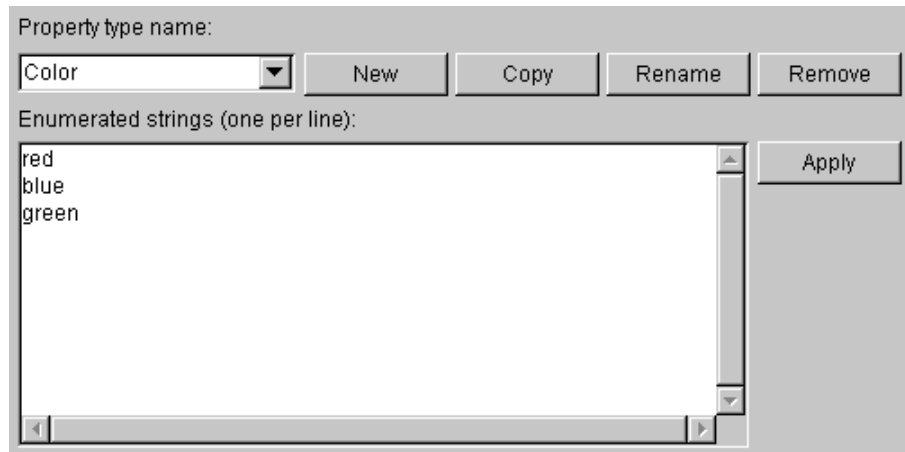
6 Click **Apply** to save the changes in memory.

7 Click **Confirm changes**. Then click **Write all** to save this change.

You can also use the **Enumerated Property Type** pane to copy, rename, and remove enumerated property types.

- Click the **Copy** button to copy the currently selected property type. A new property that has a new name, but has the same value set as the original property is created.

- Click the **Rename** button to rename the currently selected property type. The **Property name** field becomes editable. Edit the field to reflect the new name.

- Click the **Remove** button to remove the currently selected property.

Always save the package containing the modified enumerated property type.

Note You must restart the MATLAB software if you modify, add, or remove enumerated types from a class you have already instantiated.

Creating Initialization Code

You can specify code to be executed when the Simulink software creates an instance of a data object class. To specify initialization code for a class, select the class from the **Class name** field of the **Data Class Designer** and enter the initialization code in the **Class initialization** field.

The **Data Class Designer** inserts the code that you enter in the **Class initialization** field in the class instantiation function of the corresponding class. This function is invoked when an instance of this class is created. The class instantiation function has the form

```
function h = ClassName(varargin)
```

where h is the handle to the object that is created and varargin is a cell array that contains the function's input arguments.

By entering the appropriate code in the **Data Class Designer**, you can cause the instantiation function to perform such initialization operations as

- Error checking
- Loading information from data files
- Overriding factory values
- Initializing properties to user-specified values

For example, suppose you want to let a user initialize the ParamName property of instances of a class named MyPackage.Parameter. The user does this by passing the initial value of the ParamName property to the class constructor:

```
Kp = MyPackage.Parameter('Kp');
```

10-51

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 ▼	New	Copy	Rename	Remove

Parent directory (location of @directory):

c:\matlab\toolbox\simulink\simdemos

A default package name is displayed in the **Package name** field.

Package name:				
NewPackage1 ▼	OK	Cancel	Rename	Remove

2 Edit the **Package name** field to contain the package name that you want.

Package name:				
MyData ▼	OK	Cancel	Rename	Remove

3 Click **OK** to create the new package in memory.

4 In the package **Parent directory** field, enter the path of the directory where you want Simulink to create the new package.

> **Note** Do not create class package directories under *matlabroot*. Packages in these directories are treated as built-in and will not be visible in the Data Class Designer.

Package name:

| MyData | ▼ | New | Copy | Rename | Remove |

Parent directory (location of @directory):

| d:\Work |

The Simulink software creates the specified directory, if it does not already exist, when you save the package to your file system in the succeeding steps.

5 Click the **Confirm changes** button on the **Data Class Designer**.

The **Packages to write** panel is displayed.

Packages to write (only includes modified packages)

Package Name	Parent Directory	
SimulinkDemos	W:\bat\AslrtwnightlY\matlab\toolbox\simulink\	Write All
MyData	D:\Work	Write Selected

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. The Simulink software creates a copy of the package under a slightly different name. Edit the new name, if desired, and click **OK** to create the package in memory. Then save the package to make it permanent.

Renaming a package

To rename a package, select the package and click the **Rename** button next to the **Package name** field. The field becomes editable. Edit the field to reflect the new name. Save the renamed package.

Removing a package

To remove a package, select the package and click the **Remove** button next to the **Package name** field to remove the package from memory. Click the **Confirm changes** button to display the **Packages to remove** panel. Select the package and click **Remove selected** to remove the package from your file system or click **Remove all** to remove all packages that you have removed from memory from your file system as well.

Associating User Data with Blocks

You can use the set_param command to associate your own data with a block. For example, the following command associates the value of the variable mydata with the currently selected block.

```
set_param(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 saves 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.

11

Working with Lookup Tables

About Lookup Table Blocks

A *lookup table* block uses an array of data to map input values to output values, approximating a mathematical function. Given input values, the Simulink® software performs a "lookup" operation to retrieve the corresponding output values from the table. If the lookup table does not define the input values, the block estimates the output values based on nearby table values.

The following example illustrates a one-dimensional lookup table that approximates the function $y = x^3$. The lookup table defines its output (y) data discretely over the input (x) range [-3, 3]. The following table and graph illustrate the input/output relationship:

x	-3	-2	-1	0	1	2	3
y	-27	-8	-1	0	1	8	27

● — Breakpoint Value
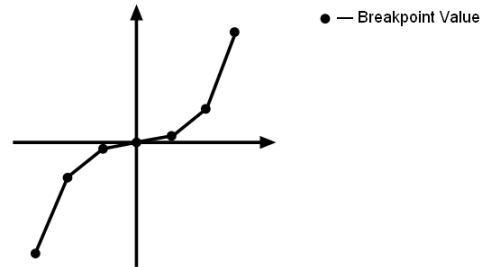
An input of -2 enables the table to look up and retrieve the corresponding output value (-8). Likewise, the lookup table outputs 27 in response to an input of 3.

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.

Linearly interpolated value

x	-3	-2	-1.5	-1	0	1	2	3
y	-27	-8	-4.5	-1	0	1	8	27

● — Breakpoint Value
○ — Interpolated Value
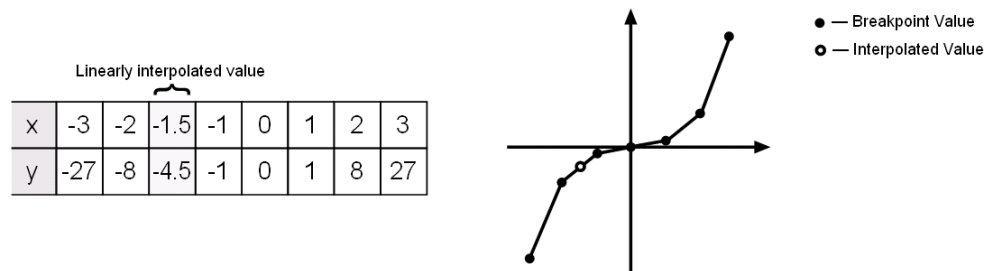
Similarly, although the lookup table does not include data for x values beyond the range of [-3, 3], the block can extrapolate values using a pair of data points at either end of the table. Given an input value of 4, the lookup table block linearly extrapolates the nearest data points (2, 8) and (3, 27) to estimate an output value of 46.

Linearly extrapolated value

x	-3	-2	-1	0	1	2	3	4
y	-27	-8	-1	0	1	8	27	46

● — Breakpoint Value
○ — Extrapolated Value
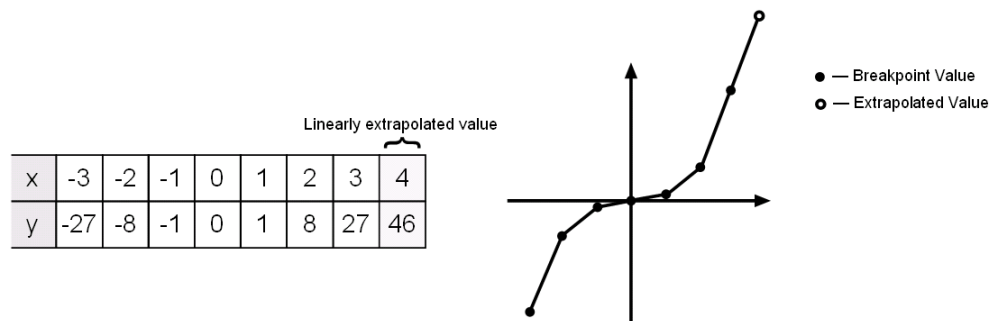
Since table lookups and simple estimations can be faster than mathematical function evaluations, using lookup table blocks may result in speed gains when simulating a model. Consider using lookup tables in lieu of mathematical function evaluations when

- An analytical expression is expensive to compute

- No analytical expression exists, but the relationship has been determined empirically

The Simulink software provides a broad assortment of lookup table blocks, each geared for a particular type of application. The sections that follow outline the different offerings, suggest how to choose the lookup table best

suited to your application, and explain how to interact with the various lookup table blocks.

Anatomy of a Lookup Table

The following figure illustrates the anatomy of a two-dimensional lookup table. Vectors or *breakpoint data sets* and an array, referred to as *table data*, constitute the lookup table.
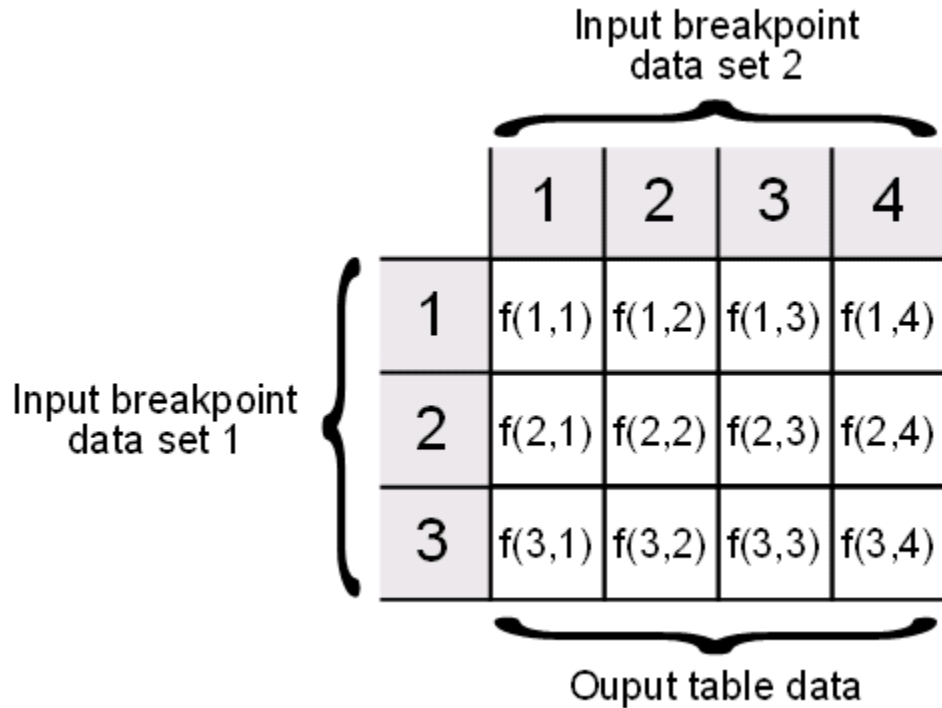
Each breakpoint data set is an index of input values for a particular dimension of the lookup table. The array of table data serves as a sampled representation of a function evaluated at the breakpoint values. Lookup table blocks use breakpoint data sets to relate a table's input values to the output values that it returns.

Lookup Tables Block Library

Several lookup table blocks are provided in the Lookup Tables block library shown here:
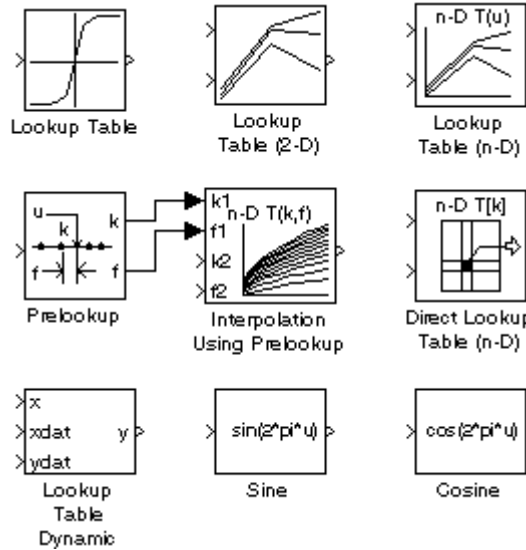
The following table summarizes the purpose of each block in the library:

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

Data Set Dimensionality

In some cases, the dimensions of your data set dictate which of the lookup table blocks is right for your application. If you are approximating a one-dimensional function, consider using either the Lookup Table or Lookup Table Dynamic block. If you are approximating a two-dimensional function, consider the Lookup Table (2-D) block. Blocks such as the Lookup Table (n-D) and Direct Lookup Table (n-D) allow you to approximate an *N*-dimensional function of even higher order.

Data Set Numeric and Data Types

The numeric and data types of your data set influence the decision of which lookup table block is most appropriate. Although all lookup table blocks support real numbers, the Direct Lookup Table (n-D) and Lookup Table (n-D) blocks are the only lookup table blocks that support complex table data. Most lookup table blocks support integer and fixed-point data in addition to `double` and `single` data types.

Data Accuracy and Smoothness

The desired accuracy and smoothness of the data returned by a lookup table determine which of the blocks should be used. Most blocks provide options to perform interpolation and extrapolation, improving the accuracy of values that fall between or outside of the table data, respectively. For instance, the Lookup Table, Lookup Table (2-D), and Lookup Table Dynamic blocks

perform linear interpolation and extrapolation, while the Lookup Table (n-D) block performs either linear or cubic spline interpolation and extrapolation. In contrast, the Direct Lookup Table (n-D) block performs table lookups without any interpolation and extrapolation. You can achieve a mix of interpolation and extrapolation methods by using the Prelookup block with the Interpolation Using Prelookup block.

Dynamics of Table Inputs

The dynamics of the lookup table inputs impact which of the lookup table blocks is ideal for your application. The blocks use a variety of index search methods to relate the lookup table inputs to the table's breakpoint data sets. Most of the lookup table blocks offer a binary search algorithm, which performs well if the inputs change significantly from one time step to the next. The Lookup Table (n-D) and Prelookup blocks offer a linear search algorithm. Using this algorithm with the option that resumes searching from the previous result performs well if the inputs change slowly. Certain lookup table blocks also provide a search algorithm that is tailored for breakpoint data sets composed of evenly spaced breakpoints. Note that you can achieve a mix of index search methods by using the Prelookup block with the Interpolation Using Prelookup block.

Efficiency of Performance

When the efficiency with which lookup tables operate is important, consider using the Prelookup block with the Interpolation Using Prelookup block. These blocks separate the table lookup process into two components — an index search that relates inputs to the table data, followed by an interpolation and extrapolation stage that computes outputs. These blocks enable you to perform a single index search and then reuse the results to look up data in multiple tables. Also, the Interpolation Using Prelookup block can perform sub-table selection, in which it interpolates a portion of the table data instead of the entire table. For example, if your 3-D table data constitutes a stack of 2-D tables to be interpolated, the block allows you to specify an input used to select and interpolate the 2-D tables. These features make table lookup operations more efficient, reducing computational effort and thus simulation time.

Summary of Lookup Table Block Features

The following table summarizes many features of lookup table blocks in Simulink®. Use the table to identify features that correspond to particular lookup table blocks, then select the block that best meets your requirements.

Feature	Lookup Table	Lookup Table (2-D)	Lookup Table Dynamic	Lookup Table (n-D)	Direct Lookup Table (n-D)	Prelookup	Interp. Using Prelookup
Interpolation Methods							
Flat (none)	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline				•			
Extrapolation Methods							
Clipping	•	•	•	•	•	•	•
Linear	•	•	•	•		•	•
Cubic spline				•			
Numeric & Data Type Support							
Complex				•	•		
Double, Single	•	•	•	•	•	•	•
Integer	•	•	•	•	•	•	•
Fixed-point	•	•	•	•		•	•
Index Search Methods							
Binary	•	•	•	•		•	
Linear				•		•	
Evenly spaced points				•	•	•	
Start at previous index				•		•	
Miscellaneous							
Sub-table selection					•		•
Dynamic table data			•		•		
Input range checking				•	•	•	•

Entering Breakpoints and Table Data

In this section...

Entering Data in a 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.
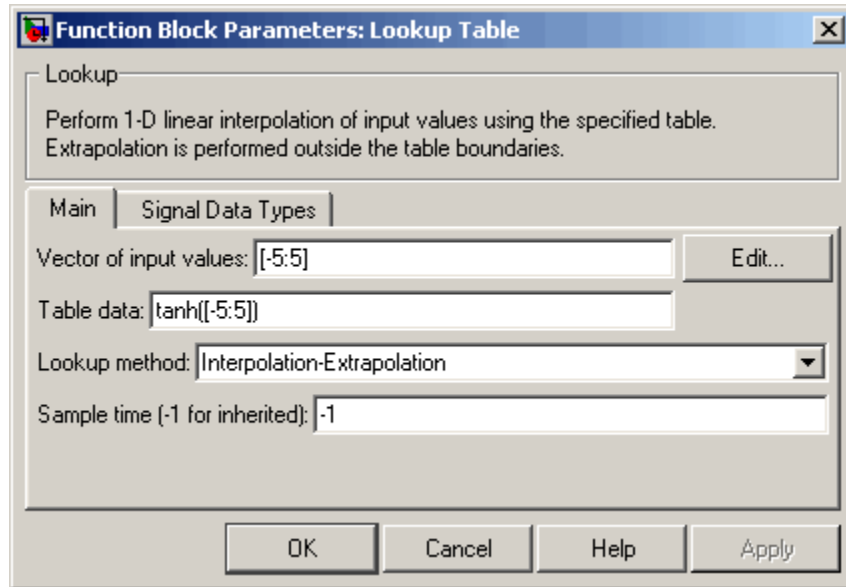
The dialog box displays the default data associated with the block.

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:
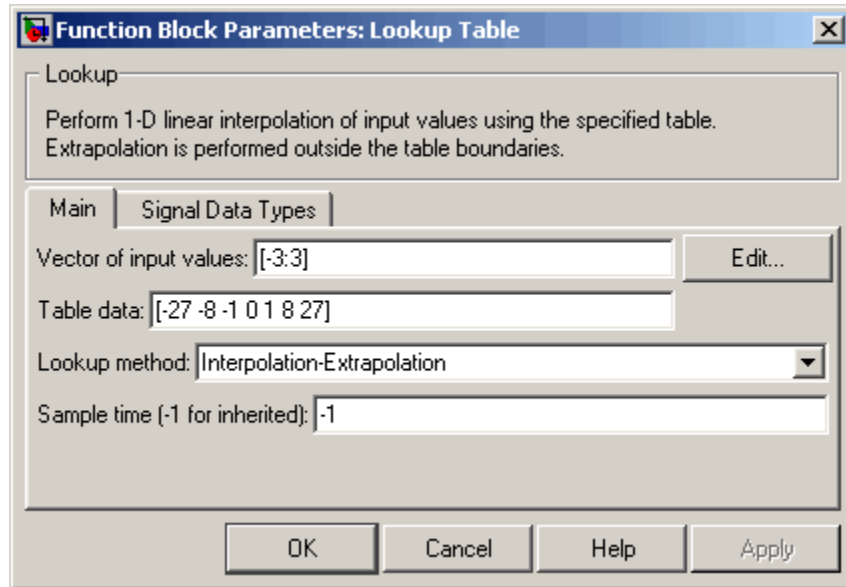
4 Click the **Apply** button to apply the changes or the **OK** button to apply the changes and close the dialog box.

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.
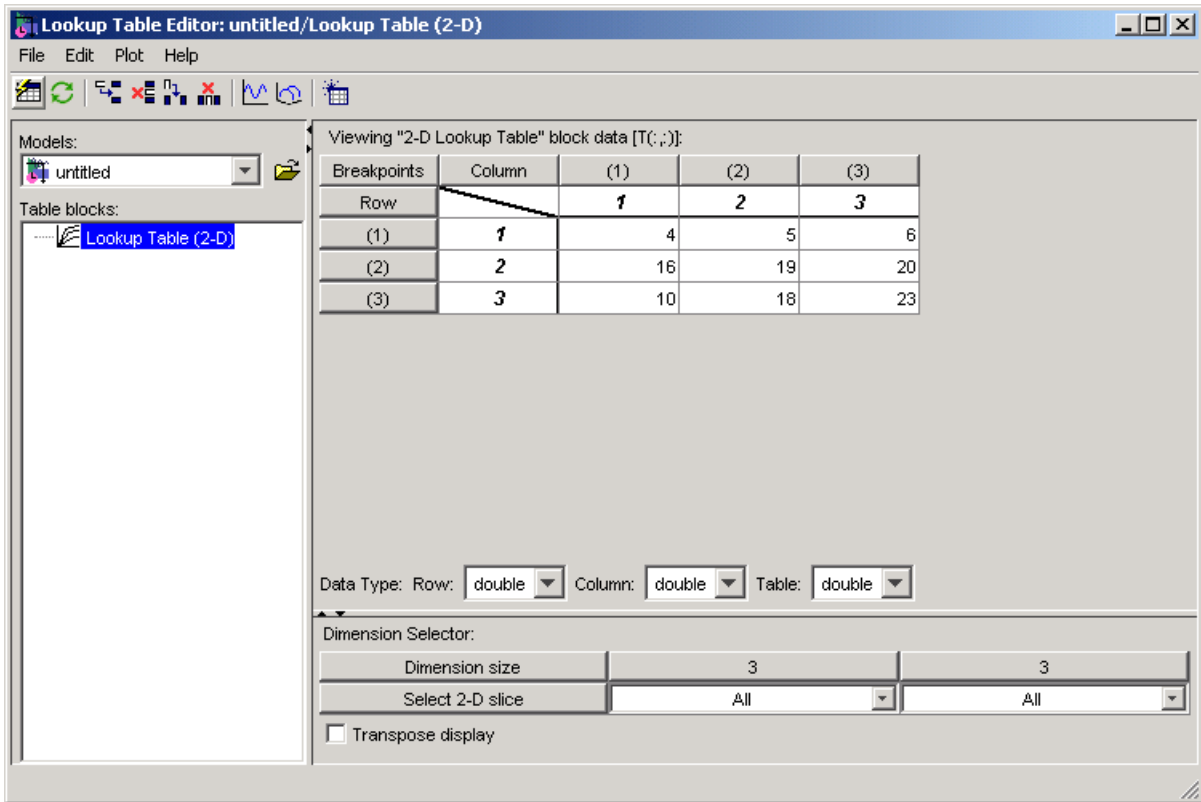
It displays the default data associated with the Lookup Table (2-D) block.

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:
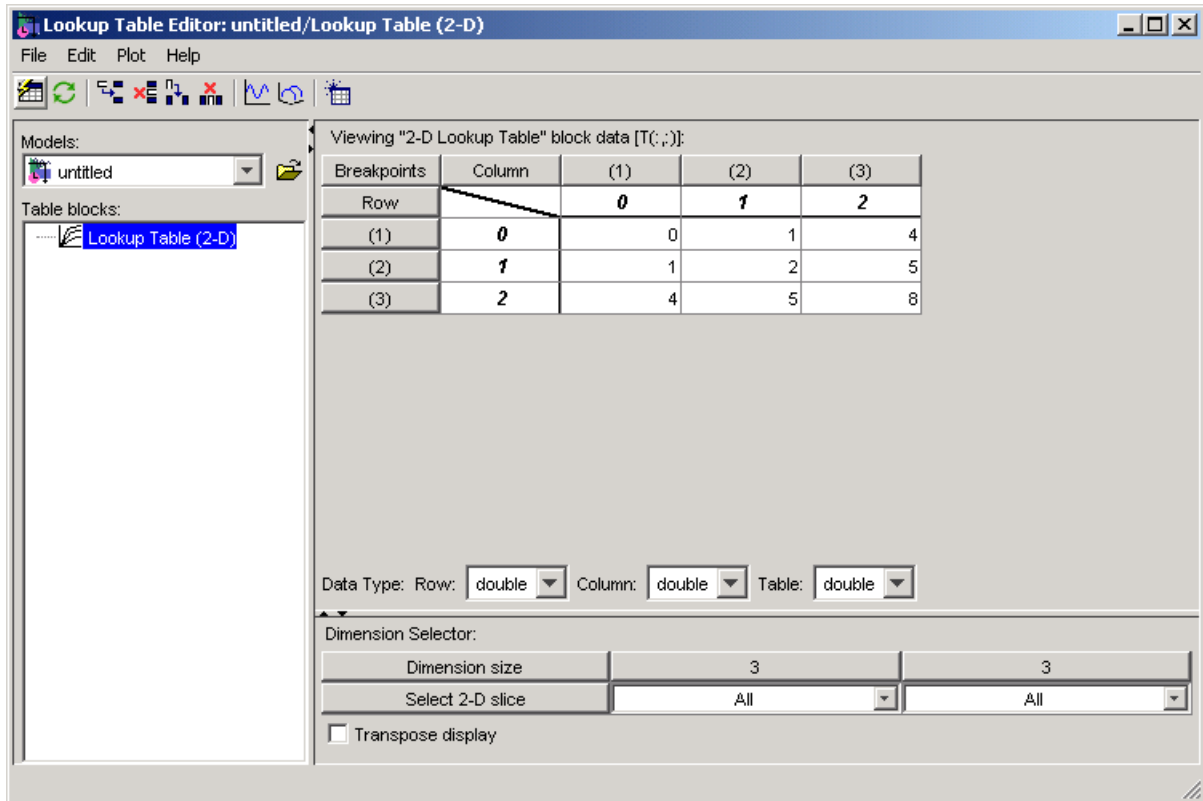
4 From the Lookup Table Editor's **File** menu, select **Update Block Data** to update the Lookup Table (2-D) block's data. Close the Lookup Table Editor.

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:
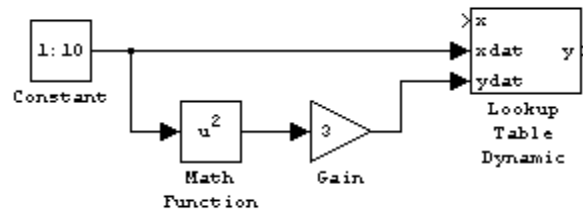
Characteristics of Lookup Table Data

In this section...

Sizes of Breakpoint Data Sets and Table Data

The following constraints are imposed 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]
```


● — Breakpoint Value
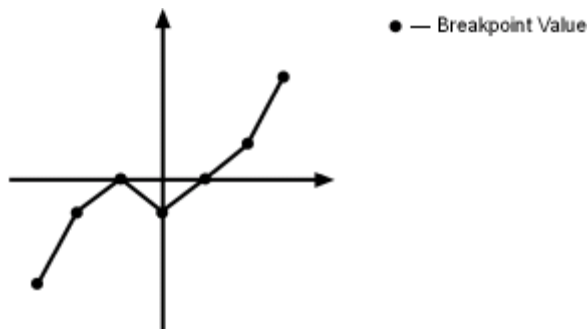
Here, the input and output data are the same size (1-by-7), making the data consistently dimensioned for a 1-D lookup table.

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]
```
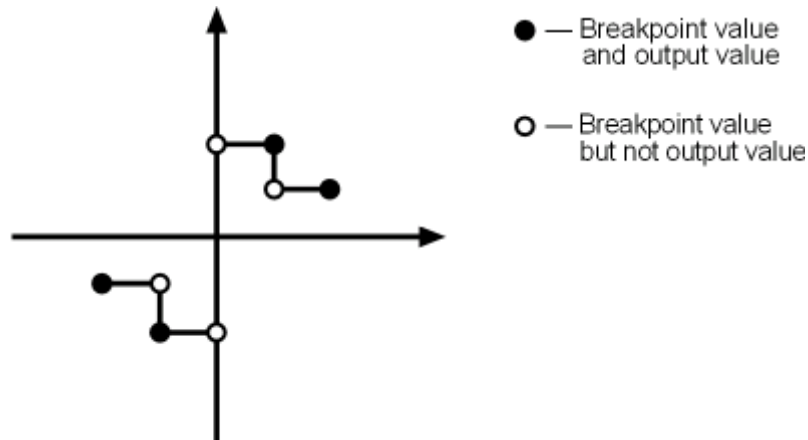

This example has discontinuities at *x* = -1, 0, and +1.

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]
```


● — Breakpoint value and output value

○ — Breakpoint value but not output value
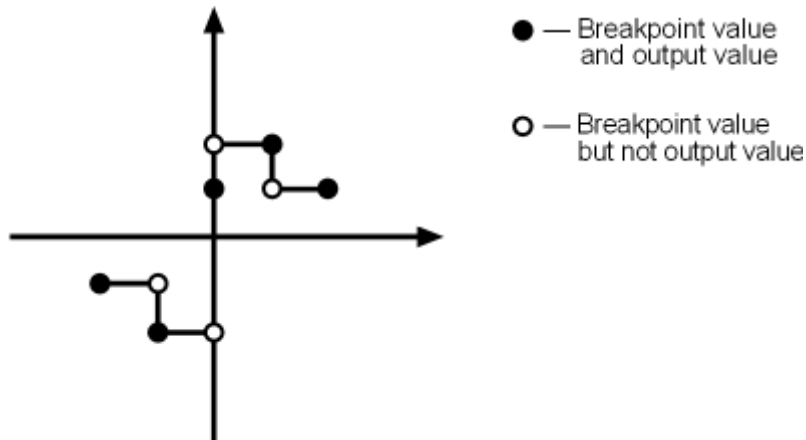
In this example, three points define the discontinuity at the origin. When the input is 0, y is 1, the value of the middle point.

You can apply this same method to create discontinuities in breakpoint data sets associated with multidimensional lookup tables.

Estimating Missing Points

In this section...
"About Estimating Missing Points" on page 11-21
"Interpolation Methods" on page 11-21
"Extrapolation Methods" on page 11-22
"Rounding Methods" on page 11-23
"Example Output" on page 11-24

About Estimating Missing Points

The second stage of a table lookup operation involves generating outputs that correspond to the supplied inputs. If the inputs match the values of indices specified in breakpoint data sets, the block outputs the corresponding values. However, if the inputs fail to match index values in the breakpoint data sets, Simulink® estimates the output. In the block'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

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 11-23.

- 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, the value is rounded 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]).
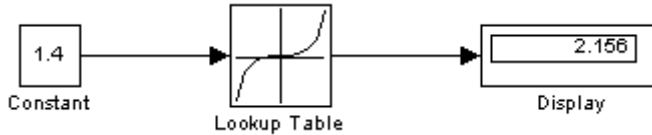
The following outputs are generated when using the specified lookup methods and inputs:

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

About the 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 11-32). 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.

Opening the LUT Editor

To open the editor, select **Lookup Table Editor** from the Simulink **Tools** menu. The editor appears.
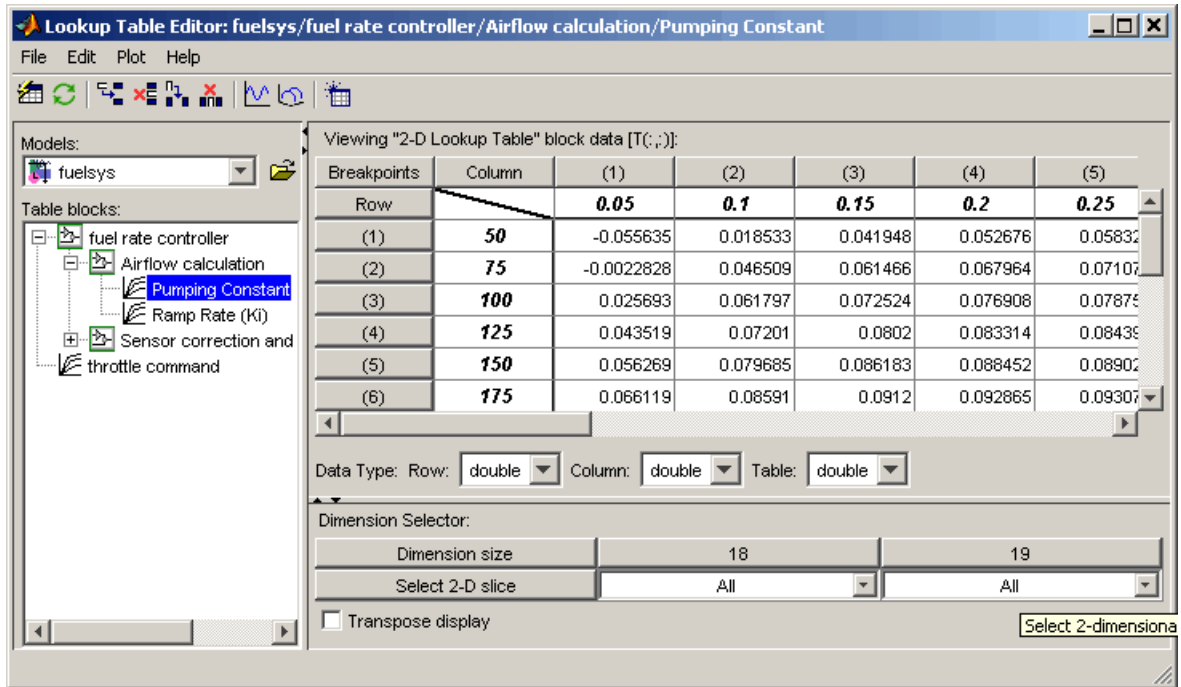
The editor contains two panes and a toolbar. The pane on the left is a LUT block browser. It allows you to browse and select LUT blocks in any open model (see "Browsing LUT Blocks" on page 11-26). The pane on the right allows you to edit the selected block's lookup table ("Editing Table Values" on page 11-28). 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® software 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.
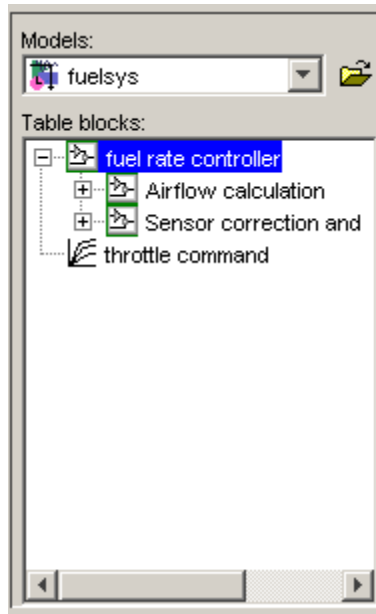
The tree view initially lists all the LUT blocks that reside at the model's root level. It also displays any subsystems that contain LUT blocks. Clicking the expand button (+) to the left of the subsystem's name expands the tree to show the LUT blocks in that subsystem. The expanded view also shows any 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 11-28).

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 11-29). 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.

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
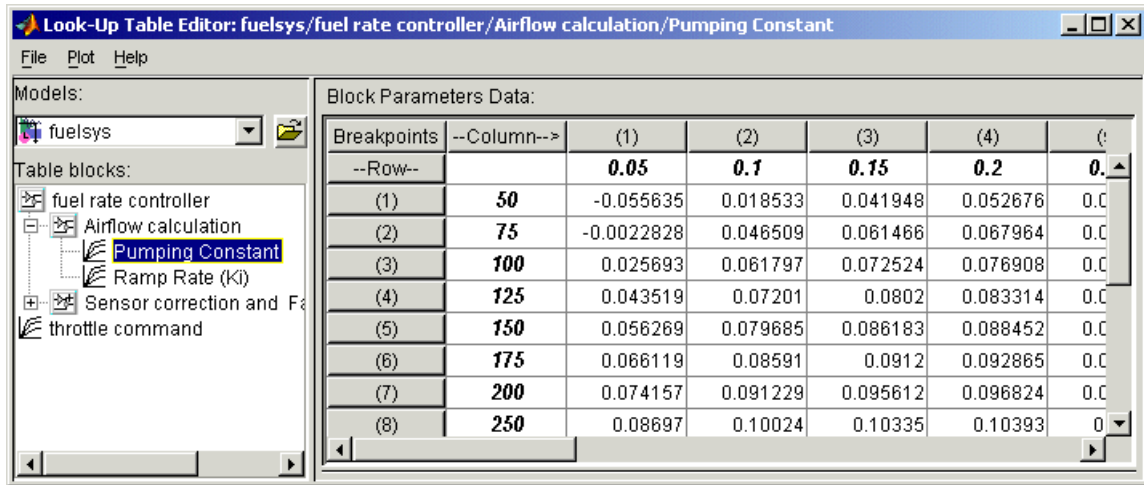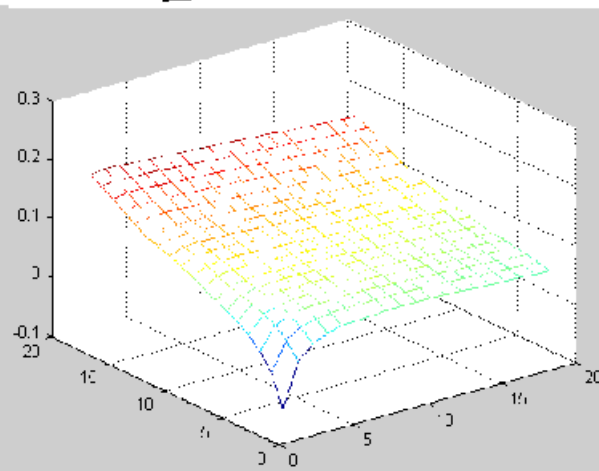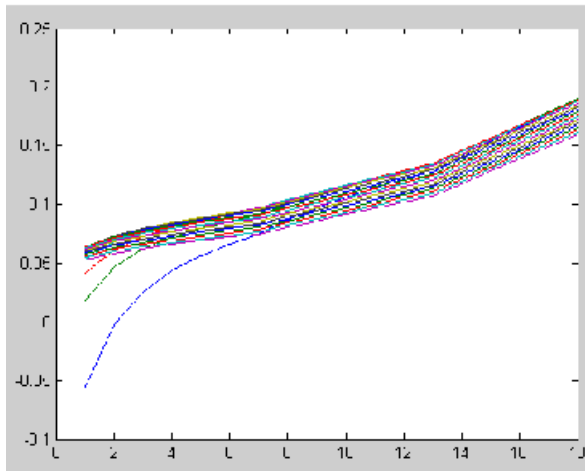
Editing Custom LUT Blocks

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		

Add   Remove   OK   Cancel

By default, the dialog box displays a table of the types of LUT blocks that the LUT Editor currently recognizes. These include the standard MATLAB 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 as described in the following procedure.

1 Copy the following blocks to the Simulink® model window:

- A Constant block to input the signal, from the Sources library

- A Lookup Table 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:
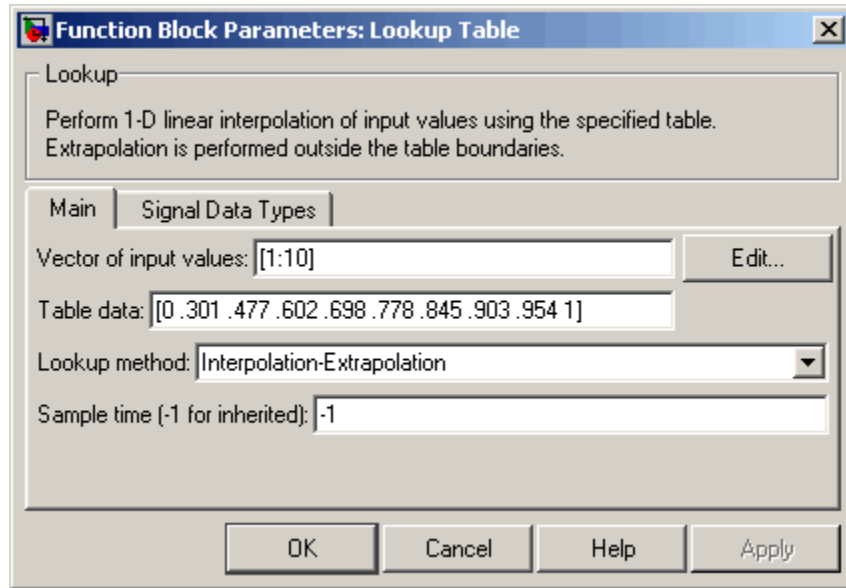
Click the **OK** button to apply the changes and close the dialog box.

3 Connect the blocks such that you input the Constant to the Lookup Table and display its output:
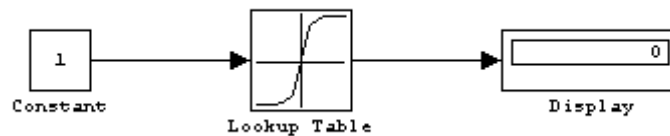
4 Double-click the Constant block to open its parameter dialog box, and change the **Constant value** parameter to, e.g., 5. Click the **OK** button to apply the changes and close the dialog box.

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.

12

Modeling with Simulink®

General Considerations when Building Simulink® Models

In this section...
"Avoiding Invalid Loops" on page 12-2
"Shadowed Files" on page 12-4
"Model Building Tips" on page 12-6

Avoiding Invalid Loops

You can connect the output of a block directly or indirectly (i.e., via other blocks) to its input, thereby, creating a loop. Loops can be very useful. For example, you can use loops to solve differential equations diagrammatically (see "Modeling a Continuous System" on page 12-8) or model feedback control systems. However, it is also possible to create loops that cannot be simulated. Common types of invalid loops include:

- Loops that create invalid function-call connections or an attempt to modify the input/output arguments of a function call (see "Function-Call Subsystems" on page 4-21 for a description of function-call subsystems)

- Self-triggering subsystems and loops containing non-latched triggered subsystems (see "Triggered Subsystems" on page 4-12 in the Using Simulink® documentation for a description of triggered subsystems and Inport in the Simulink reference documentation for a description of latched input)

- Loops containing action subsystems

The Subsystem Examples block library in the Ports & Subsystems library contains models that illustrates examples of valid and invalid loops involving triggered and function-call subsystems. Examples of invalid loops include the following models:

- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr1 (sl_subsys_trigerr1)`

- `simulink/Ports&Subsystems/sl_subsys_semantics/Triggered subsystem/sl_subsys_trigerr2 (sl_subsys_trigerr2)`

- `simulink/Ports&Subsystems/sl_subsys_semantics/Function-call
  systems/sl_subsys_fcncallerr3 (sl_subsys_fcncallerr3)`

You might find it useful to study these examples to avoid creating invalid loops in your own models.

Detecting Invalid Loops

To detect whether your model contains invalid loops, select **Update Diagram** from the model's **Edit** menu. If the model contains invalid loops, the invalid loops are highlighted. This is illustrated in the following model ,
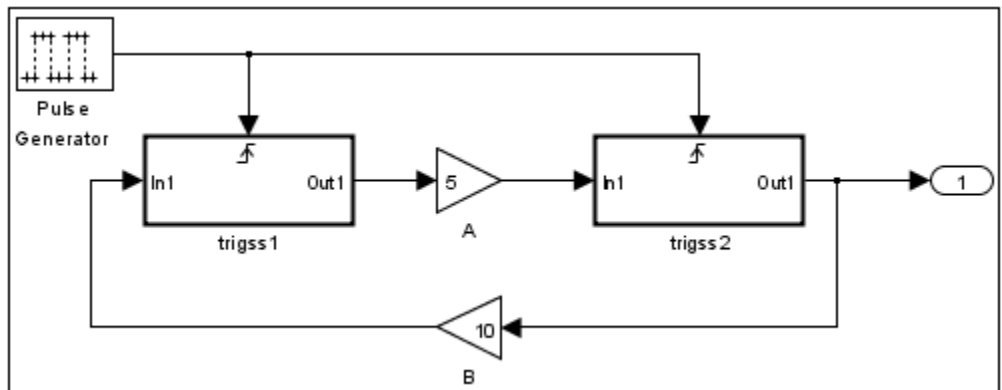
and displays an error message in the Simulation Diagnostics Viewer.
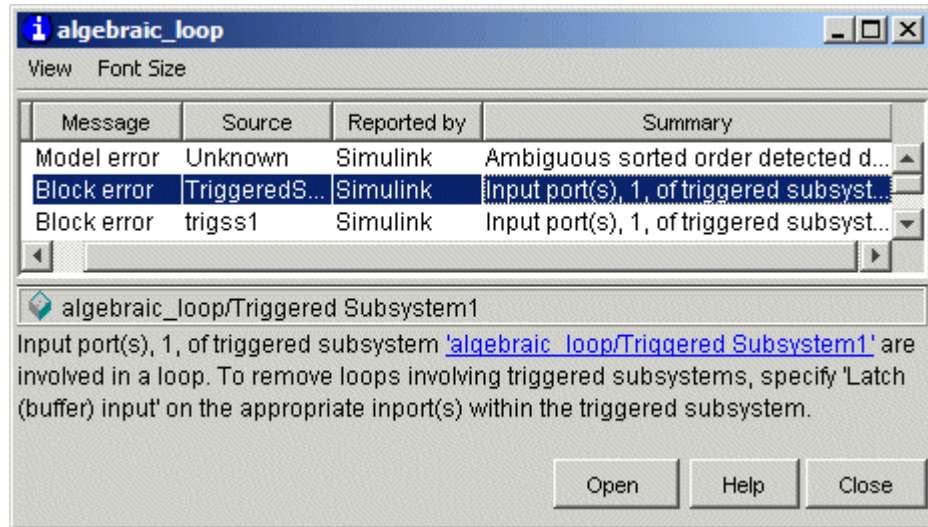
Shadowed Files

If there are two Model files with the same name (e.g. `mylibrary.mdl`) on the MATLAB® path, the one higher on the path is loaded, and the one lower on the path is said to be "shadowed".

The rules which the Simulink software uses to find Model files are similar to those used by the MATLAB software. See "How the Search Path Determines Which Function to Use" in the MATLAB documentation. However, there is an important difference between the way in which Simulink block diagram and MATLAB functions are handled: a loaded block diagram takes precedence over any unloaded ones, regardless of its position on the MATLAB path. This is done for performance reasons, as part of the Simulink software's incremental loading methodology.

The precedence of a loaded block diagram over any others can have important implications, particularly since a block diagram can be loaded without the corresponding Simulink window being visible.

Making Sure the Correct Block Diagram Is Loaded

When using libraries and referenced models, a block diagram may be loaded without showing its window. If the MATLAB path or the current MATLAB directory changes while block diagrams are in memory, these block diagrams may interfere with the use of other files of the same name. For example, after a change of directory, a loaded but invisible library may be used instead of the one the user expects.

To see an example:

1 Enter `sldemo_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 the Simulink software next needs to use a block in a library called hydlib it will use the file called hydlib.mdl which is highest on the MATLAB path at the time. Alternatively, to make the library visible, enter:

```
open_system('hydlib')
```

Detecting and Fixing Problems

When updating a block diagram, the Simulink software checks the position of its file on the MATLAB path and will issue a warning if it detects that another file of the same name exists and is higher on the MATLAB path. The warning reads:

```
The file containing block diagram 'mylibrary' is shadowed
by a file of the same name higher on the MATLAB path.
```

This may indicate that the wrong file called mylibrary.mdl is being used. To see which file called mylibrary.mdl is loaded into memory, enter:

```
 which mylibrary

C:\work\Model1\mylibrary.mdl
```

To see all the files called mylibrary which are on the MATLAB path (note that this can include M-files), enter:

```
which -all mylibrary

C:\work\Model1\mylibrary.mdl
C:\work\Model2\mylibrary.mdl  % Shadowed
```

To close the block diagram called mylibrary and let the Simulink software load the file which is highest on the MATLAB path, enter:

```
close_system('mylibrary')
```

Model Building Tips

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

- Memory issues

  In general, more memory will increase performance.

- Using hierarchy

  More complex models often benefit from adding the hierarchy of subsystems to the model. Grouping blocks simplifies the top level of the model and can make it easier to read and understand the model. For more information, see "Creating Subsystems" on page 3-35. The Model Browser provides useful information about complex models (see "The Model Browser" on page 13-28).

- 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 and "Annotating Diagrams" on page 3-24.

- Modeling strategies

  If several of your models tend to use the same blocks, you might find it easier to save these blocks in a model. Then, when you build new models, just open this model and copy the commonly used blocks from it. You can create a block library by placing a collection of blocks into a system and saving the system. You can then access the system by typing its name in the MATLAB Command Window.

  Generally, when building a model, design it first on paper, then build it using the computer. Then, when you start putting the blocks together into a model, add the blocks to the model window before adding the lines that connect them. This way, you can reduce how often you need to open block libraries.

Modeling a Continuous System

To model the differential equation

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

where *u(t)* is a square wave with an amplitude of 1 and a frequency of 1 rad/sec. The Integrator block integrates its input x' to produce x. Other blocks needed in this model include a Gain block and a Sum block. To generate a square wave, use a Signal Generator block and select the Square Wave form but change the default units to radians/sec. Again, view the output using a Scope block. Gather the blocks and define the gain.

In this model, to reverse the direction of the Gain block, select the block, then use the **Flip Block** command from the **Format** menu. To create the branch line from the output of the Integrator block to the Gain block, hold down the **Ctrl** key while drawing the line. For more information, see "Drawing a Branch Line" on page 3-17.

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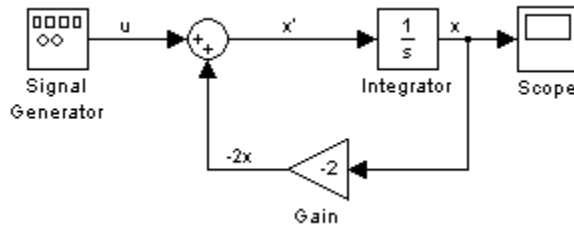
An important concept in this model is the loop that includes the Sum block, the Integrator block, and the Gain block. In this equation, x is the output of the Integrator block. It is also the input to the blocks that compute x', on which it is based. This relationship is implemented using a loop.

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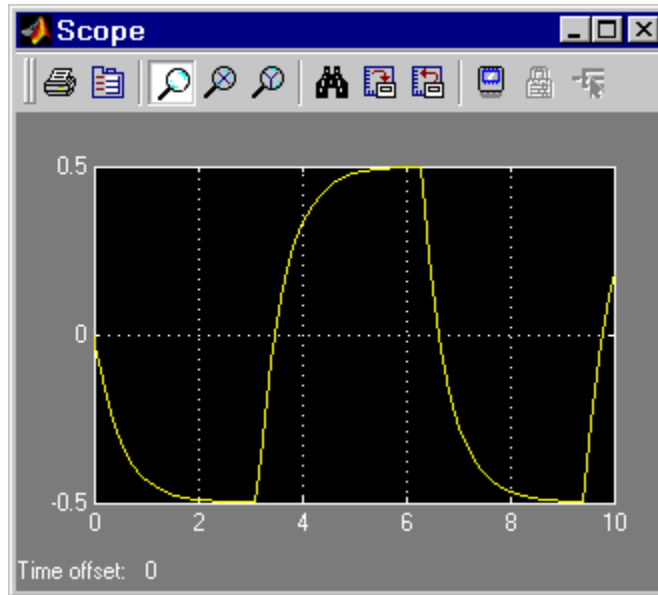
The equation you modeled in this example can also be expressed as a transfer function. The model uses the Transfer Fcn block, which accepts *u* as input and outputs *x*. So, the block implements x/u. If you substitute *sx* for *x´* in the above equation, you get

$$sx = -2x + u$$

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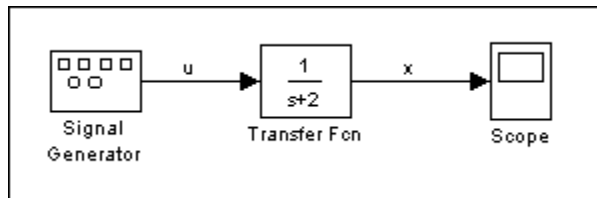
The results of this simulation are identical to those of the previous model.

Best-Form Mathematical Models

Series RLC Example

You can often formulate the mathematical system you are modeling in several ways. Choosing the best-form mathematical model allows the simulation to execute faster and more accurately. For example, consider a simple series RLC circuit.
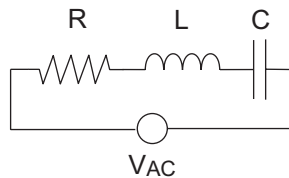
According to Kirchoff's voltage law, the voltage drop across this circuit is equal to the sum of the voltage drop across each element of the circuit.

$$V_{AC} = V_R + V_L + V_C$$

Using Ohm's law to solve for the voltage across each element of the circuit, the equation for this circuit can be written as

$$V_{AC} = Ri + L\frac{di}{dt} + \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

You can model this system in Simulink® by solving for either the resistor voltage or inductor voltage. Which you choose to solve for affects the structure of the model and its performance.

Solving Series RLC Using Resistor Voltage

Solving the RLC circuit for the resistor voltage yields

$$V_R = Ri$$

$$Ri = V_{AC} - L\frac{di}{dt} - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

Circuit Model

The following diagram shows this equation modeled in Simulink where R is 70, C is 0.00003, and L is 0.04. The resistor voltage is the sum of the voltage source, the capacitor voltage, and the inductor voltage. You need the current in the circuit to calculate the capacitor and inductor voltages. To calculate the current, multiply the resistor voltage by a gain of $1/R$. Calculate the capacitor voltage by integrating the current and multiplying by a gain of $1/C$. Calculate the inductor voltage by taking the derivative of the current and multiplying by a gain of L.

Series RLC Circuit: Formulated to solve for resistor current
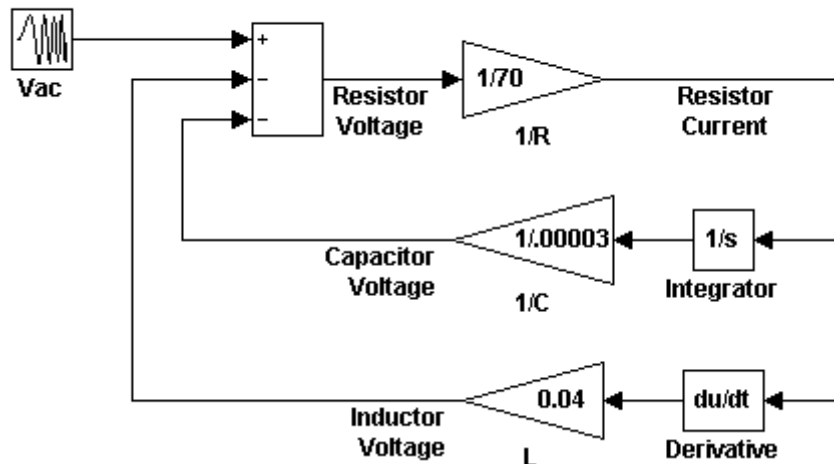
This formulation contains a Derivative block associated with the inductor. Whenever possible, you should avoid mathematical formulations that require Derivative blocks as they introduce discontinuities into your system.

Numerical integration is used to solve the model dynamics 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 2-31 for more information.

Solving Series RLC Using Inductor Voltage

To avoid using a Derivative block, formulate the equation to solve for the inductor voltage.

$$V_L = L\frac{di}{dt}$$

$$L\frac{di}{dt} = V_{AC} - Ri - \frac{1}{C}\int_{-\infty}^{t} i(t)dt$$

Circuit Model

The following diagram shows this equation modeled in Simulink. The inductor voltage is the sum of the voltage source, the resistor voltage, and the capacitor voltage. You need the current in the circuit to calculate the resistor and capacitor voltages. To calculate the current, integrate the inductor voltage and divide by L. Calculate the capacitor voltage by integrating the current and dividing by C. Calculate the resistor voltage by multiplying the current by a gain of R.

Series RLC Circuit: Formulated to solve for inductor voltage
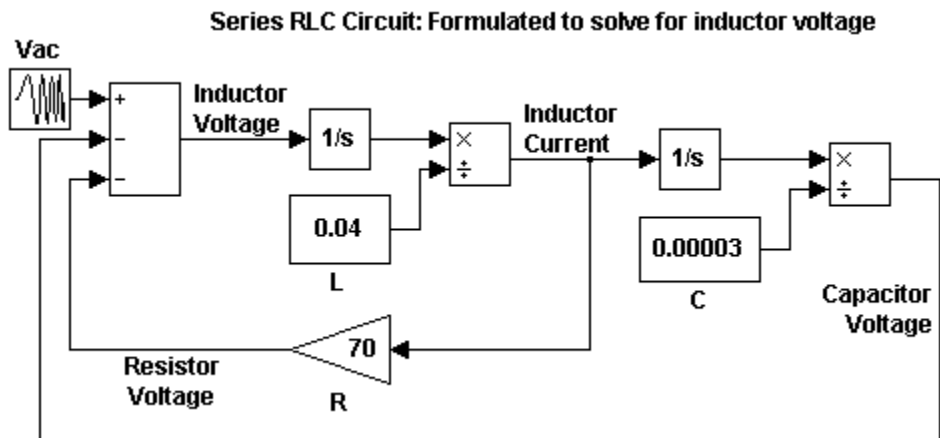
This model contains only integrator blocks and no algebraic loops. As a result, the model simulates faster and more accurately.

Example: Converting Celsius to Fahrenheit

To model the equation that converts Celsius temperature to Fahrenheit

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

First, consider the blocks needed to build the model:

- A Ramp block to input the temperature signal, from the Sources library

- A Constant block to define a constant of 32, also from the Sources library

- A Gain block to multiply the input signal by 9/5, from the Math Operations library

- A Sum block to add the two quantities, also from the Math Operations library

- A Scope block to display the output, from the Sinks library

Next, gather the blocks into your model window.
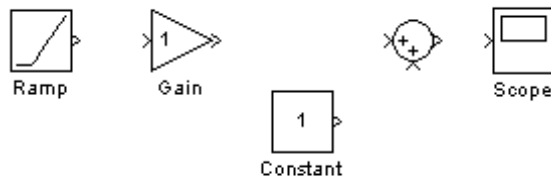
Assign parameter values to the Gain and Constant blocks by opening (double-clicking) each block and entering the appropriate value. Then, click the **OK** button to apply the value and close the dialog box.

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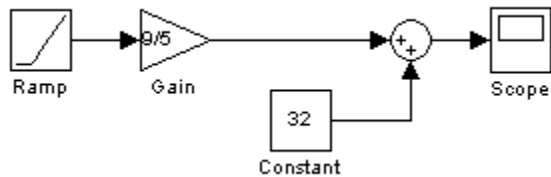
The Ramp block inputs Celsius temperature. Open that block and change the **Initial output** parameter to 0. The Gain block multiplies that temperature by the constant 9/5. The Sum block adds the value 32 to the result and outputs the Fahrenheit temperature.

Open the Scope block to view the output. Now, choose **Start** from the **Simulation** menu to run the simulation. The simulation runs for 10 seconds.

13

Exploring, Searching, and Browsing Models

The *Model* Explorer

Introduction to the Model Explorer

The Model Explorer allows you to quickly locate, view, and change elements of a Simulink® model or Stateflow® chart. To display the Model Explorer, select **Model Explorer** from the Simulink **View** menu or select an object in the block diagram and select **Explore** from its context menu. The Model Explorer appears.
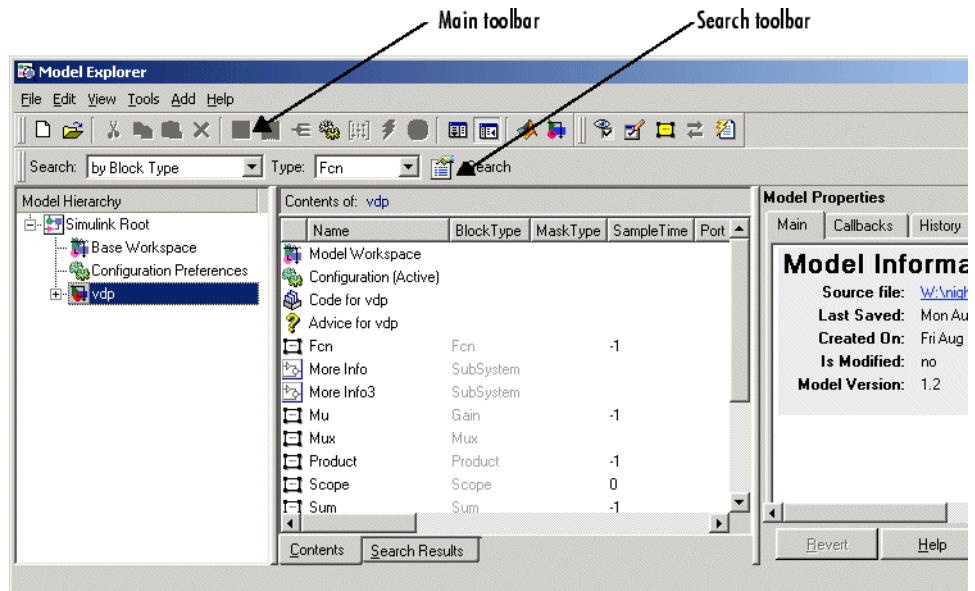
Model Hierarchy pane   Contents pane   Dialog pane

The Model Explorer includes the following components:

- **Model Hierarchy** pane (see "Model Hierarchy Pane" on page 13-4)

- **Contents** pane (see "Contents Pane" on page 13-6)

- **Dialog** pane (see "Dialog Pane" on page 13-12)

- **Main** toolbar (see "Main Toolbar" on page 13-14)

- **Search** bar (see "Search Bar" on page 13-17)

You can use the Model Explorer's **View** menu to hide the **Dialog** pane and the toolbars, thereby making more room for the other panes.

Model Hierarchy Pane

The **Model Hierarchy** pane displays a tree-structured view of the Simulink model hierarchy.
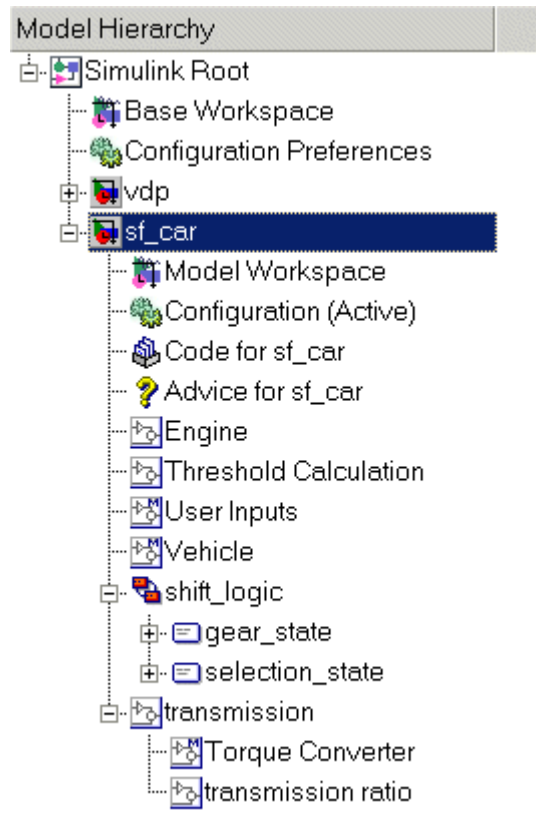
Simulink® Root

The first node in the view represents the Simulink root. Expanding the root node displays nodes representing the MATLAB® workspace (the Simulink base workspace) and each model and library loaded in the current session.

Base Workspace

This node represents the MATLAB workspace. The MATLAB workspace is the base workspace for Simulink models. Variables defined in this workspace are visible to all open Simulink models, i.e., to all models whose nodes appear beneath the **Base Workspace** node in the **Model Hierarchy** pane.

Configuration Preferences

If you check the **Show Configuration Preferences** option on the Model Explorer's **View** menu, the expanded Simulink Root node also displays a Configuration Preferences node. Selecting this node displays the preferred model configuration (see "Configuration Sets" on page 14-37) 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 14-45).

Model Nodes

Expanding a model node displays nodes representing the model's configuration sets (see "Configuration Sets" on page 14-37), 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 14-37 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 Chapter 5, "Referencing a Model"), you must first open the referenced model. To do this, right-click on the node to display its context menu, then select **Open Model** from the menu. The model to which the reference refers to is then opened, a node for it is displayed in the **Model Hierarchy** pane, and all references to the model are made expandable. You cannot edit the contents of a reference node, however. To edit the referenced model, you must expand its node.

Contents Pane

The **Contents** pane displays either of two tabular views selectable by tabs. The **Contents** tab displays the contents of the object selected in the **Model Hierarchy** pane. The **Search Results** tab displays the results of a search operation (see "Search Bar" on page 13-17) .
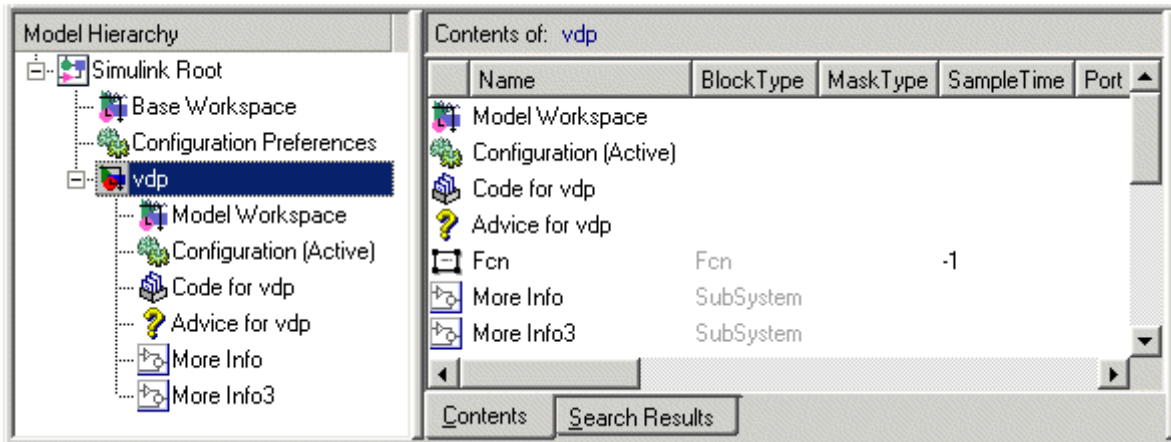
In both views, the table rows correspond to objects (e.g., blocks or states); the table columns, to object properties (e.g., name and type). The table cells display the values of the properties of the objects contained by the object selected in the **Model Hierarchy** pane or found by a search operation.

The objects and properties displayed in the **Contents** pane depend on the type of object (e.g., subsystem, chart, or configuration set) selected in the **Model Hierarchy** pane. For example, if the object selected in the **Model Hierarchy** pane is a model or subsystem, the **Contents** pane by default displays the name and type of the top-level blocks contained by that model or subsystem. If the selected object is a Stateflow chart or state, the **Contents** pane by default shows the name, scope, and other properties of the events and data that make up the chart or state.

Customize Contents Pane

The **Customize Contents** pane allows you to select the properties that the **Contents** pane displays for the object selected in the **Model Hierarchy** pane. When visible, the pane appears in the lower-left corner of the Model Explorer window.
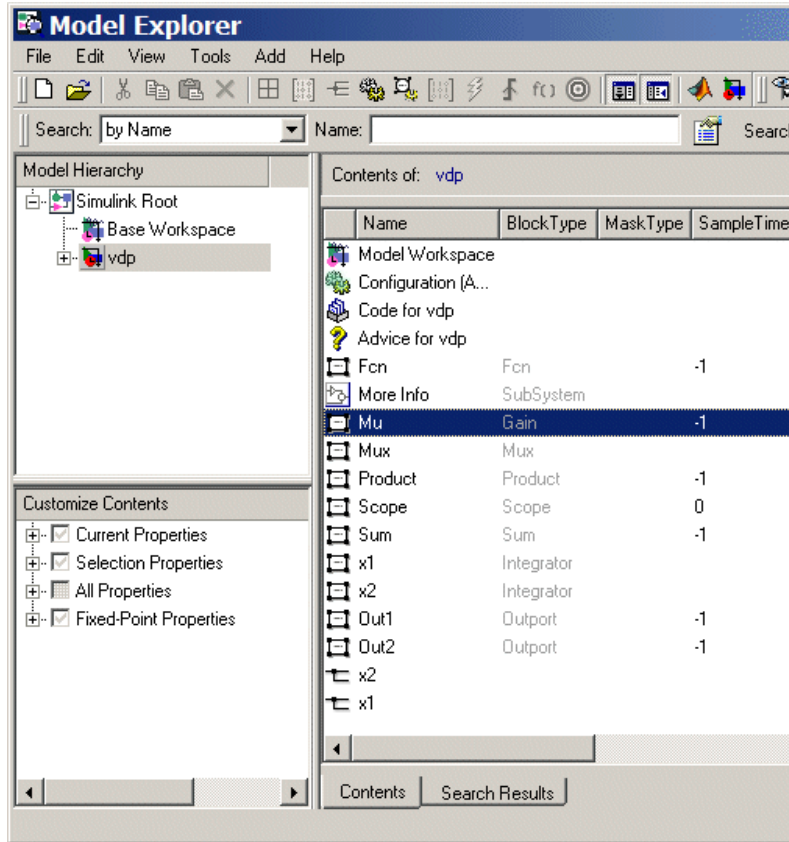
A splitter divides the **Customize Contents** pane from the **Model Hierarchy** pane above it. Drag the splitter up or down to adjust the relative size of the two panes.

The **Customize Contents** pane contains a tree-structured property list. The list's top-level nodes group object properties into the following categories:

- `Current Properties`

  Properties that the **Contents** pane currently displays.

- `Selection Properties`

  Properties of the object currently selected in the **Contents** pane.

- `All Properties`

  Properties of the contents of all models displayed in the Model Explorer thus far in this session.

- `Fixed Point Properties`

  Fixed-point properties of blocks.

By default, the **Contents** pane displays a standard subset of properties for the currently selected model. The **Customize Contents** pane allows you to perform the following customizations:

- To display additional properties of the selected model, expand the `All Properties` node, if necessary, and check the desired properties.

- To delete some but not all properties from the **Contents** pane, expand the `Current Properties` node, if necessary, and uncheck the properties that you do not want to appear in the **Contents** pane.

- To delete all properties from the **Contents** pane (except the selected object's name), uncheck `Current Properties`.

- To display only the properties of the currently selected object, uncheck `Current Properties` to clear the properties display, then check `Selection Properties`.

- To add or remove fixed-point block properties from the **Contents** pane, check or uncheck `Fixed Point Properties`.

Customizing the Contents Pane

The Model Explorer's **View** menu allows you to control the type of objects and properties displayed in the **Contents** pane.

- To display only object names in the **Contents** pane, uncheck the **Show Properties** item on the **View** menu.

- To customize the set of properties displayed in the **Contents** pane, select **Customize Contents** from the **View** menu or click the **Customize Contents** button on the Model Explorer's main toolbar (see "Main Toolbar" on page 13-14). 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.
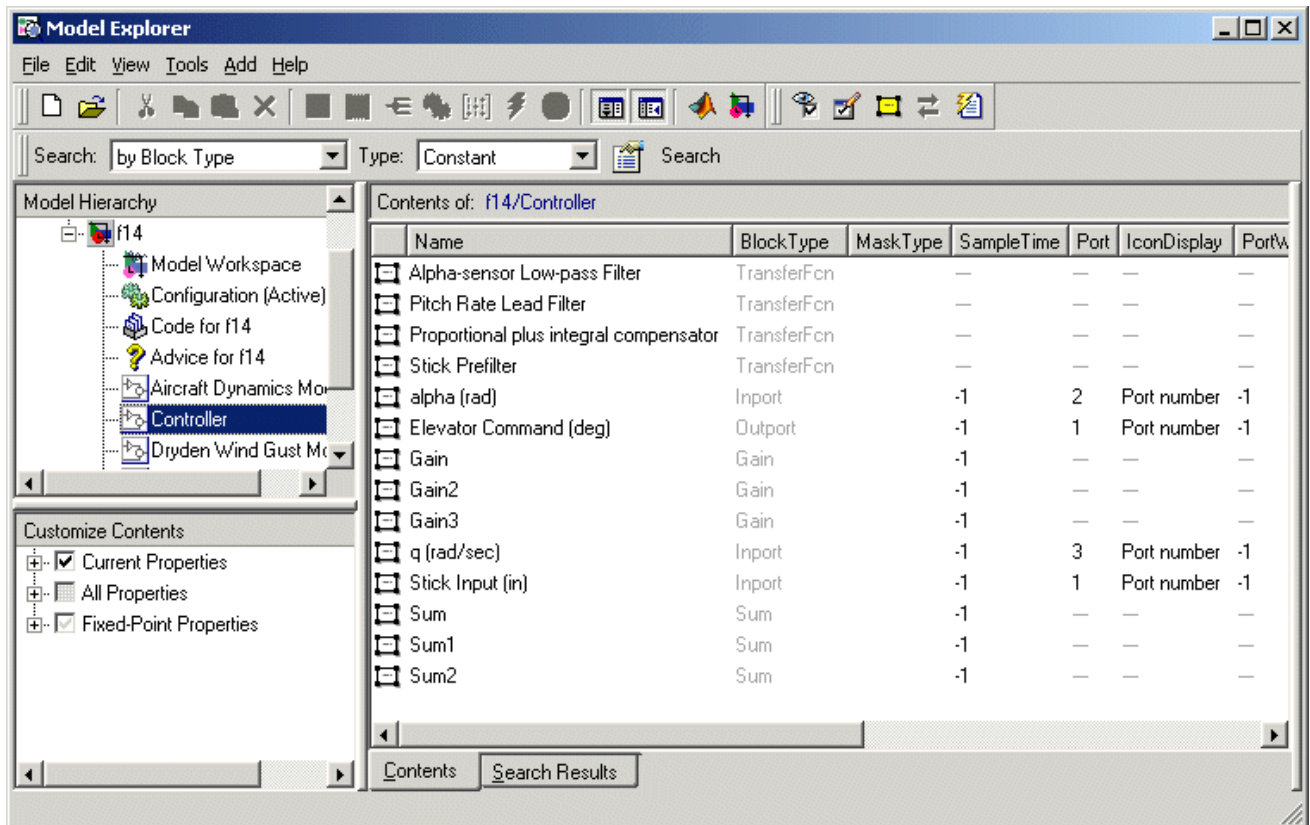
Changing Property Values

You can change modifiable properties displayed in the **Contents** pane (e.g., a block's name) by editing the displayed value. To edit a displayed value, first select the row that contains it. Then click the value. An edit control replaces the displayed value (e.g., an edit field for text values or a pull-down list for a range of values). Use the edit control to change the value of the selected property.

To assign the same property value to multiple objects displayed in the **Contents** pane, select the objects and then change one of the selected objects to have the new property value. The Model Explorer assigns the new property value to the other selected objects as well.

Dialog Pane

Use the **Dialog** pane to view and change properties of the blocks or signals in your model.

1 To show the **Dialog** pane, select **Dialog View** from the **View** menu, located on the Model Explorer's main toolbar.

2 In the **Contents** pane, select an object (such as a block or signal). The properties are displayed in the **Dialog** pane.
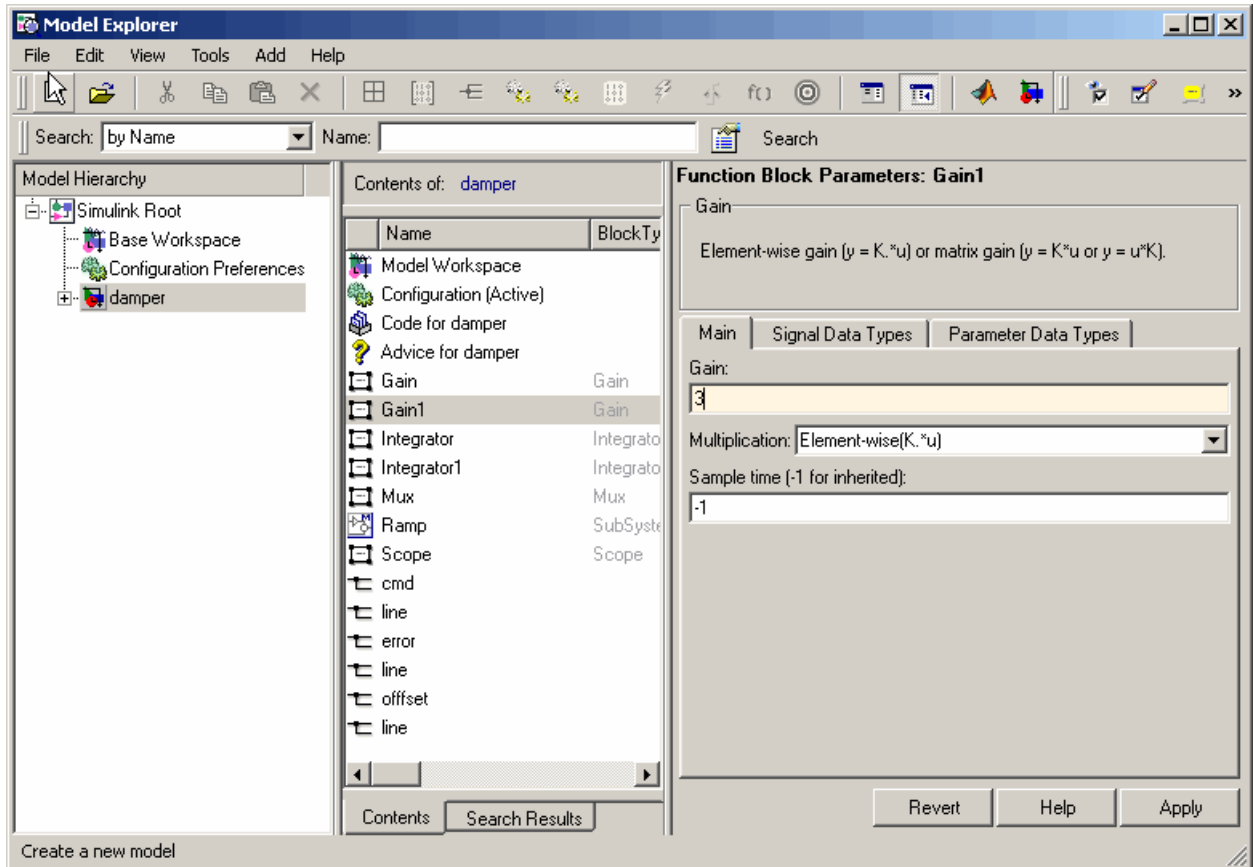
3 Change a property (for example, the gain of a gain block) in the **Dialog** pane.

4 Click **Apply** to accept the change, or click **Revert** to return to the original value.

By default, clicking outside a dialog with unapplied changes causes the **Model Explorer – Apply Changes** dialog box to appear:
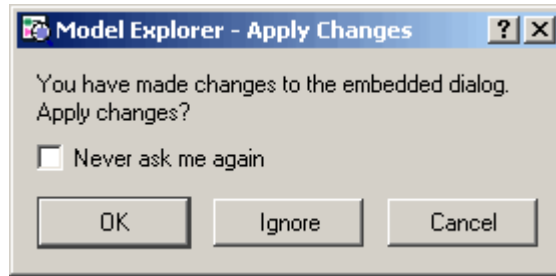
Click **OK** to accept the changes, **Ignore** to revert to the original settings, or **Cancel** to dismiss the dialog box without making any changes. To suppress this dialog box in the future, check **Never ask me again** on the dialog box, or check **Auto Apply/Ignore Dialog Changes** on the Model Explorer **Tools** menu.

If you suppress the dialog box, and thereafter click outside a dialog that has unapplied changes, the Model Explorer automatically applies or discards changes, depending on which action you most recently chose in the dialog box. You can re-enable the dialog box for future operations by unchecking **Tools > Auto Apply/Ignore Dialog Changes**.

Main Toolbar

The Model Explorer's main toolbar appears near the top of the Model Explorer window under the Model Explorer's menu.
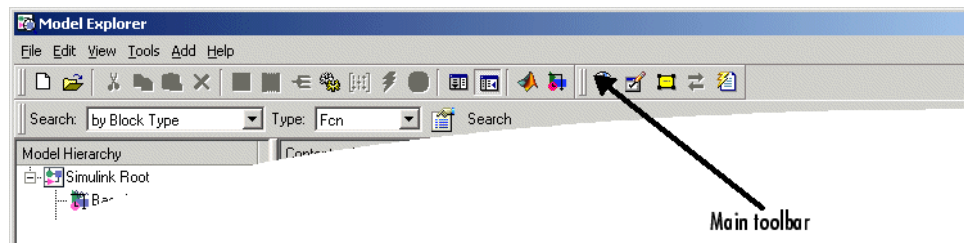
The toolbar contains buttons that select commonly used Model Explorer commands:

Button	Usage
▯	Create a new model.
🗁	Open an existing model.
✂	Cut the objects (e.g., variables) selected in the **Contents** pane from the object (e.g., a workspace) selected in the **Model Hierarchy** pane. Save a copy of the object on the system clipboard.
🗐	Copy the objects selected in the **Contents** pane to the system clipboard.
📋	Paste objects from the clipboard into the object selected in the Model Explorer's **Model Hierarchy** pane.
✕	Delete the objects selected in the **Contents** pane from the object selected in the **Model Hierarchy** pane.
⊞	Add a MATLAB variable to the workspace selected in the **Model Hierarchy** pane.
▦	Add a Simulink.Parameter object to the workspace selected in the **Model Hierarchy** pane.
⊟	Add a Simulink.Signal object to the workspace selected in the **Model Hierarchy** pane.
⚙	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.

Button	Usage
	Add a code generation target to the model selected in the **Model Hierarchy** pane.
	Turn the Model Explorer's **Dialog** pane on or off.
	Customize the Model Explorer's **Contents** pane.
	Bring the MATLAB desktop to the front.
	Display the Simulink Library Browser.

To show or hide the main toolbar, select **Main Toolbar** from the Model Explorer's **View** menu.

Search Bar

The Model Explorer's search bar allows you to select, configure, and initiate searches of the object selected in the **Model Hierarchy** pane. It appears at the top of the Model Explorer window.

Search bar
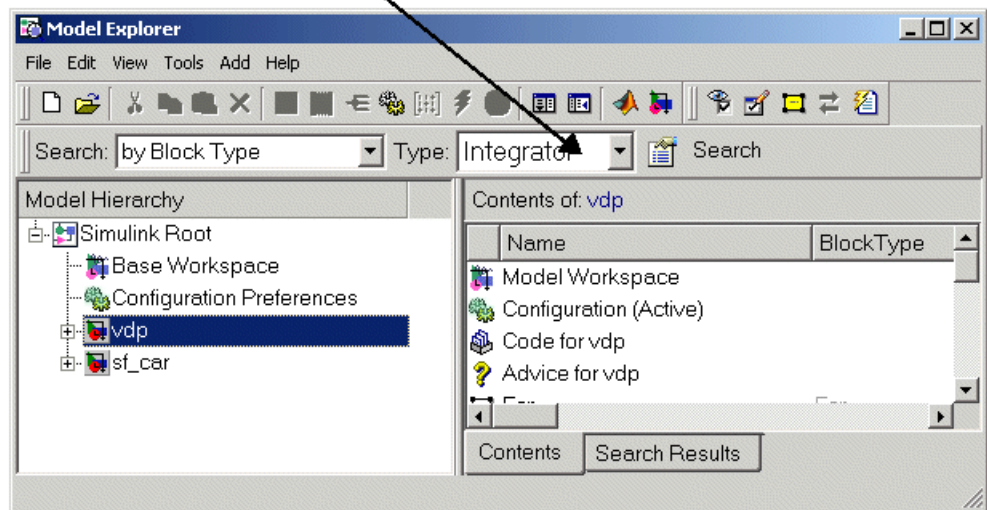
To show or hide the search bar, check or uncheck **Search Bar** in the Model Explorer's **View > Toolbars** menu.

The search bar includes the following controls:

Select search type.
Specify search criteria.
Start search.
Select search options.
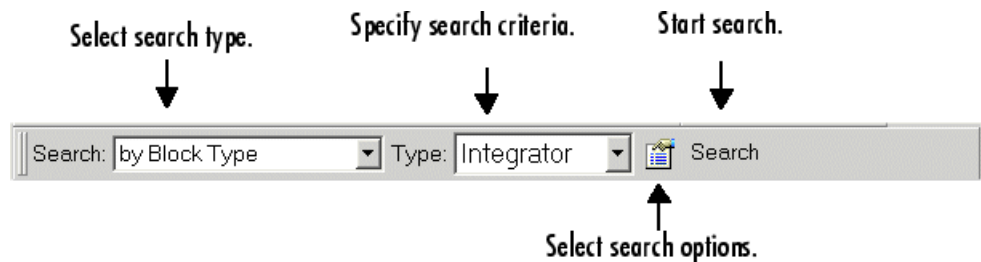
Search Type

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 13-21).

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
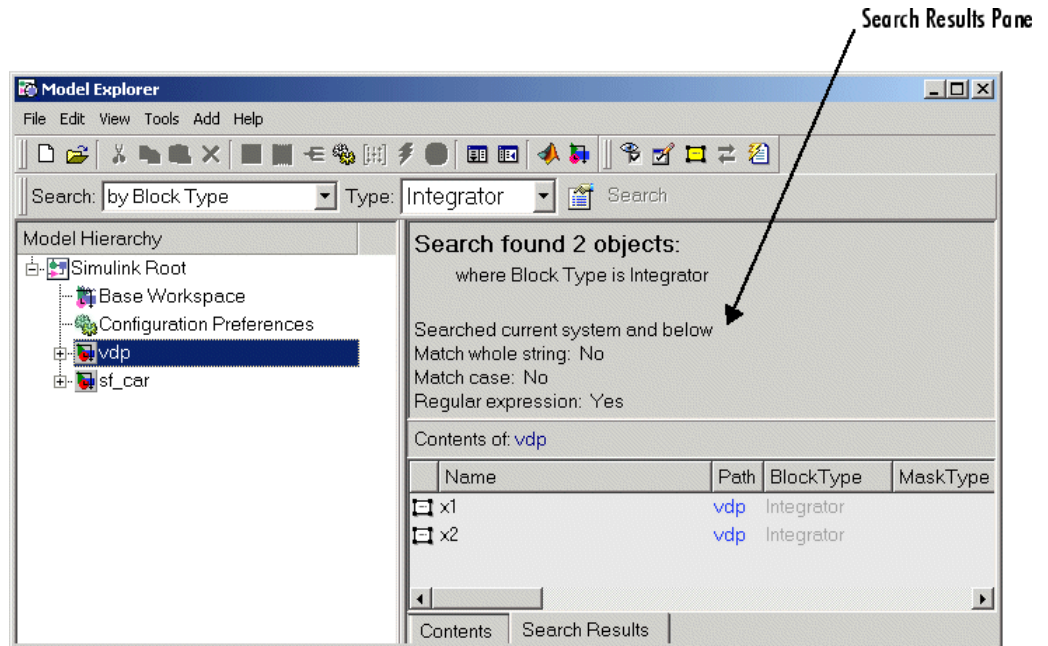
You can edit the results displayed in the **Search Results** pane. For example, to change all objects found by a search to have the same property value, select the objects in the **Search Results** pane and change one of them to have the new property value.

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 13-2).

- Press the **Ctrl+** keys to increase the font size

  Alternatively, from the Model Explorer's **View** menu, select **Increase Font Size**

- Press the **Ctrl-** keys to decrease the font size

  Alternatively, from the Model Explorer's View menu, select **Decrease Font Size**

Note These changes simultaneously alter the font size used by Model Explorer and in the Simulink dialog boxes. The changes remain in effect across Simulink sessions.

The Finder

In this section...
"About the Finder" on page 13-22
"Filter Options" on page 13-24
"Search Criteria" on page 13-25

About the Finder

The Finder locates blocks, signals, states, or other objects in a model. To display the Finder, select **Find** from the Simulink® Model Editor's **Edit** menu. The **Find** dialog box appears.
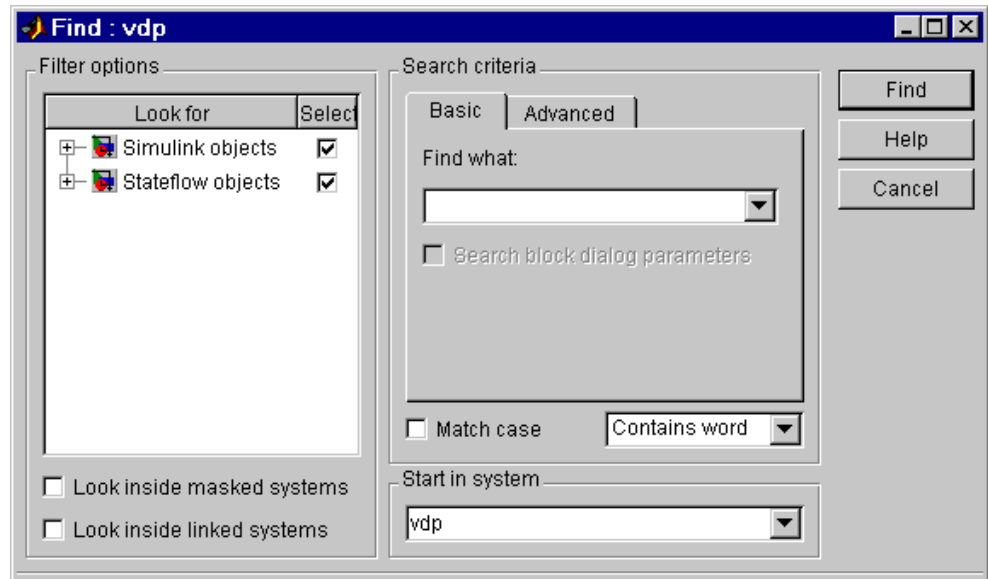
Use the **Filter options** (see "Filter Options" on page 13-24) and **Search criteria** (see "Search Criteria" on page 13-25) panels to specify the characteristics of the object you want to find. Next, if you have more than one system or subsystem open, select the system or subsystem where you want the 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.
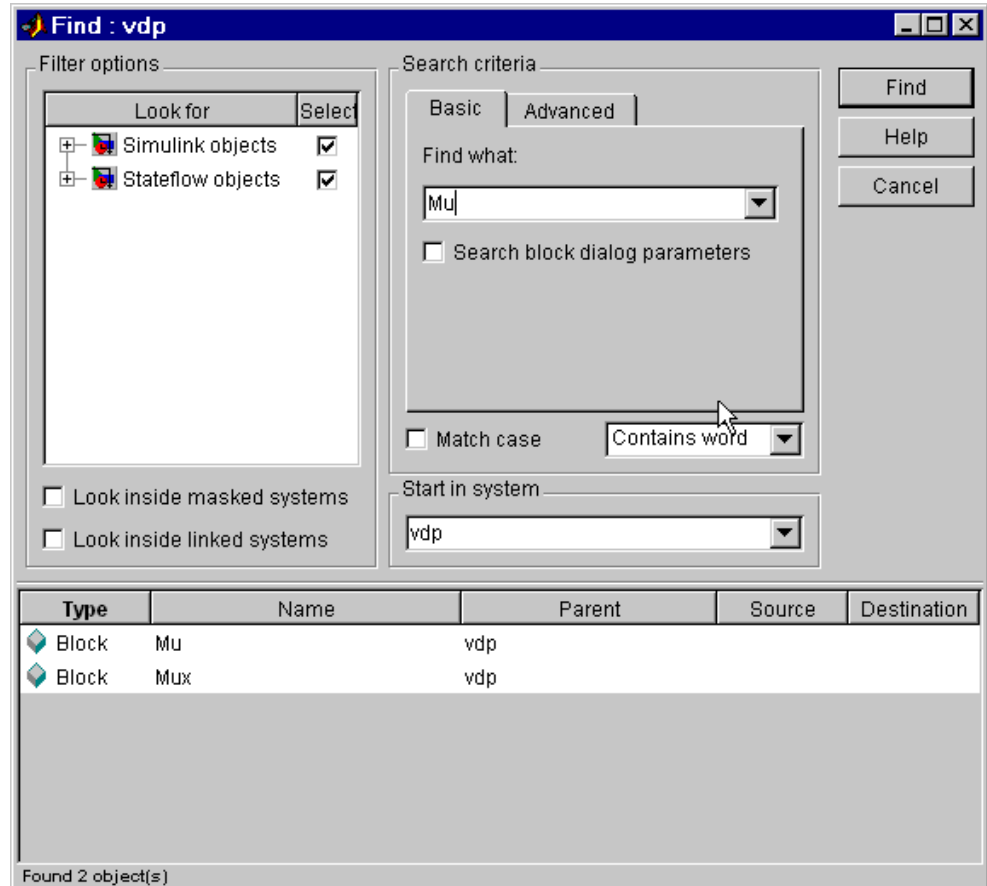
You can display an object by double-clicking its entry in the search results list. Simulink opens the system or subsystem that contains the object (if necessary) and highlights and selects the object. To sort the results list, click any of the buttons at the top of each column. For example, to sort the results by object type, click the **Type** button. Clicking a button once sorts the list in

ascending order, clicking it twice sorts it in descending order. To display an object's parameters or properties, select the object in the list. Then press the right mouse button and select **Parameter** or **Properties** from the resulting context menu.

Filter Options

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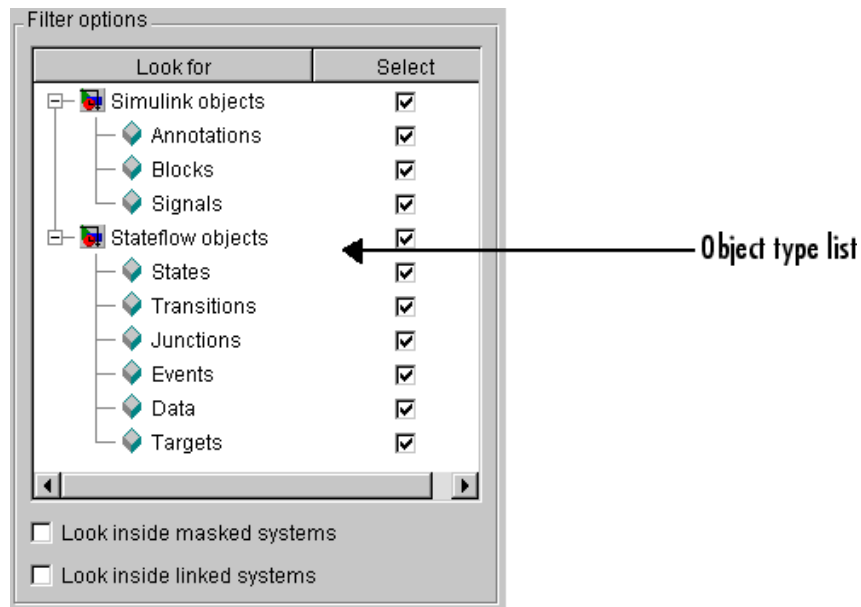
Object type list

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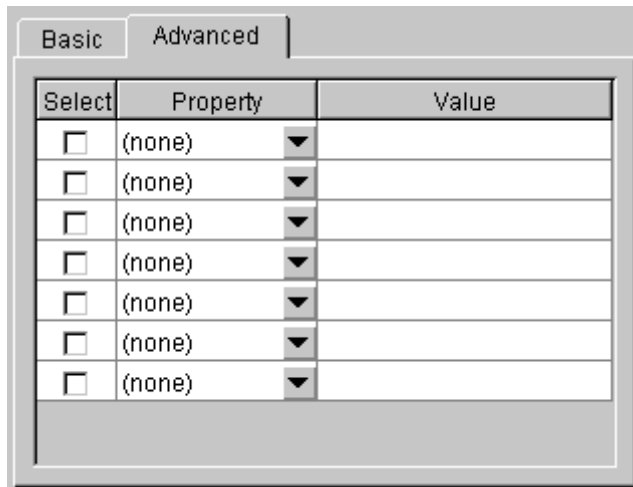
The **Search criteria** panel allows you to specify the criteria that objects must meet to satisfy your search request.

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.

Basic	Advanced	

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.

Character	Meaning
\	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]).
\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

About the Model Browser

The Model Browser enables you to

- Navigate a model hierarchically

- Open systems in a model

- Determine the blocks contained in a model

Note The browser is available only on Microsoft® Windows® platforms.

To display the Model Browser, select **Model Browser Options > Model Browser** from the Simulink® **View** menu.
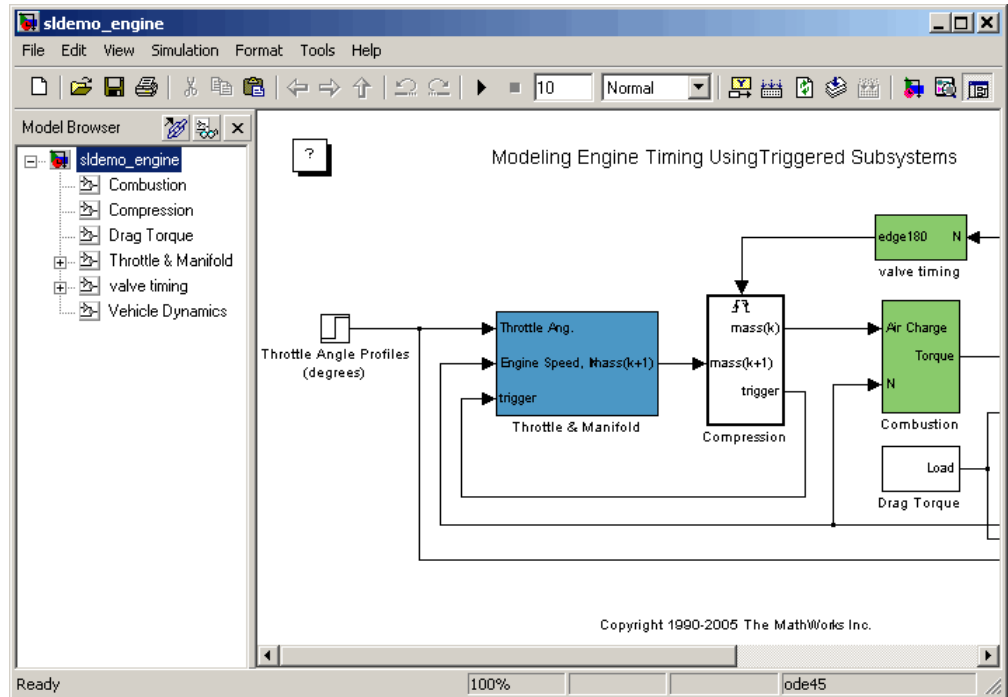
The model window splits into two panes. The left pane displays the browser, a tree-structured view of the block diagram displayed in the right pane.

Note The **Browser initially visible** preference causes Simulink to open models by default in the Model Browser. To set this preference, select **Preferences** from the Simulink **File** menu.

The top entry in the tree view corresponds to your model. A button next to the model name allows you to expand or contract the tree view. The expanded view shows the model's subsystems. A button next to a subsystem indicates that the subsystem itself contains subsystems. You can use the button to list the subsystem's children. To view the block diagram of the model or any subsystem displayed in the tree view, select the subsystem. You can use either the mouse or the keyboard to navigate quickly to any subsystem in the tree view.

Navigating with the Mouse

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

Navigating with the Keyboard

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

Showing Library Links

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

Showing Masked Subsystems

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

Model Dependencies

What Are Model Dependencies?

Each Simulink® model requires a set of files to run successfully. These files can include referenced models, data files, S-functions, and other files without which the model cannot run. These required files are called *model dependencies*.

The Simulink Manifest Tools allow you to analyze a model to determine its model dependencies. After you identify these dependencies, you can:

- View the files required by your model in a "manifest" file.

- Package the model with its required files into a zip file to send to another Simulink user.

- Compare older and newer manifests for the same model.

- Save a specific version of the model and its required files in a revision control system.

You can also view the libraries and models referenced by your model in a graphical format using the Model Dependency Viewer. See "Using the Model Dependency Viewer" on page 13-49.

Dependency Analysis

The Simulink Manifest Tools identify required files and list them in an XML file called a *manifest*. When Simulink generates a manifest file, it performs a static analysis on your model, which means that the model does not need to be capable of performing an "update diagram" operation (see "Updating a Block Diagram" on page 1-13).

You can specify the type of dependencies you want to detect when you generate the manifest. For more information on what the tool analyzes, refer to the following sections:

- "Analysis Options" on page 13-32
- "Analysis Limitations" on page 13-34
- "M-Code Analysis" on page 13-35
- "Special Cases" on page 13-35

Analysis Options

The Simulink Manifest Tools allow you to specify the scope of analysis when generating the manifest. The dependencies identified by the analysis depend upon the scope you specify.

The following table describes the Analysis Scope options.

Check Box Option	Description
Find model references	Searches for Model blocks in the model, and identifies any referenced models as dependencies.
Find library links	Searches for links to library blocks in the model, and identifies any library links as dependencies.
Find S-functions	Searches for S-Function blocks in the model, and identifies S-function files (M-code and C) as dependencies.

Check Box Option	Description
Analyze model and block callbacks (including "MATLAB Fcn" blocks)	Searches for file dependencies introduced by the M-code in MATLAB Fcn blocks, block callbacks, and model callbacks. For more detail on how callbacks are analyzed, see "M-Code Analysis" on page 13-35.
Find files required for code generation	Searches for file dependencies introduced by Real-Time Workshop® custom code, and Real-Time Workshop® Embedded Coder™ templates.
Find data files (e.g. in "From File" blocks)	Searches for explicitly referenced data files, such as those in From File blocks, and identifies those files as dependencies. See "Special Cases" on page 13-35.
Analyze Stateflow charts	Searches for file dependencies introduced through the use of syntax such as `ml.mymean(myvariable)` in models that use Stateflow®.
Analyze code in Embedded MATLAB blocks	Searches for Embedded MATLAB Function blocks in the model, and identifies any file dependencies (outside toolboxes) introduced in the M-code. Toolbox dependencies introduced by an Embedded MATLAB™ Function block are not detected.

Check Box Option	Description
Find requirements documents	Searches for requirements documents linked using the Requirements Management Interface. For more information, see "Managing Model Requirements" in the *Simulink® Verification and Validation™ User's Guide*. **Note** This option is disabled if your installation does not include a Simulink Verification and Validation license, and Simulink ignores any requirements links in your model.
Analyze files in "user toolboxes"	Searches for file dependencies introduced by files in user-defined toolboxes. See "Special Cases" on page 13-35.
Analyze M-files	Searches for file dependencies introduced by M-files called from the model. For example, if this option is selected and you have a callback to mycallback.m, then the referenced file mycallback.m is also analyzed for further dependencies. See "M-Code Analysis" on page 13-35.

Analysis Limitations

The analysis might not find all files required by your model (for examples, see "M-Code Analysis" on page 13-35).

The analysis might not report certain blocksets or toolboxes required by a model. You should be aware of this limitation when sending a model to another user. Blocksets that do not introduce dependence on any files (such as Simulink® Fixed Point™) cannot be detected.

To include dependencies that the analysis cannot detect, you can add additional file dependencies to a manifest file using the View/Edit Manifest Contents option (see "Editing Manifests" on page 13-42).

M-Code Analysis

When the Simulink Manifest Tools encounter M-code, for example in a model or block callback, or in an M-file S-function, they attempt to identify the files it references. If those files contain M-code, *and* the analysis scope option **Analyze M-files** is selected, the referenced files are also analyzed. This function is similar to depfun but with some enhancements:

- Files that are in MathWorks™ toolboxes are not analyzed.

- Strings passed into calls to eval, evalc, and evalin are analyzed.

- File names passed to load, fopen, xlsread, importdata, dlmread, csvread, wk1read, textread and imread are identified.

File names passed to load, etc., are identified only if they are literal strings. For example:

```
load('mydatafile')
load mydatafile
```

The following example, and anything more complicated, is not identified as a file dependency:

```
str = 'mydatafile';
load(str);
```

Similarly, arguments to eval, etc., are analyzed only if they are literal strings.

The Simulink Manifest Tools look inside MAT-files to find the names of variables to be loaded. This enables them to distinguish reliably between variable names and function names in block callbacks.

Special Cases

The following list contains additional information about specific cases:

- If your model uses classes created using the Data Class Designer and references a class called `MyPackage.MyClass`, all files inside the directory `@MyPackage` and its subdirectories are added to the manifest.

- A user-defined toolbox must have a properly configured `Contents.m` file. The Simulink Manifest Tools search user-defined toolboxes as follows:

  - If you have a `Contents.m` file in directory `X`, any file inside a sub-directory of `X` is considered part of your toolbox.

  - If you have a `Contents.m` file in directory `X/X`, any file inside all sub-directories of the "outer" directory `X` will be considered part of your toolbox.

  For more information on the format of a `Contents.m` file, see `ver`.

- If your S-functions require TLC files, these are detected.

- If you create a UI using GUIDE and add this to a model callback, then the dependency analysis detects the M-file and .fig file dependencies.

- Various blocksets and toolboxes can introduce a dependence on a file through their additional source blocks. If the analysis scope option **Find data files (e.g. in "From File" blocks)** is selected, the analysis detects file dependencies introduced by the following blocks:

Product	Blocks
Signal Processing Blockset™	From Wave File block (Microsoft® Windows® operating system only)
	From Multimedia File block (Windows only)
Video and Image Processing Blockset™	Image From File block
	Read Binary File block
Virtual Reality Toolbox™	VR Sink block

  The option **Find data files** also detects dependencies introduced by setting a "Model Workspace" for a model to either `MAT-File` or `M-Code`.

- To programmatically check for file dependencies, use the method `dependencies.fileDependencyAnalysis` as follows.

```
[files, missing, depfile] =
dependencies.fileDependencyAnalysis('modelname')
```

This returns the following:

- *files* — a cell-array of strings containing the full-paths of all existing files referenced by the model *modelname*.

- *missing* — a cell-array of strings containing the names all files that are referenced by the model *modelname*, but cannot be found.

- *depfile* — the full-path of the file containing information about any user-defined files associated with the model *modelname*.

If you try this analysis on a demo model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the demo models.

Best Practices for Dependency Analysis

The starting point for dependency analysis is the model itself. Make sure that the model refers to any data files it needs, even if you would normally load these manually. For example, add code to the model's PreLoadFcn to load them automatically, e.g.,

```
load mydatafile
load('my_other_data_file.mat')
```

This way, the Simulink Manifest Tools can add them to the manifest. For more detail on callback analysis, see the notes on M-code analysis (see "M-Code Analysis" on page 13-35).

More generally, ensure that the model creates or loads any variables it uses, either in model callbacks or in scripts called from model callbacks. This reduces the possibility of the Simulink Manifest Tools confusing variable names with function names when analyzing block callbacks.

If you plan to export the manifest after creating it, ensure that the model does not refer to any files by their absolute paths, for example:

```
load C:\mymodel\mydata\mydatafile.mat
```

13-37

Absolute paths can become invalid when you export the model to another machine. If referring to files in other directories, do it by relative path, for example:

```
load mydata\mydatafile.mat
```

Select **Preserve directory hierarchy** when exporting, so that the exported files are in the same locations relative to each other. Also, choose a root directory so that all the files listed in the manifest are inside it. Otherwise, any files outside the root are copied into a new directory called `external` underneath the root, and relative paths to those files become invalid.

If you are exporting a model that uses an M-file inside a MATLAB® class (in a directory called `@myclass`, for example), you must select the **Preserve directory hierarchy** check box when exporting, to maintain the directory structure of the class.

Always test exported zip files by extracting the contents to a new location on your computer and testing the model. Be aware that in some cases required files may be on your path but not in the zip file, if your path contains references to directories other than MathWorks toolboxes.

Generating Manifests

Generating a manifest performs the dependency analysis and saves the list of model dependencies to a manifest file. You must generate the manifest before using any of the other Simulink Manifest Tools.

Note The model dependencies identified in a manifest depend upon the Analysis scope you specify. For example, performing an analysis without selecting **Find Library Links** may not find all the Simulink Blocksets that your model requires, since these are often included in a model as library links.

You should always look at the **Dependency analysis settings:** section at the top of the Model Manifest Report to see the scope of analysis used to generate it. For more information, see "Viewing the Model Manifest Report" on page 13-40.

To generate a manifest:

1 Select **Tools > Model Dependencies > Generate Manifest**.

The Generate Model Manifest dialog box appears.
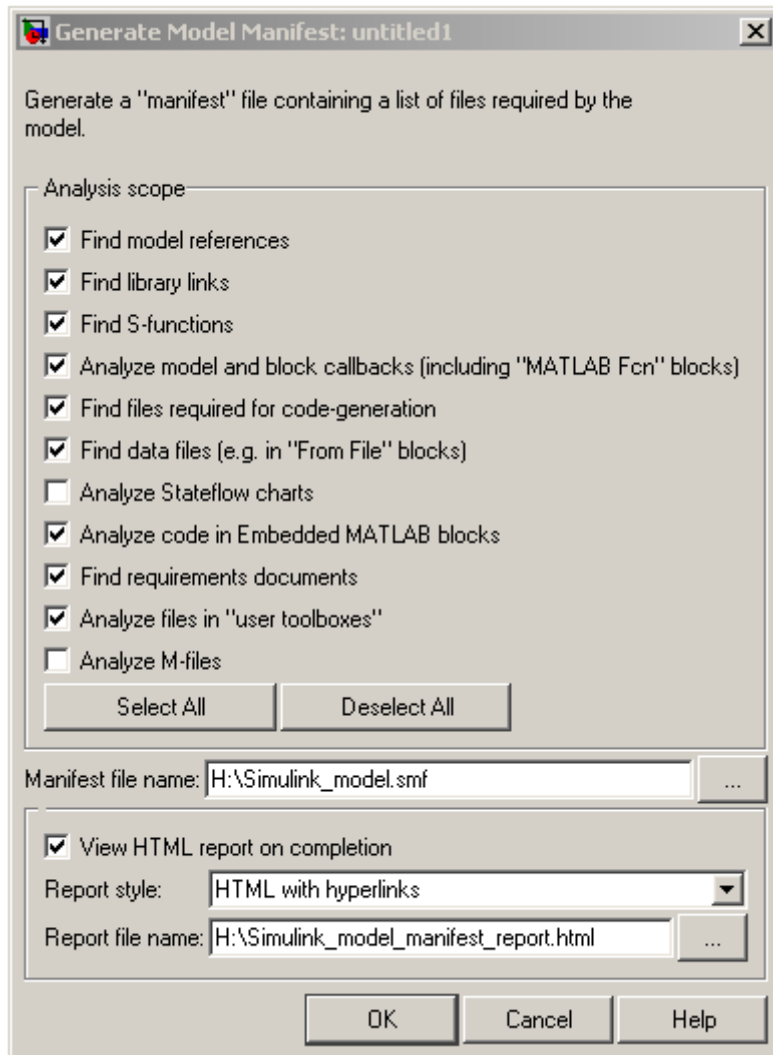
2 Select the **Analysis scope** check boxes to specify the type of dependencies you want to detect (see "Analysis Options" on page 13-32).

3 Specify the **Manifest file name** and location in which to save the file.

4 To generate a report when you generate the manifest, select **View HTML report on completion**, then specify the Report style (`Plain HTML` or `HTML with Hyperlinks`) and Report file name.

5 Click **OK**.

Simulink generates a manifest file containing a list of the model dependencies.

> **Note** The manifest is an XML file with the extension `.smf` located (by default) in the same directory as the model itself.

Viewing the Model Manifest Report

If you selected **View HTML report on completion**, the Model Manifest Report appears after Simulink generates the manifest. The report shows details of the analysis scope under the heading **Dependency analysis settings**, lists the required files and toolboxes, and provides details of references to other files so you can identify where dependencies arise.

Model Manifest Report: powertrain

Analysis performed: 15-Jun-2007 11:18:41

Dependency analysis settings:

- Analyze files in MathWorks toolboxes: **false**
- Analyze files in user-defined toolboxes: **true**
- Analyze M-files: **true**
- Find model references: **true**
- Find library links: **true**
- Find S-functions: **true**
- Analyze model and block callbacks: **true**
- Find code-generation files: **true**
- Find data files: **true**
- Analyze Stateflow charts: **true**
- Analyze Embedded MATLAB code: **true**
- Find Requirements documents: **false**

Model Reference and Library Link hierarchy

- powertrain
    - menginemodel
    - mtransmission
        - mtransmission_ratio
        - mpowertrainlib
    - mpowertrainlib

Files used by this model

File Name	Size	Last Modified Date	Will be Exported
c:/work/powertrain.mdl _(open)_	52071 bytes	2007-06-15 11:15:11	true
c:/work/powertrain_data.mat _(open)_	1976 bytes	2006-10-27 12:12:02	true
c:/work/mpowertrainlib.mdl _(open)_	34759 bytes	2006-08-08 09:13:14	true
c:/work/menginemodel.mdl _(open)_	19260 bytes	2006-08-08 09:13:11	true
c:/work/mtransmission.mdl _(open)_	29829 bytes	2006-08-08 09:13:15	true
c:/work/msfunctiondemo.m _(open)_	11549 bytes	2007-06-01 20:52:18	true
c:/work/mtransmission_ratio.mdl _(open)_	20390 bytes	2006-10-27 12:11:58	true

Toolboxes required by this model

- MATLAB (7.5)
- Simulink (7.0)
- Stateflow (7.0)

References in this model

Use the table below to determine where in a model a dependence upon a particular file or toolbox originates.

Reference Type	Reference Location	File Name
ModelReference	powertrain/engine model !! (show)	c:/work/menginemodel.mdl (open)
ModelReference	powertrain/transmission (show)	c:/work/mtransmission.mdl (open)
LibraryLink	powertrain/Library Shift Logic (show)	c:/work/mpowertrainlib.mdl (open)
ModelCallback,PreLoadFcn	powertrain (show)	c:/work/powertrain_data.mat (open)
MFile	c:/work/msfunctiondemo.m (show)	c:/MATLAB/matlab/toolbox/matlab/demos/peaks.m (open)
Stateflow	mpowertrainlib/shift_logic/shift_logic (show)	c:/MATLAB/matlab/toolbox/stateflow/stateflow/sf_sfun.mexa64 (open)
ModelReference	mtransmission/transmission ratio (show)	c:/work/mtransmission_ratio.mdl (open)
LibraryLink	mtransmission/Grouped Unit Delay (show)	c:/work/mpowertrainlib.mdl (open)
LibraryLink	mtransmission/Torque Converter/Torque Conversion (show)	c:/work/mpowertrainlib.mdl (open)
MSFunction	mtransmission_ratio/Level-2 M-file S-Function (show)	c:/work/msfunctiondemo.m (open)

Directories referenced by this model

References

(This model does not refer to any directories)

Actions

- Re-generate this manifest
- Edit this manifest
- Compare this manifest with another one
- Export the files in this manifest to a ZIP file

Editing Manifests

After you generate a manifest, you can view the list of files identified as dependencies, and manually add or delete files from the list.

To edit the list of required files in a manifest:

1 Select **Tools > Model Dependencies > View/Edit Manifest Contents**.

The View and Edit Manifest dialog box appears, showing the latest manifest for the current model.
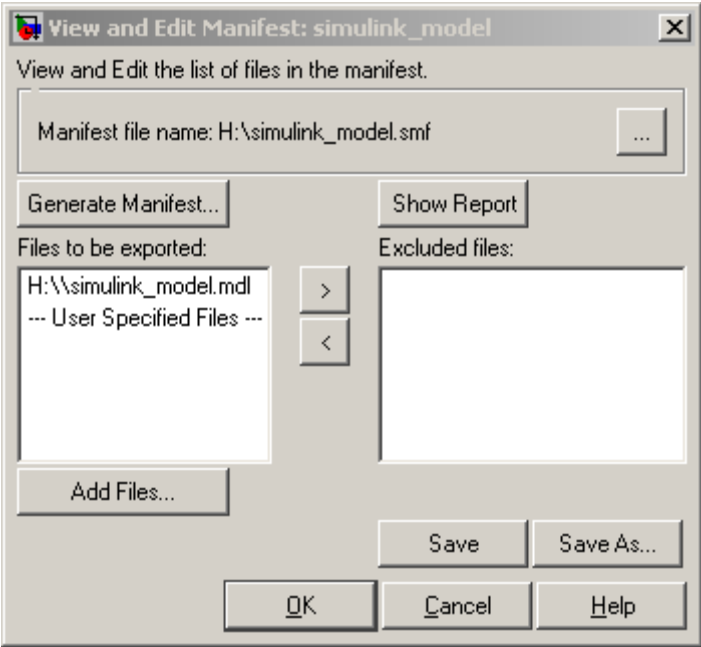
Note You can open a different manifest by clicking the Browse for

manifest file button [...]. If you have not generated a manifest, select **Generate Manifest** to open the Generate Model Manifest dialog box (see "Generating Manifests" on page 13-38).

2 Examine the **Files to be exported** list on the left side of the dialog box. This list shows the files identified as dependencies.

3 To add a file to the manifest:

a Click **Add Files**.

The Add Files to Manifest dialog box opens.

b Select the file you want to add, then click **Open**.

The selected file is added to the **Files to be exported** list.

4 To remove a file from the manifest:

a Select the file you want to remove from the **Files to be exported** list.

b Click the Exclude selected files button .

The selected file is moved to the **Excluded files** list.

> **Note** If you add a file to the manifest and then exclude it, that file is removed from the dialog (it is not added to the **Excluded files** list). Only files detected by the Simulink Manifest Tools are included in the Excluded files list.

5 Click **Save** to save your changes to the manifest file.

Simulink saves the manifest (`.smf`) file, and creates a user dependencies (`.smd`) file that stores the names of any files you added manually, as well as those you manually excluded. Simulink uses the `.smd` file to remember your changes the next time you generate a manifest. The user dependencies (`.smd`) file has the same name and directory as the model. By default, the user dependencies (`.smd`) file is also included in the manifest.

> **Note** If the user dependencies (`.smd`) file is read-only, a warning is displayed when you save the manifest.

6 To view the Model Manifest Report for the updated manifest, click **Show Report**.

An updated Model Manifest Report appears, listing the required files and toolboxes, and details of references to other files. See "Viewing the Model Manifest Report" on page 13-40 for an example.

7 When you are finished editing the manifest, click **OK**.

Comparing Manifests

You can compare two manifests to see how the list of model dependencies differs between two models, or between two versions of the same model.

To compare manifest files:

1 Select **Tools > Model Dependencies > Compare Manifests**.

The Compare Manifests dialog box appears.
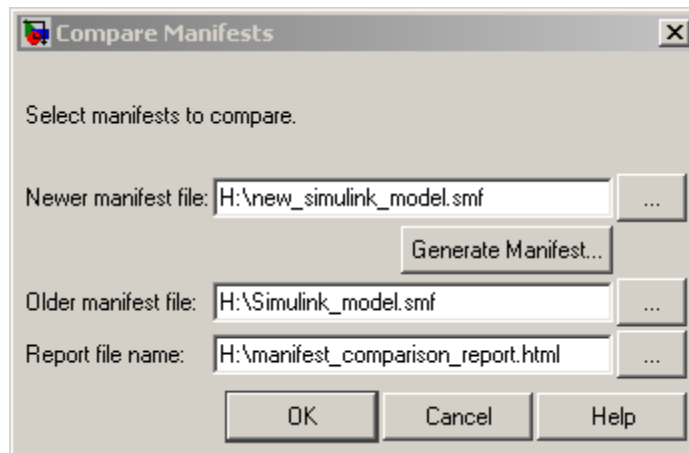
2 Select the newer manifest file.

Note You can click **Generate Manifest** to create a new manifest. See "Generating Manifests" on page 13-38 for more information. After you generate the manifest, you return to the Compare Manifests dialog box.

3 Select the older manifest file.

4 Specify a report file name and location.

Note The default report file is `manifest_comparison_report.html` in the current working directory.

---

5 Click **OK**.

The two manifests are compared and displays the Model Manifest Differences Report. The report provides details about each manifest file, and lists the differences between the files.

Model Manifest Differences Report

First Manifest

    File: H:\new_simulink_model.smf

    Analysis performed: 07-Jun-2007 12:06:03

    Simulink block diagram names:

        • simulink_model

Second Manifest

    File: H:\Simulink_model.smf

    Analysis performed: 07-Jun-2007 10:58:18

    Simulink block diagram names:

        • simulink_model

Files in first manifest only:

File Name	Size	Last Modified Date
H:\simulink_model.smd	172 bytes	2007-06-07 12:04:58

Files in second manifest only:

File Name	Size	Last Modified Date
H:\test.mdl	19230 bytes	2007-06-07 10:57:10

Modified files

(none)

Unmodified files

(These are the files which appear in both manifests and have the same date-stamps in both cases.)

File Name	Size	Last Modified Date
H:\\simulink_model.mdl	19250 bytes	2007-06-07 10:57:42

Directories included in first manifest only:

(none)

Directories included in second manifest only:

(none)

Directories included in both manifests:

(none)

Exporting Files in a Manifest

You can export copies of the files listed in the manifest to a zip file. Exporting the files allows you to package the model with its required files into a single zip file, so you can easily send it to another user or save it in a revision control system.

To export your model with its required files:

1 Select **Tools > Model Dependencies > Export Files in Manifest**.

The Export Files in Manifest dialog box appears, showing the latest manifest for the current model.
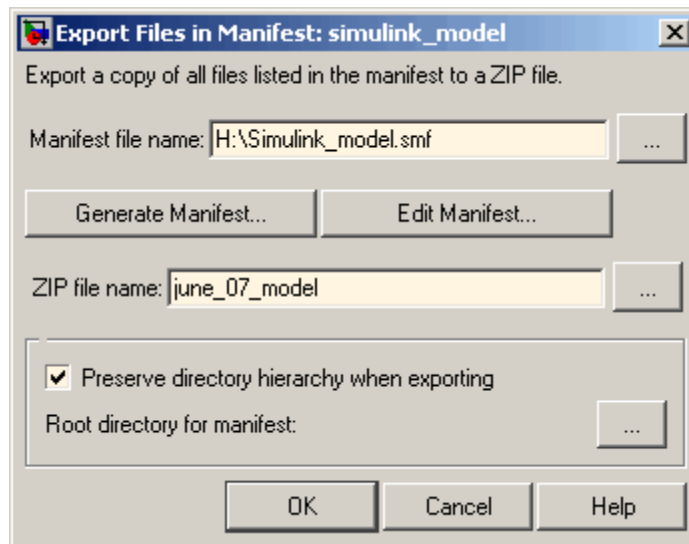
Note You can export a different manifest by clicking the Browse for manifest file button [...]. If you have not generated a manifest, select **Generate Manifest** to open the Generate Model Manifest dialog box (see "Generating Manifests" on page 13-38).

---

2 If you want to view or edit the manifest before exporting it, select **Edit Manifest** to view or change the list of required files. See "Editing Manifests" on page 13-42. When you close the View and Edit Manifest dialog box, you return to the Export Files in Manifest dialog box.

3 Select **Preserve directory hierarchy** if you want to keep directory structure for your exported model and files. Then, select the root directory to use for this structure.

Note You must select **Preserve directory hierarchy** if you are exporting a model that uses an M-file inside a MATLAB class (to maintain the directory structure of the class), or if the model refers to files in other directories (to ensure the exported files maintain the same relative paths).

---

4 Enter the zip file name to which you want to export the model.

5 Click **OK**.

The model and its file dependencies are exported to the specified zip file.

Using the Model Dependency Viewer

The Model Dependency Viewer displays a dependency view of a model. The dependency view is a graph of all the models and libraries referenced directly or indirectly by the model. You can use the dependency view to quickly find and open referenced libraries and models.

See the following topics for information on using the viewer:

- "About Model Dependency Views" on page 13-50
- "Opening the Model Dependency Viewer" on page 13-55
- "Manipulating a Dependency View" on page 13-56
- "Browsing Dependencies" on page 13-61
- "Saving a Dependency View" on page 13-61
- "Printing a Dependency View" on page 13-62

About Model Dependency Views

The Model Dependency Viewer allows you to choose between the following types of dependency views of a model reference hierarchy.

File Dependency View. A *file dependency* view shows the model and library files referenced by a top model. A referenced model or library file appears only once in the view even if it is referenced more than once in the model hierarchy displayed in the view. A file dependency view consists of a set of blocks connected by arrows. Blue blocks represent model files; brown boxes, libraries. Arrows represent dependencies. For example, the arrows in the following view indicate that the aero_guidance model references two libraries: aerospace and simulink_need_slupdate.
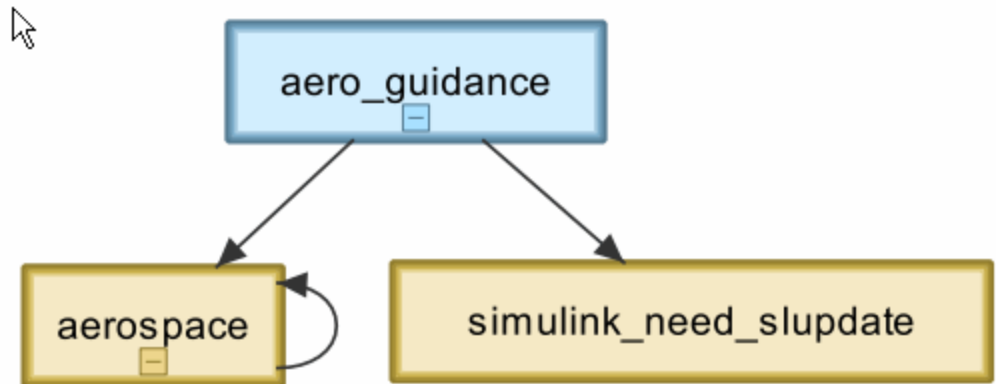
An arrow that points to the library from which it emerges indicates that the library references itself, i.e., blocks in the library reference other blocks in that same library. For example, the preceding view indicates that the aerospace library references itself.

A file dependency view optionally includes a legend that identifies the model in the view and the date and time the view was created.

Legend

Dependency viewer: aero_guidance
06-Jul-2007 16:17:10

Model

Library

aero_guidance

aerospace

simulink_need_slupdate
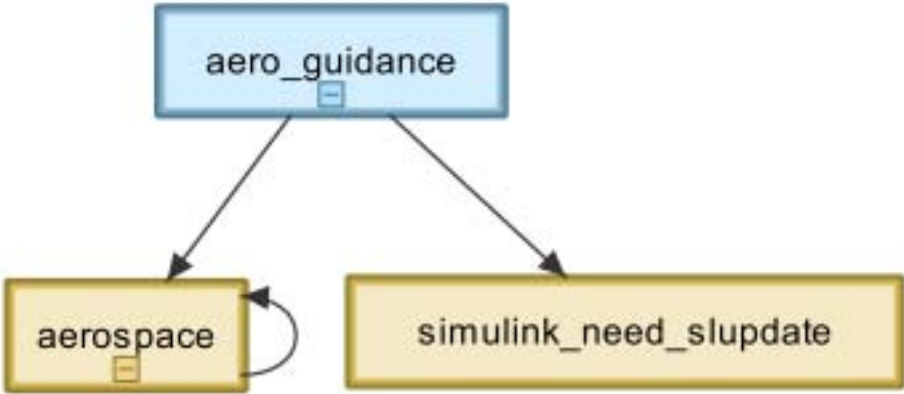
Referenced Model Instances View. A *referenced model instances* view shows every reference to a model in a model reference hierarchy (see Chapter 5, "Referencing a Model") rooted at the top model targeted by the view. If a model hierarchy references the same model more than once, the referenced model appears multiple times in the instance view, once for each reference. For example, the following view indicates that the model reference hierarchy rooted at sldemo_mdlref_depgraph contains two references to the model sldemo_mdlref_F2C.
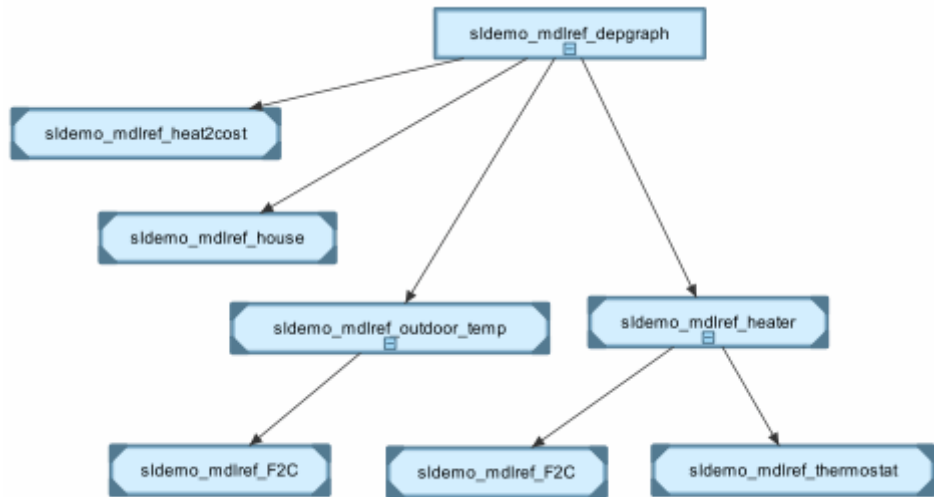
In an instance view, boxes represent a top model and model references.
Boxes representing accelerated-mode instances (see "Referenced Model
Simulation Modes" on page 5-13) have filled triangles in their corners; boxes
representing normal-mode instances, have unfilled triangles in their corners.
For example, the following diagram indicates that one of the references to
sldemo_mdlref_F2C operates in normal mode; the other, in accelerated mode.
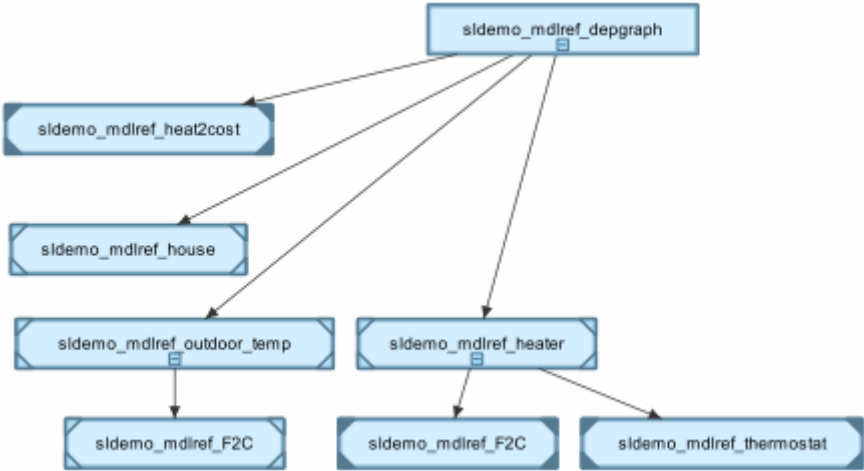
An instance view displays warning or error icons on instance boxes to indicate invalid normal-mode references (see "Mixing Simulation Modes" on page 5-14).

Icon	Indicates
⚠	A reference configured to run in normal mode actually runs in accelerated mode because it is directly or indirectly referenced by another model reference that runs in accelerated mode.
⬟	One of multiple normal-mode references to the same model in a model reference hierarchy. Such a reference is invalid because it violates the rule that only one normal-mode reference to a model can occur in a given model reference hierarchy.

The following instance view, for example, indicates that the accelerated-mode configuration of the reference to sldemo_mdlref_heater overrides the normal-mode configurations of its reference to sldemo_mdlref_F2C.
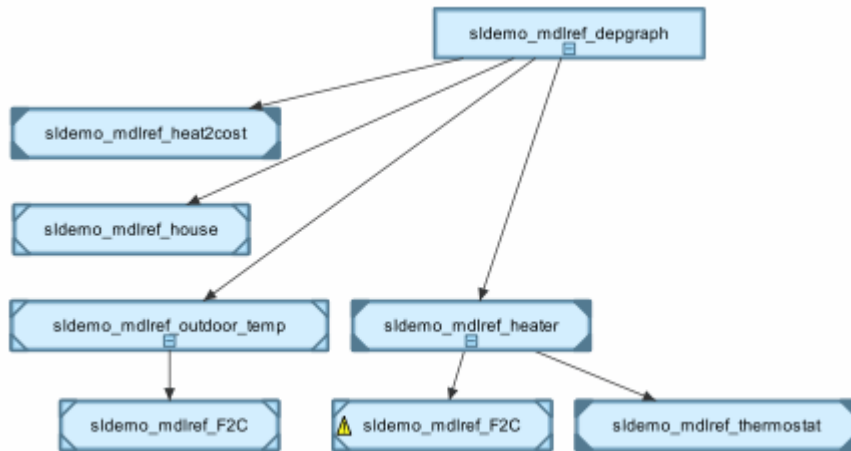
Changing the simulation mode of the reference to sldemo_mdlref_heater to normal creates two normal-mode references in the model reference hierarchy to the same model, i.e., sldemo_mdlref_F2C, which is invalid as indicated in the following instances view.
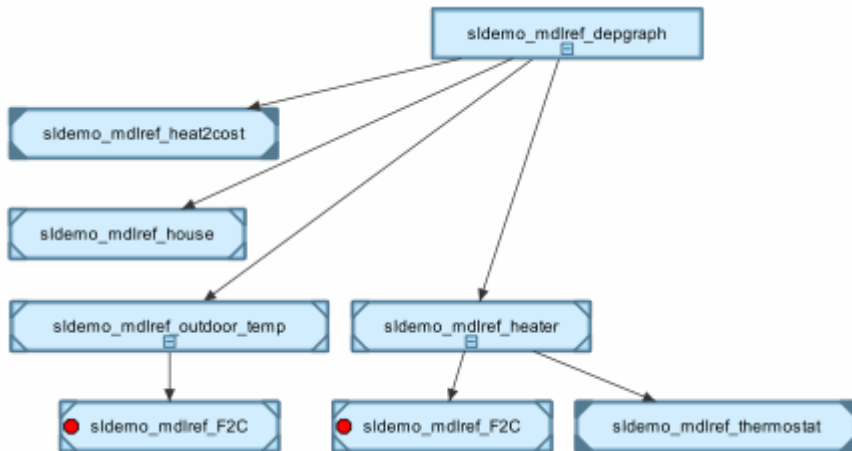
Opening the Model Dependency Viewer

The Model Dependency Viewer displays a graph of all the models and libraries referenced directly or indirectly by the model. You can use the dependency viewer to quickly find and open referenced libraries and models.

To display a dependency view for a model:

1 Open the model.

2 Select **Tools > Model Dependencies > Model Dependency Viewer**, then select the type of view you want to see:

- **.mdl File Dependencies Including Libraries**
- **.mdl File Dependencies Excluding Libraries**
- **Referenced Model Instances**

The Model Dependency Viewer appears, displaying a dependency view of the model.
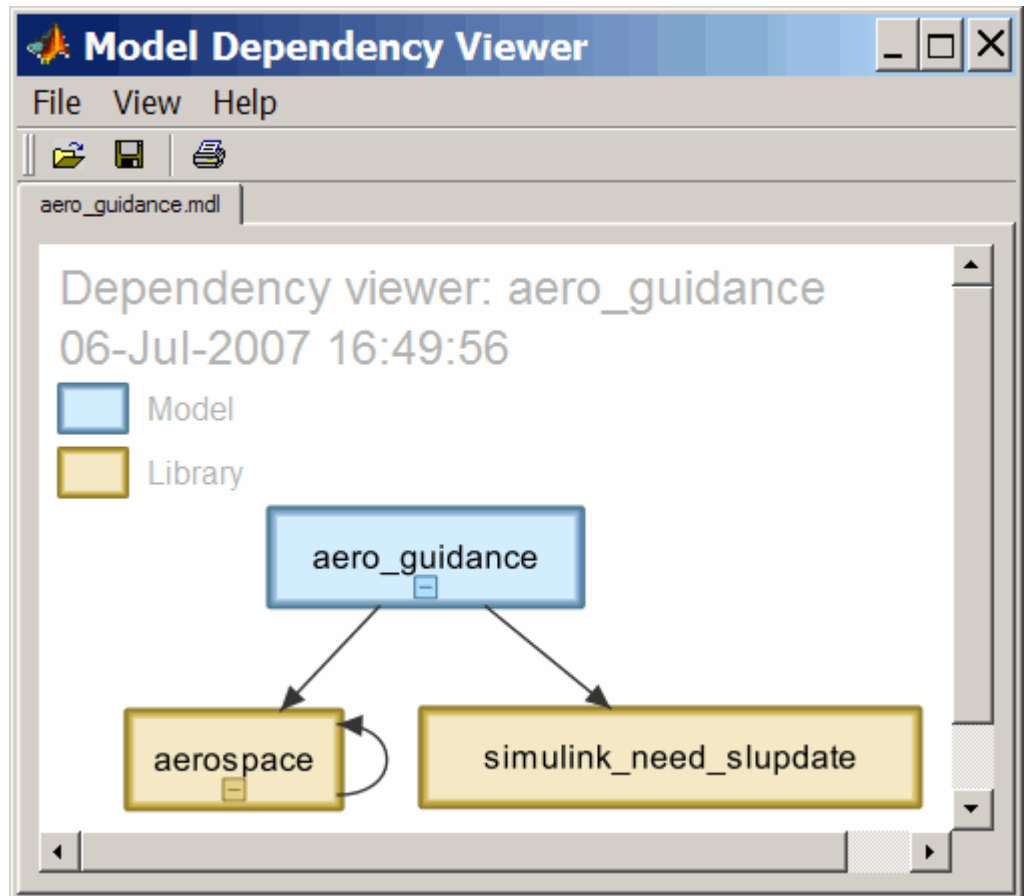
Manipulating a Dependency View

The Model Dependency Viewer allows you to manipulate dependency views in various ways. See the following topics for more information:

- "Changing Dependency View Type" on page 13-57

- "Excluding Block Libraries from a File Dependency View" on page 13-57

- "Including Simulink® Blocksets in a File Dependency View" on page 13-57

- "Changing View Orientation" on page 13-57

- "Expanding or Collapsing Views" on page 13-58
- "Zooming a Dependency View" on page 13-58
- "Moving a Dependency View" on page 13-59
- "Rearranging a Dependency View" on page 13-59
- "Displaying and Hiding a Dependency View's Legend" on page 13-59
- "Displaying Full Paths of Referenced Model Instances" on page 13-60
- "Refreshing a Dependency View" on page 13-61

Changing Dependency View Type. You can change the type of dependency view displayed in the viewer.

To change the type of dependency view:

- Select **View > Dependency Type > .mdl File Dependencies** (see "File Dependency View" on page 13-50)

  or

- Select **View > Dependency Type > Referenced Model Instances** (see "Referenced Model Instances View" on page 13-51).

Excluding Block Libraries from a File Dependency View. By default a file dependency view includes libraries on which a model depends.

To exclude block libraries:

- Deselect **View > Include Libraries**.

Including Simulink Blocksets in a File Dependency View. By default, a file dependency view omits block libraries supplied by The MathWorks™ when **View > Include Libraries** is selected.

To include libraries supplied by The MathWorks:

- Select **View > Show MathWorks Dependencies**.

Changing View Orientation. By default the orientation of the dependency graph displayed in the viewer is vertical.

To change the orientation to horizontal:

- Select **View > Orientation > Horizontal**.
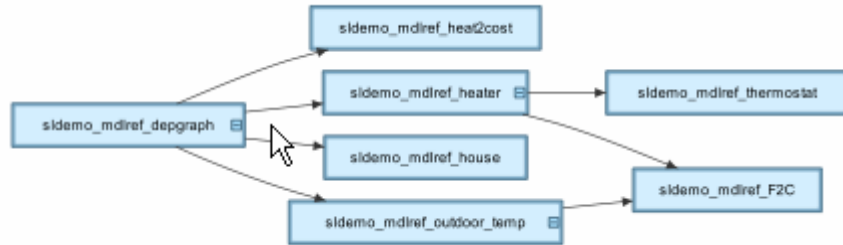
Expanding or Collapsing Views. You can expand or collapse each model or library in the dependency view to display or hide the dependencies for that model or library.

To expand or collapse views:

- Click the expand(+)/collapse(-) button on the box representing the model or library to expand or collapse that view.
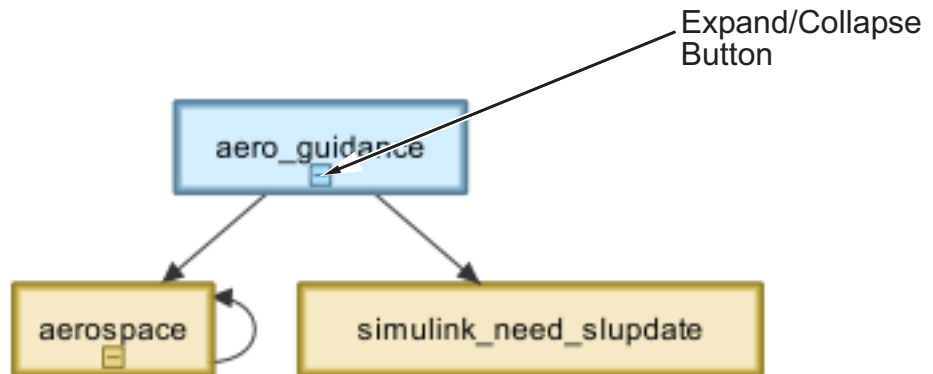
Zooming a Dependency View. You can enlarge or reduce the size of the dependency graph displayed in the viewer.

To zoom a dependency view in or out, do either of the following:

- Press and hold down the **spacebar** key and then press the **+** or **-** key on the keyboard.

- Move the scroll wheel on your mouse forward or backward.

To fit the view to the viewer window:

- Press and release the **spacebar** key.

Moving a Dependency View. You can move a dependency view to another location in the viewer window.

To move the dependency view:

1 Move the cursor over the view.

2 Press your keyboard's space bar and your mouse's left button simultaneously.

3 Move the cursor to drag the view to another location.

Rearranging a Dependency View. You can rearrange a dependency view by moving the blocks representing models and libraries. This can make a complex view easier to read.

To move a block to another location:

1 Select the block you want to move by clicking it.

2 Drag and drop the block in the new location.

Displaying and Hiding a Dependency View's Legend. The dependency view can display a legend that identifies the model in the view and the date and time the view was created.

To display or hide a dependency view's legend:

- Select **View > Show Legend** from the viewer's menu bar.

Displaying Full Paths of Referenced Model Instances. In an instance view (see "Referenced Model Instances View" on page 13-51) , you can display the full path of a model reference in a tooltip or in the box representing the reference.

To display the full path in a tooltip:

- Move the cursor over the box representing the reference in the view.

  A tooltip appears, displaying the path displays the full path of the Model block corresponding to the instance.
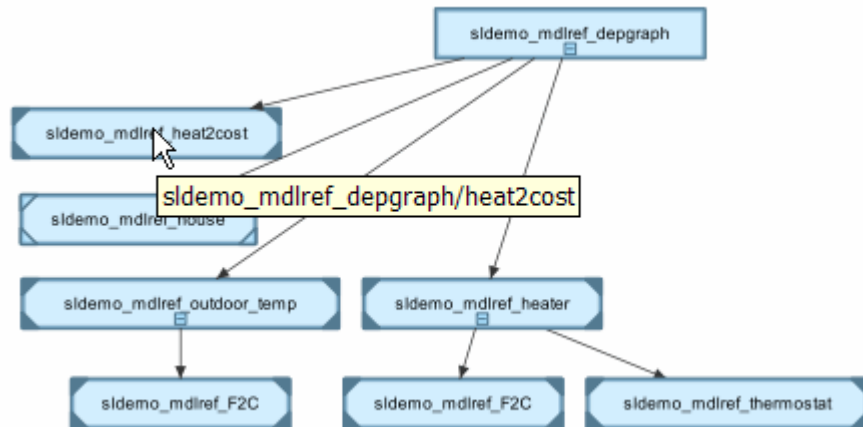
To display full paths in the boxes representing the instances:

- Select **View > Show Full Path**.

  Each box in the instance view now displays the path of the Model block corresponding to the instance. The name of the referenced model appears in parentheses as illustrated in the following figure.
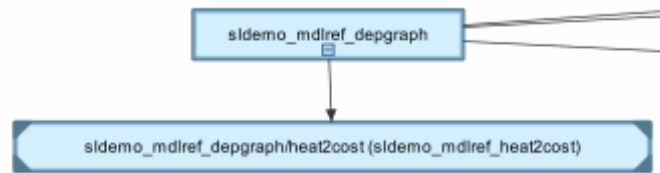
Refreshing a Dependency View. After changing a model displayed in a dependency view or any of its dependencies, you must update the view to reflect any dependency changes.

To update the view:

- Select **View > Refresh** from the dependency viewer's menu bar.

Browsing Dependencies

You can use a dependency view to browse a model's dependencies:

- To open a model or library displayed in the view, double-click its block.

- To display the Model block corresponding to an instance in an instance view, right-click the instance and select **Highlight Block** from the menu that appears.

- To open all models displayed in the view, select **File > Open All Models** from the viewer's menu bar.

- To save all models displayed in the view, select **File > Save All Models**.

- To close all models displayed in the view, select **File > Close All Models**.

Saving a Dependency View

You can save a dependency view for later viewing.

To save the current view:

- Select **File > Save As** from the viewer's menu bar, then enter a name for the view.

To reopen the view:

- Select **File > Open** from any viewer's menu bar, then select the view you want to open.

Printing a Dependency View

To print a dependency view:

- Select **File > Print** from the dependency viewer's menu bar.

14

Running Simulations

Simulation Basics

You can simulate a model at any time simply by clicking the **Start** button
on the Model Editor displaying the model (see "Starting a Simulation"
on page 14-4). 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.
With the Simulink® software you can create multiple model configurations,
called configuration sets, modify existing configuration sets, and switch
configuration sets with a click of a mouse button (see "Configuration Sets" on
page 14-37 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. The simulation runs from the specified
start time to the specified stop time. While the simulation is running, you can
interact with the simulation in various ways, stop or pause the simulation (see
"Pausing or Stopping a Simulation" on page 14-6), and launch simulations
of other models. If an error occurs during a simulation, the simulation is
halted and a diagnostic viewer pops up that helps you to determine the cause
of the error.

Controlling Execution of a Simulation

In this section...

Starting a Simulation

This sections explains how to run a simulation interactively. See "Running a Simulation Programmatically" on page 14-74 for information on running a simulation from a program, S-function, or the MATLAB® command line.

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.
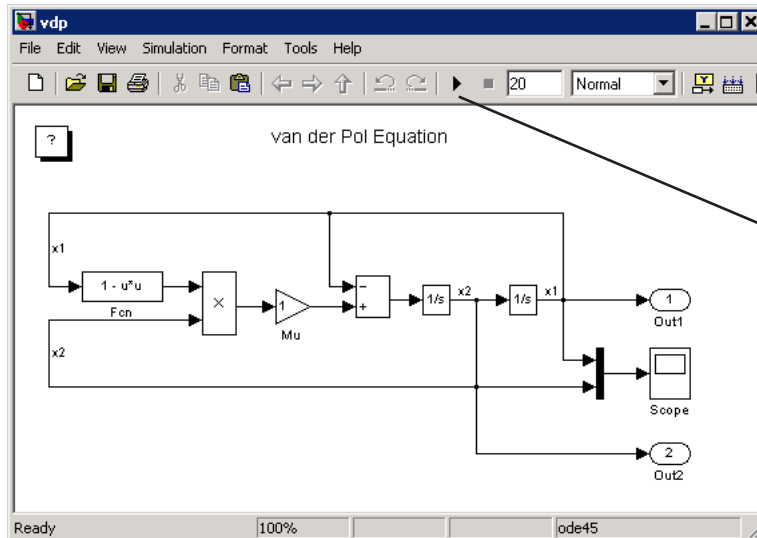
Start button

> **Note** A common mistake that new 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.

The model execution begins 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").

While the simulation is running, a progress bar at the bottom of the model window shows how far the simulation has progressed. A **Stop** command replaces the **Start** command on the **Simulation** menu. A **Pause** command appears on the menu and replaces the **Start** button on the model toolbar.
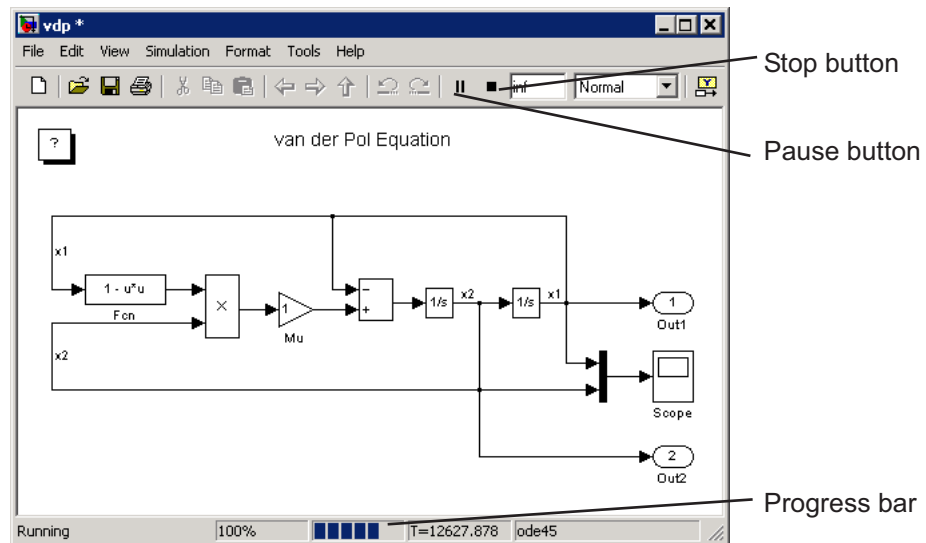
Your computer beeps to signal the completion of the simulation.

Pausing or Stopping a Simulation

Select the **Pause** command or button to pause the simulation. The execution of the current time step is completed and the simulation is suspended. 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. Simulink completes The execution of the current time step is completed before the mode is terminated. 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, the Simulink software writes the data when the simulation is terminated or suspended.

Using Blocks to Stop or Pause a Simulation

Using Stop Blocks

You can use the Stop Simulation block to terminate a simulation when the 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.
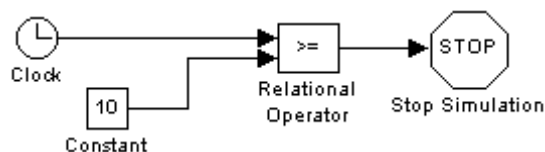
If the block input is a vector, any nonzero element causes the simulation to terminate.

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.
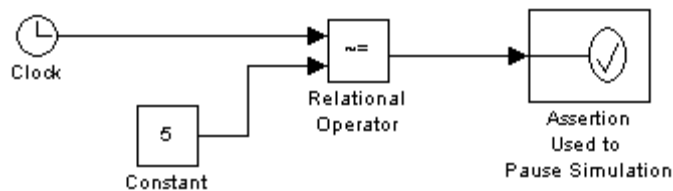
When the simulation pauses, the Assertion block displays the following message at the MATLAB command line.

```
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 14-74 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 computer running the Microsoft® Windows® operating system (see "Block Data Tips" on page 6-2).

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

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" for more information. On computers running the Microsoft® Windows® operating system, 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

In this section...
"What is a Solver?" on page 14-11
"Choosing a Solver Type" on page 14-11
"Choosing a Fixed-Step Solver" on page 14-13
"Choosing a Variable-Step Solver" on page 14-17

What is a Solver?

A solver is a component of the Simulink® software that determines the next time step that a simulation needs to take to meet target accuracy requirements that you specify. The Simulink product provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. The following sections explain how to choose the solver best suited to your application. For information on tailoring the selected solver to your model, see "Improving Simulation Accuracy" on page 14-72.

Choosing a Solver Type

Solvers are divided into two types: fixed-step and variable-step. Both types of solvers compute the next simulation time as the sum of the current simulation time and a quantity known as the step size. With a fixed-step solver, the step size remains constant throughout the simulation. 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").

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

14-11

cause the simulation to miss error conditions that can occur on a real-time computer system.

If you do not plan to deploy your model as generated code, the choice between a variable-step and a fixed-step solver depends on the dynamics of your model. If your model's states change rapidly or contain discontinuities, a variable-step solver can shorten the time required to simulate your model significantly. This is because, for such a model, a variable-step solver can require fewer time steps than a fixed-step solver to achieve a comparable level of accuracy.

The following model illustrates how a variable-step solver can shorten simulation time for a multirate discrete model.
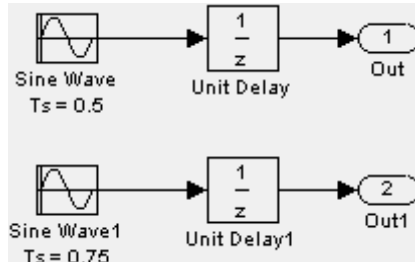
This model generates outputs at two different rates, every 0.5 second and every 0.75 second. To capture both outputs, the fixed-step solver must take a time step every 0.25 second (the *fundamental sample time* for the model).

```
[0.0 0.25 0.5 0.75 1.0 1.25 ...]
```

By contrast, the variable-step solver need take a step only when the model actually generates an output.

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

This significantly reduces the number of time steps required to simulate the model.

The variable-step discrete solver uses zero-crossing detection (see "Zero-Crossing Detection" on page 2-20) to handle continuous signals. This solver is used 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 provided. 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 the Simulink software to choose the size of the step size (the default) or you can choose the step size yourself. If you allow the Simulink software to choose the step size, it is set 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, an error message is displayed. Thus, updating or simulating a model is a quick way to determine whether it has continuous states.

14-13

About Fixed-Step Continuous Solvers

A set of fixed-step continuous solvers are provided 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, the fixed-step discrete solver is always used for a model that contains no states or only discrete states, even if you specify a fixed-step continuous solver for the model.

Two distinct types of fixed-step continuous solvers are provided: explicit and implicit solvers. Explicit solvers (see "Explicit Fixed-Step Continuous Solvers" on page 14-14) 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 14-16) 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. A set of explicit fixed-step continuous solvers are provided. The solvers differ in the specific integration technique used to compute the model's state derivatives. The following table lists the available solvers and the integration techniques they use.

Solver	Integration Technique
ode1	Euler's Method
ode2	Heun's Method
ode3	Bogacki-Shampine Formula
ode4	Fourth-Order Runge-Kutta (RK4) Formula
ode5	Dormand-Prince Formula

The integration techniques used by the fixed-step continuous solvers trade accuracy for computational effort. The table lists the solvers in order of the computational complexity of the integration methods they use from least complex (ode1) to most complex (ode5).

As with the fixed-step discrete solver, the accuracy and length of time of a simulation driven by a fixed-step continuous solver depends on the size of the steps taken by the solver: the smaller the step size, the more accurate the results but the longer the simulation takes. For any given step size, the more computationally complex the solver, the more accurate the simulation.

If you specify a fixed-step solver type for a model, the solver's model is set 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, by default the step size is set 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. One solver is provideed 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. You can specify the number of Newton's method iterations and the extrapolation order that the solver uses to compute the next value of a model state (see "Fixed step size (fundamental sample time)""Fixed-step solver size (fundamental sample time)" in the online documentation). The more iterations and the higher the extrapolation order that you select, the greater the accuracy but also the greater the computational burden per step size.

Choosing a Fixed-Step Continuous Solver

Any of the fixed-step continuous solvers in the Simulink product can simulate a model to any desired level of accuracy, given 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 provided 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 is provided. As with fixed-step solvers, 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, the discrete solver will be used to simulate the model even if the model specifies a continuous solver.

About Variable-Step Continuous Solvers

The variable-step solvers in the Simulink product 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.

The following variable-step continuous solvers are provided:

- ode45 is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver; that is, in computing $y(t_n)$, it

needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, ode45 is the best solver to apply as a first try for most problems. For this reason, ode45 is the default solver used for models with continuous states.

- ode23 is also based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It can be more efficient than ode45 at crude tolerances and in the presence of mild stiffness. ode23 is a one-step solver.

- ode113 is a variable-order Adams-Bashforth-Moulton PECE solver. It can be more efficient than ode45 at stringent tolerances. ode113 is a *multistep* solver; that is, it normally needs the solutions at several preceding time 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

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

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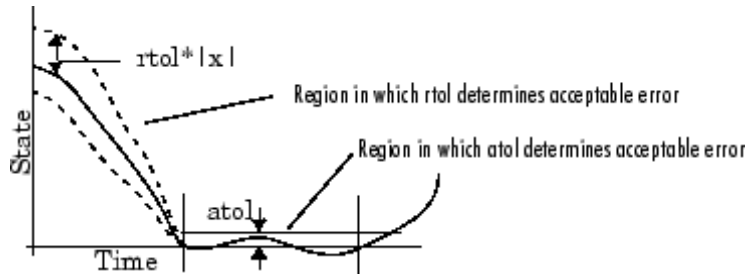
If you specify `auto` (the default), the absolute tolerance for each state is initially set to 1e-6. As the simulation progresses, the absolute tolerance for each state is reset to the maximum value that the state has assumed thus far times the relative tolerance for that state. Thus, if a state goes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

Importing and Exporting Capabilities

During simulation you can import input signal and initial state data from a workspace, and export output signal and state data to a workspace. This capability allows you to use standard or custom MATLAB® functions to generate a simulated system's input signals and to graph, analyze, or otherwise postprocess the system's outputs.

Importing Data from a Workspace

The Simulink® software can input data from a workspace and apply it to the model's top-level input ports during a simulation run. To specify this option:

1 Select the **Input** box in the **Load from workspace** area of the **"Data Import/Export Pane"** pane.

2 Enter an external input specification in the adjacent edit box and click **Apply**.

The Simulink software resolves symbols used in the external input specification as described in "Resolving Symbols" on page 3-69. See the documentation of the `sim` command for some data import capabilities that are available only for programmatic simulation.

Note The use of the **Input** box is independent of the setting of the **Format** list on the **Data Import/Export** pane.

The input data can take any of the following forms:

- Time series — see "Importing Time-Series Data" on page 14-22

- Array — see "Importing Data Arrays" on page 14-24

- Time expression — see "Using a MATLAB® Time Expression to Import Data" on page 14-24

- Structure — see "Importing Data Structures" on page 14-25

The Simulink software linearly interpolates or extrapolates input values as necessary if the **Interpolate data** option is selected for the corresponding Inport.

Importing Time-Series Data

Any root-level Inport block can import data specified by a time-series object (see `Simulink.Timeseries` in the online reference) residing in a workspace. In addition, any root-level input port defined by a bus object (see `Simulink.Bus` in the online documentation) can import data from a time-series array object (see `Simulink.TSArray` in the online reference) that has the same structure as the bus object. Time-series objects are a derivation of standard MATLAB time-series objects and, therefore, can be manipulated using the MATLAB Time Series Tools. See "Using Time Series Tools" in the MATLAB Data Analysis documentation for further details.

Importing time-series objects allows you to import data logged by a previous simulation run (see "Logging Signals"). For example, suppose that you have a model that references several other models. You could use data logged from the inputs of the referenced models when simulating the top model as inputs for the referenced models simulated by themselves. This allows you to test the referenced models independently of the top model and each other.

To import data from time-series objects and time-series array objects, enter a comma-separated list of variables or expressions into the **Input** edit field on the **Data Import/Export** pane of the Configuration Parameters

dialog box (see "Configuration Parameters Dialog Box"). Each variable or expression in the **Input** list should evaluate to the appropriate time-series object or time-series array object that corresponds to one of the model's root-level input ports, with the first item corresponding to the first root-level input port, the second to the second root-level input port, and so on. The model `sldemo_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.

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, In1 that accepts two signals, and In2 that accepts one signal. Also, suppose that the MATLAB workspace defines `t` and `u` as follows:

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

In1 accepts the signals `sin(t)` and `cos(t)`, and In2 accepts the signal `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 a workspace. To use a time expression, enter the expression as a string (i.e., enclosed in single quotes) in the **Input** field of the **Data Import/Export** pane. The time expression can be any MATLAB expression that evaluates to a row vector equal in length to the number of signals entering the model's input ports. For example, suppose that a model has one vector Inport that accepts two signals. Furthermore, suppose that `timefcn` is a user-defined function that returns a row vector two elements long. The following are valid input time expressions for such a model:

```
'[3*sin(t), cos(2*t)]'

'4*timefcn(w*t)+7'
```

The expression is evaluated at each step of the simulation, applying the resulting values to the model's input ports. Note that the Simulink software defines the variable t when it runs the simulation. Also, you can omit the time variable in expressions for functions of one variable. For example, the expression sin is interpreted as sin(t).

Importing Data Structures

The Simulink software can read data from the workspace in the form of a structure whose name is specified in the **Input** text field. You can import structures that include only signal data or both signal and time data. The type of data structure is evaluated based on the structure itself.

Importing Signal-and-Time Data Structures. To import structures that include both signal and time data, the input structure must have two top-level fields: time and signals. The time field contains a column vector of the simulation times. The signals field contains an array of substructures, each of which corresponds to a model input port.

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

Note You must set the **Port dimensions** parameter of the Inport to be the same value as the dimensions field of the corresponding input structure. If the values differ, an error message is displayed when you try to simulate the model.

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

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

To load this data into the model's input port, you would select the **Input** option on the **Data Import/Export** pane and enter `a` in the input expression field.

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

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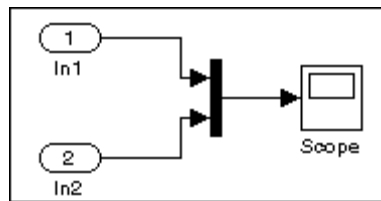
Suppose that you want to input a sine wave into the first port and a cosine wave into the second port. To do this, define a vector, `a`, as follows, in the MATLAB workspace:

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

```
a.signals(2).dimensions = 1;
```

Select the **Input** box for this model, and enter a in the adjacent text field.

Importing Signal-Only Structures. The Structure format is the same as the Structure with time format except that the time field is empty. For example, in the preceding example, you could set the time field as follows:

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

In this case, the input for the first time step is read from the first element of an input port's value array, the value for the second time step from the second element of the value array, etc. If you enter the structure without time, the Inport block must have a discrete sample time.

Per-Port Structures. This format consists of a separate structure-with-time or structure-without-time for each port. Each port's input data structure has only one signals field. To specify this option, enter the names of the structures in the **Input** text field as a comma-separated list, in1, in2,..., inN, where in1 is the data for your model's first port, in2 for the second input port, and so on.

Specifying Time Vectors for Discrete Systems

In some cases, the Simulink software calculates block sample hits at sample times different from those specified by a time vector generated in MATLAB. Typically, these are small floating point inaccuracies that can cause the Simulink product to apparently miss a specified time step in lieu of a different sample point. In order to avoid these numerical inaccuracies, generate the time vector either in MATLAB or in Simulink based on the fundamental sample time of the model.

For example, if the model has a fundamental sample time 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

You can export a model's states and root-level outputs to the MATLAB workspace during simulation of the model. To do this, select the type of data that you want to export on the **Save to workspace** area of the **"Data Import/Export Pane"** pane of the Configuration Parameters dialog box. The field adjacent to each type specifies the name of a MATLAB workspace variable to be used by the Simulink software to store the exported data.

Each field initially specifies a default variable. You can edit the fields to specify names of your own choosing. Select **Signal logging** to enable signal logging for the model. See "Logging Signals" on page 8-49 for more information. See the documentation of the `sim` command for some data export capabilities that are available only for programmatic simulation.

Note The output is saved to the MATLAB workspace at the base sample rate of the model. Use a To Workspace block if you want to save output at a different sample rate.

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

See the documentation of the `sim` command for some capabilities that are available only for programmatic simulation. Format options for model states and outputs are listed below.

Format Options

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

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

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

Note You can use array format to save your model's outputs and states only if the outputs are either all scalars or all vectors (or all matrices for states), are either all real or all complex, and are all of the same data type. Use the `Structure` or `Structure with time` output formats (see "Structure with time" on page 14-29) if your model's outputs and states do not meet these conditions.

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

The structure used to save outputs has two top-level fields:

- `time`

  Contains a vector of the simulation times.

- `signals`

  Contains an array of substructures, each of which corresponds to a model output port.

Each substructure has four fields:

- `values`

  Contains the outputs for the corresponding output port. If the outputs are scalars or vectors, the `values` field is a matrix each of whose rows represents an output at the time specified by the corresponding element of the time vector. If the outputs are matrix (2-D) values, the `values` field is a 3-D array of dimensions `M-by-N-by-T` where `M-by-N` is the dimensions of the output signal and `T` is the number of output samples. If `T = 1`, MATLAB drops the last dimension. Therefore, the `values` field is an `M-by-N` matrix.

- `dimensions`

Specifies the dimensions of the output signal.

- label

  Specifies the label of the signal connected to the output port or the type of state (continuous or discrete).

- blockName

  Specifies the name of the corresponding output port or block with states.

- inReferencedModel

  Contains a value of 1 if the signals field records the final state of a block that resides in the submodel. Otherwise, the value is false (0).

The following is an example of the structure-with-time format for a nonreferenced model.

```
>> xout.signals(1)

ans =

                 values: [296206x1 double]
             dimensions: 1
                  label: 'CSTATE'
              blockName: 'vdp/x1'
      inReferencedModel: 0
```

The structure used to save states has a similar organization. The states structure has two top-level fields:

- time

  The time field contains a vector of the simulation times.

- signals

  The field contains an array of substructures, each of which corresponds to one of the model's states.

Each signals structure has four fields: values, dimensions, label, and blockName. The values field contains time samples of a state of the block specified by the blockName field. The label field for built-in blocks indicates

the type of state: either CSTATE (continuous state) or DSTATE (discrete state). For S-Function blocks, the label contains whatever name is assigned to the state by the S-Function block.

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

Note The Simulink software can read back simulation data saved to the MATLAB workspace in the Structure with time output format. See "Importing Signal-and-Time Data Structures" on page 14-25 for more information.

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

Per-Port Structures. This format consists of a separate structure-with-time or structure-without-time for each output port. Each output data structure has only one signals field. To specify this option, enter the names of the structures in the **Output** text field as a comma-separated list, out1, out2,..., outN, where out1 is the data for your model's first port, out2 for the second input port, and so on.

Importing and Exporting States

You can import the initial values of a system's states, i.e., its initial conditions, at the beginning of a simulation and save the final values of the states at the end of the simulation. This feature allows you to save a steady-state solution and restart the simulation at that known state.

Saving Final States

To save the final values of a model's states, check **Final states** in the **Save to workspace** area of the **Data Import/Export** pane and enter a name in the adjacent edit field. The states are saved in a workspace variable having the specified name. The saved data has the format that you specify in the **Save options** area of the **Data Import/Export** pane.

When saving states from a referenced model in the structure-with-time format, a boolean subfield is added named `inReferencedModel` to the `signals` field of the saved data structure. This field's value is true (1) if the `signals` field records the final state of a block that resides in the submodel, and a 0 otherwise. For example,

```
>> xout.signals(1)

ans =

                  values: [101x1 double]
              dimensions: 1
                   label: 'DSTATE'
               blockName: [1x66 char]
       inReferencedModel: 1
```

If the signals field records a submodel state, its `blockName` subfield contains a compound path comprising a top model path and a submodel path. The top model path is the path from the model root to the Model block that references the submodel. The submodel path is the path from the submodel root to the block whose state the `signals` field records. The compound path uses a | character to separate the top and submodel paths, e.g.,

```
>> xout.signals(1).blockName

ans =

sldemo_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 the Simulink software reorders the states, but the initial state array is not correspondingly reordered.

- Assign a different data type to each state's initial value.

- Initialize only a subset of the states.

For example, the following commands create an initial state structure that can be used to initialize the x2 state of the vdp model. The x1 state is not initialized in the structure and, therefore, the value entered into the state's associated Integrator block is used during the simulation.

```
% Open the vdp demo model
vdp

% Use getInitialState to obtain an initial state structure
states = Simulink.BlockDiagram.getInitialState('vdp');

% Set the initial value of the signals structure element
% associated with x2 to 2.
states.signals(2).values = 2;

% Remove the signals structure element associated with x1
states.signals(1) = [];
```

To use this states variable, open the **Configuration Parameters** dialog box for the vdp model. Check **Initial state** in the **Load from workspace** area

of the **Data Import/Export** pane and type `states` into the associated edit field. When you run the model, note that both states have the initial value of 2. The initial value of the `x2` state is assigned in the `states` structure, while the initial value of the `x1` state is assigned in its Integrator block.

Note You must use the structure or structure-with-time format to initialize the states of a top model and the models that it references.

Limiting Output

Saving data to a workspace can slow down the simulation and consume memory. To avoid this, you can limit the number of samples saved to the most recent samples or you can skip samples by applying a decimation factor. To set a limit on the number of data samples saved, select the check box labeled **Limit data points to last** and specify the number of samples to save. To apply a decimation factor, enter a value in the field to the right of the **Decimation** label. For example, a value of 2 saves every other point generated.

Specifying Output Options

The **Output options** list on the **Data Import/Export** configuration pane ("Data Import/Export Pane") enables you to control how much output the simulation generates. You can choose from three options:

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

Refining Output

The `Refine output` choice provides additional output points when the simulation output is too coarse. This parameter provides an integer number of output points between time steps; for example, a refine factor of 2 provides output midway between the time steps as well as at the steps. The default refine factor is 1.

To get smoother output, it is much faster to change the refine factor instead of reducing the step size. When the refine factor is changed, the solvers generate additional points by evaluating a continuous extension formula at those points. This option changes the simulation step size so that time steps coincide with the times that you have specified for additional output.

The refine factor applies to variable-step solvers and is most useful when you are using `ode45`. The `ode45` solver is capable of taking large steps; when graphing simulation output, you might find that output from this solver is not sufficiently smooth. If this is the case, run the simulation again with a larger refine factor. A value of 4 should provide much smoother results.

Note This option helps the solver locate zero crossings (see "Zero-Crossing Detection" on page 2-20). 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, an **Output times** field is displayed on the **Data Import/Export** pane. Enter a MATLAB expression in this field that evaluates to an additional time or a vector of additional times. This option causes the solver to produce hit times at the output times you have specified, in addition to the times it needs to accurately simulate the model.

Note This option helps the solver locate zero crossings (see "Zero-Crossing Detection" on page 2-20). 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 2-20). 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

About Configuration Sets

A *configuration set* is a named set of values for a model's parameters, such as solver type and simulation start or stop time. Every new model is created with a default configuration set, called Configuration, that initially specifies default values for the model's parameters. You can subsequently create and modify additional configuration sets and associate them with the model. The sets associated with a model can specify different values for any or all configuration parameters.

This section describes techniques for defining and using configuration sets that are stored in individual models. Such configuration sets are available only to the model that contains them. The next section, "Referencing Configuration Sets" on page 14-47, 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. You can change the active or inactive set at any time (except when executing the model). In this way, you can quickly reconfigure a model for different purposes, e.g., testing and production, or apply standard configuration settings to new models.

Displaying Configuration Sets

To display the configuration sets associated with a model, open the Model Explorer (see "The Model Explorer" on page 13-2). The configuration sets associated with the model appear as gear-shaped nodes in the Model Explorer's **Model Hierarchy** pane.
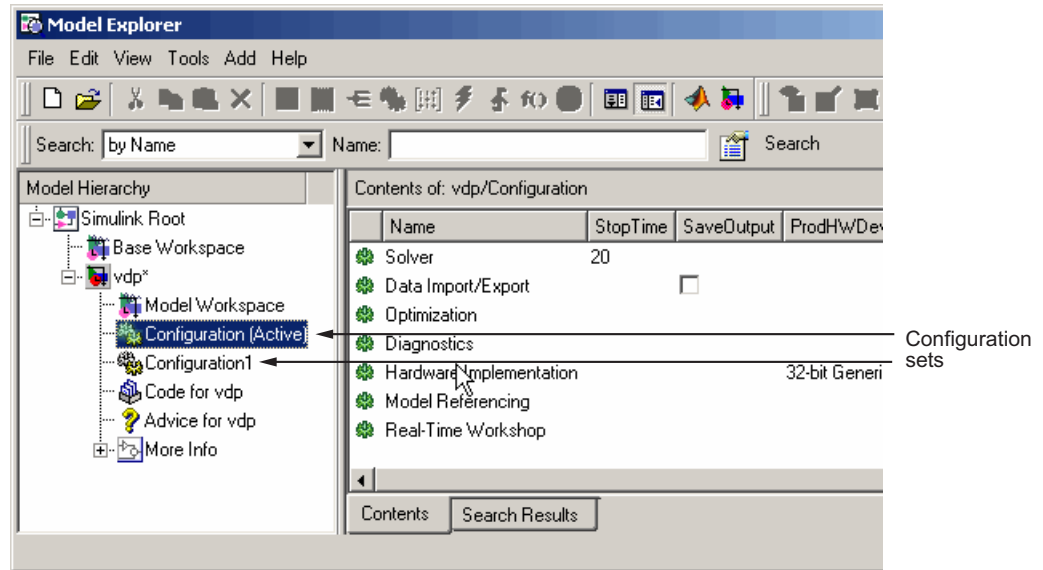
The Model Explorer's **Contents** pane displays the components of the selected configuration set. The Model Explorer's Dialog pane displays a dialog for setting the parameters of the selected group (see "Configuration Parameters Dialog Box").

Activating a Configuration Set

To activate a configuration set, right-click the configuration set's node to display the node's context menu, then select **Activate** from the context menu.

Opening Configuration Sets

In 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.
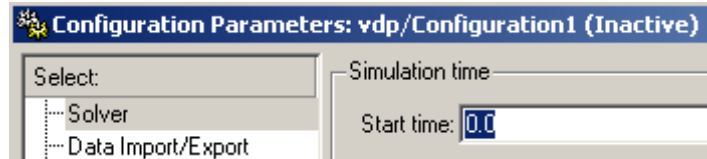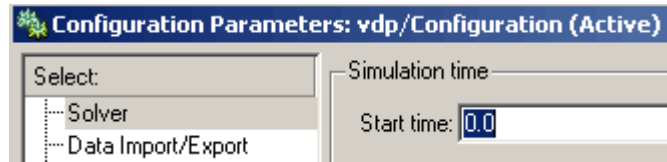
Note Every configuration set has its own configuration parameter dialog. As you change the state of a configuration set, the top-left corner label changes to reflect the state.

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.

4 Select the model in which you want to create the copy.

Note You can create a copy in the same model as the original.

---

5 Select **Paste** from the Model Explorer's **Edit** menu or from the model's context menu.

To copy the configuration set, using object drag-and-drop, hold the right mouse button down and drag the configuration set's node to the node of the model in which you want to create the copy. To move a configuration set from one model to another, using drag-and-drop, hold the left mouse button down and drag the configuration set's node to the node of the destination model.

Note You cannot move or delete a model's active configuration set.

Copying Configuration Set Components

To copy a configuration set component from one configuration set to another:

1 Select the component in the Model Explorer's **Contents** pane.

2 Select **Copy** from the Model Explorer's **Edit** menu or the component's context menu.

3 Select the configuration set into which you want to copy the component.

4 Select **Paste** from the Model Explorer's **Edit** menu or the component's context menu.

Note The copy replaces the component of the same name in the destination configuration set. For example, if you copy the Solver component of configuration set A and paste it into configuration set B, the copy replaces B's existing Solver component.

Creating Configuration Sets

To create a new configuration set, select **Configuration Set** from the Model Explorer's **Add** menu or press the **Add Configuration** button in the Model Explorer's toolbar. You can also create a new configuration set by copying an existing configuration set.

Setting Values in Configuration Sets

To set the value of a parameter in a configuration set, select the configuration set in the Model Explorer and then edit the value of the parameter on the corresponding dialog in the Model Explorer's dialog view.

Configuration Set API

An application program interface (API) is provided 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.
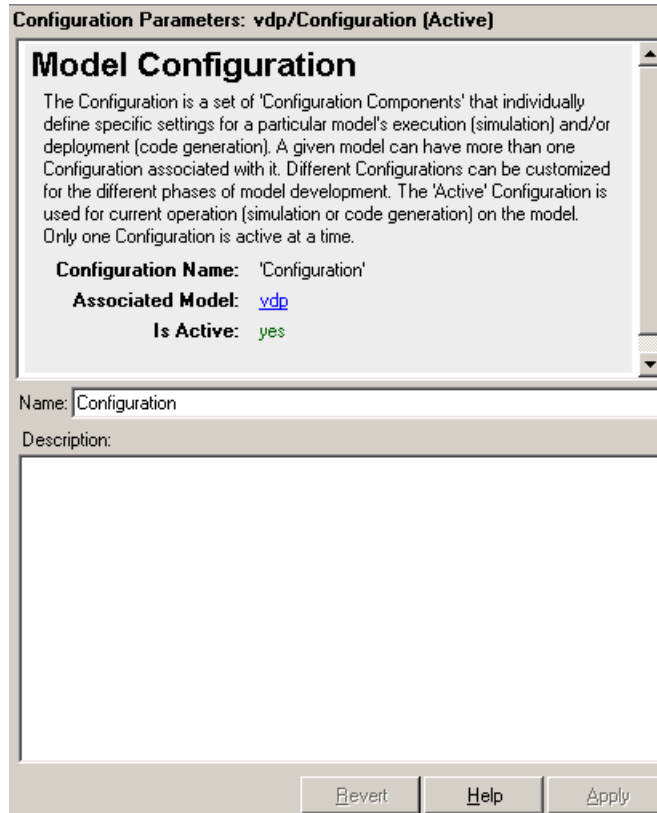
You can edit the name and description of your configuration. See "Model Configuration Pane".

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.

You can also edit the configuration defaults in the Simulink Preferences window. See "Model Configuration Pane".

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 used in different contexts without changing the model file**

  A submodel that uses different configuration sets in different contexts contains a configuration reference that specifies the submodel's configuration set as a variable. When an instance of the submodel is called, the Simulink® software assigns that variable a freestanding configuration set for the current context.

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.
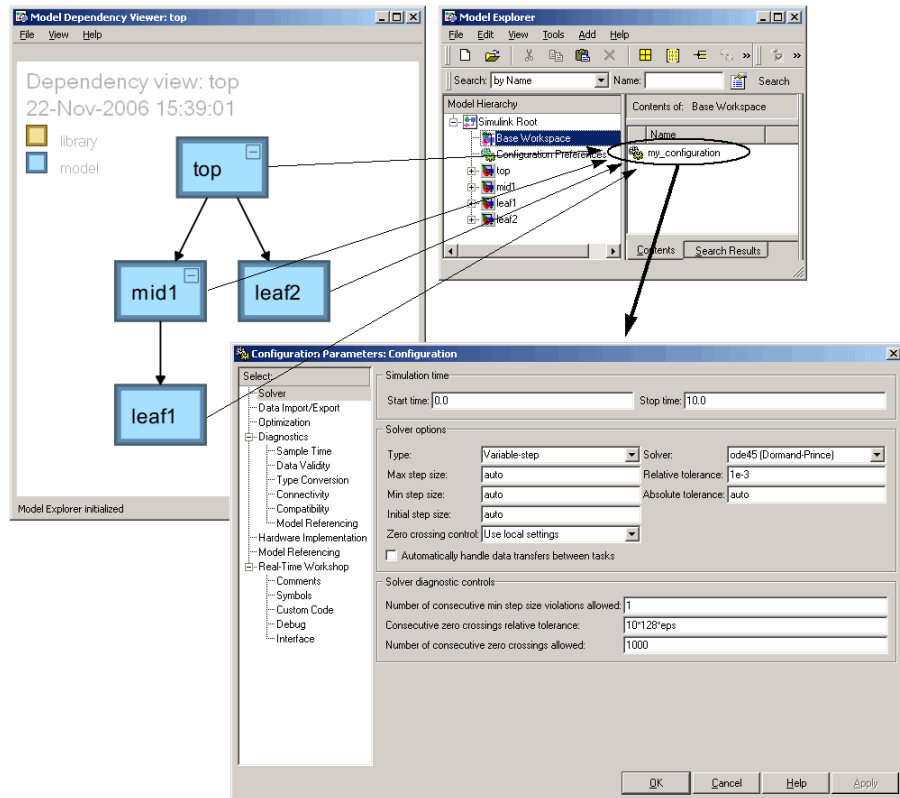
To use a configuration reference to link a freestanding configuration set to a model, you:

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.

14-49

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 14-60 for details. General information about configuration sets appears in "Configuration Sets" on page 14-37.

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 using any of the techniques described in "Creating Configuration Sets" on page 14-41 or as follows:

```
cset = Simulink.ConfigSet
```

where *cset* is a new or existing base workspace variable. The new configuration set initially has default parameter values, copied from the default configuration set.

Copying a Configuration Set Stored in a Model

You can copy an existing configuration set to the base workspace using drag and drop operations described in "Copying, Deleting, and Moving Configuration Sets" on page 14-40, 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 14-50, 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.
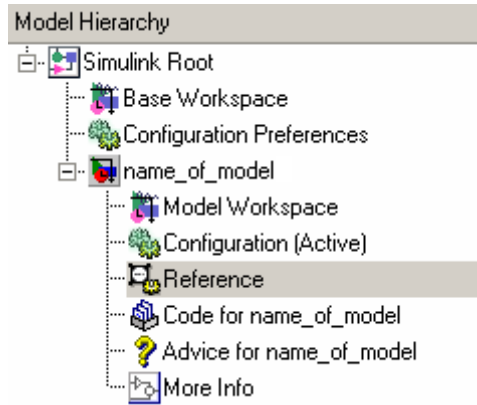
3 Select the new configuration reference in the Model Hierarchy pane, or right-click the configuration reference and choose **Open** from the context menu.

The Configuration Reference dialog appears in the Dialog pane or a separate window.
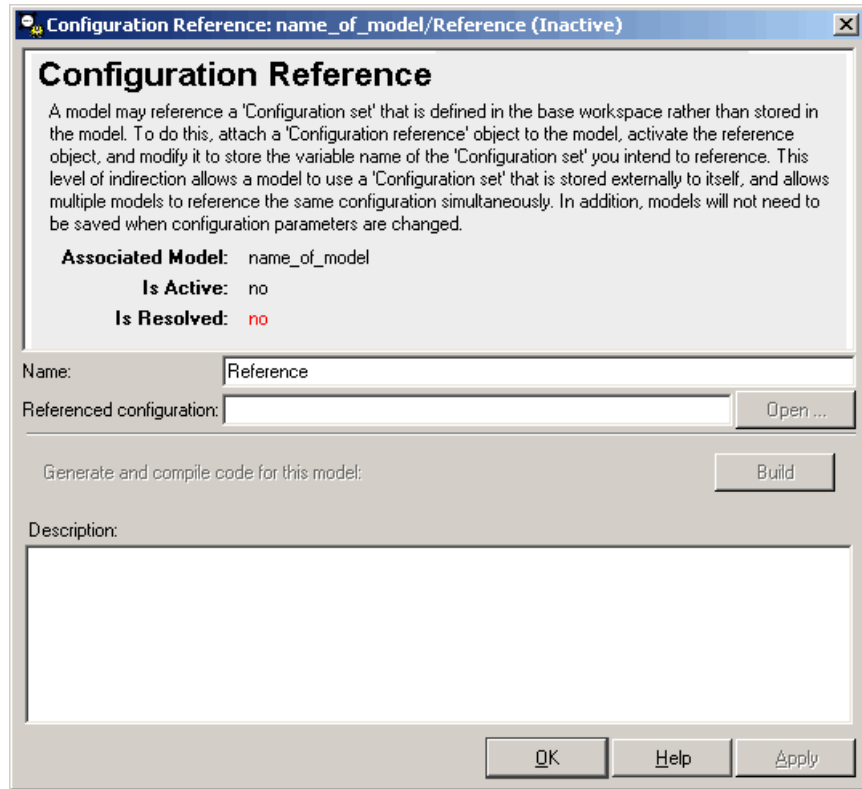
4 Change the default **Name** if desired. This name exists for human readability, and does not affect the configuration reference functionally.

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.
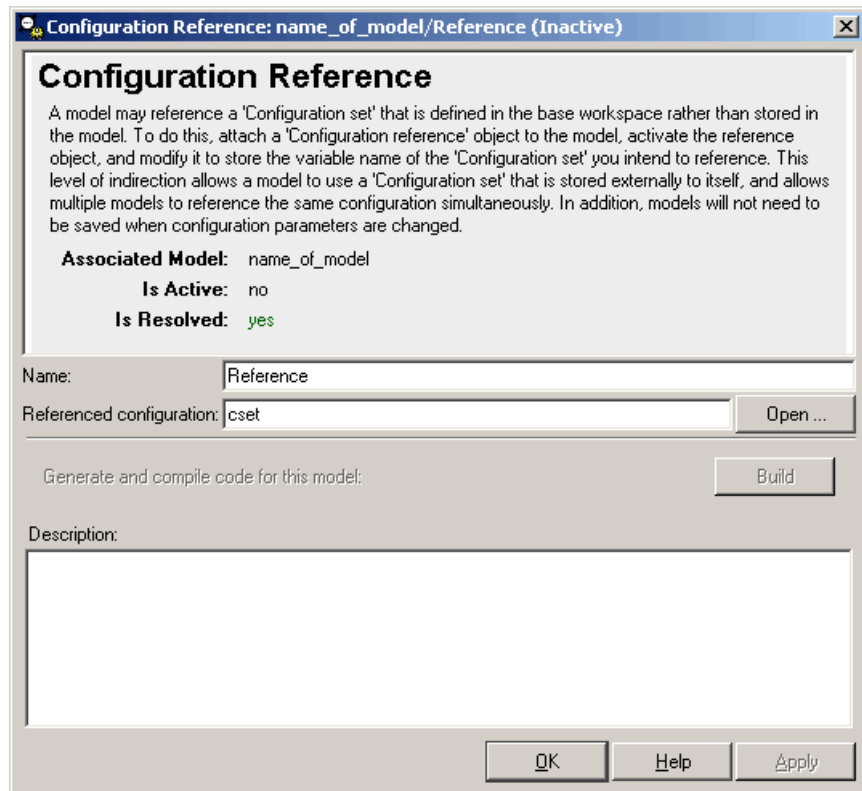
If you do not specify a valid **Referenced configuration**, a warning is posted. Any attempt to use a configuration reference that lacks a valid **Referenced configuration** generates an error. The API equivalent of **Referenced configuration** is WSVarName. You can later use the GUI or API to correct the specification or provide a configuration set with the correct name. See "Unresolved Configuration References" on page 14-59 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 14-59 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 14-40 and "Configuration Set API" on page 14-42.

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')
% Get handle to local cset
cset = getConfigSet('mdl1', 'Configuration')
% Create freestanding copy; original remains in model
cset1 = copy(cset)
% In the original model, create a configuration
% reference to the cset copy
cref1 = Simulink.ConfigSetRef
cref1.WSVarName = 'cset1'
% Rename if name conflict occurs
attachConfigSet('mdl1', cref1, true)

% In a second model, create a configuration
% reference to the same cset
open_system('mdl2')
% Rename if name conflict occurs
attachConfigSetCopy('mdl2', cref1, true)
% Delete original cset from first model
detachConfigSet('mdl1', 'Configuration')
```

Changing a Configuration Reference

You can change an existing configuration reference 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).
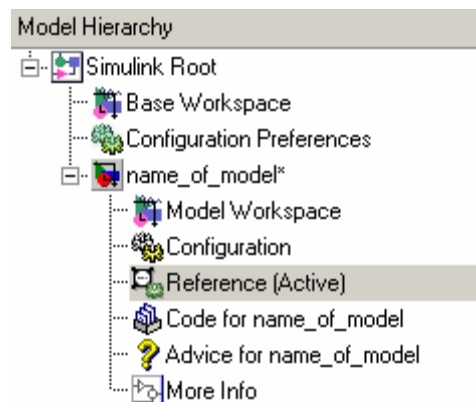
The freestanding configuration set to which the active reference points now provides the configuration parameters for the model containing the reference.

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 14-52 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. The Simulink software retrieves the referenced configuration set and performs the indicated get_param on it.

For example, if configuration reference *cref* links to configuration set *cset*, the following operations give identical results:

```
get_param (cset, 'StopTime')
get_param (cref, 'StopTime')
```

Changing Values in a Referenced Configuration Set

You cannot change a configuration set in any way, 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.
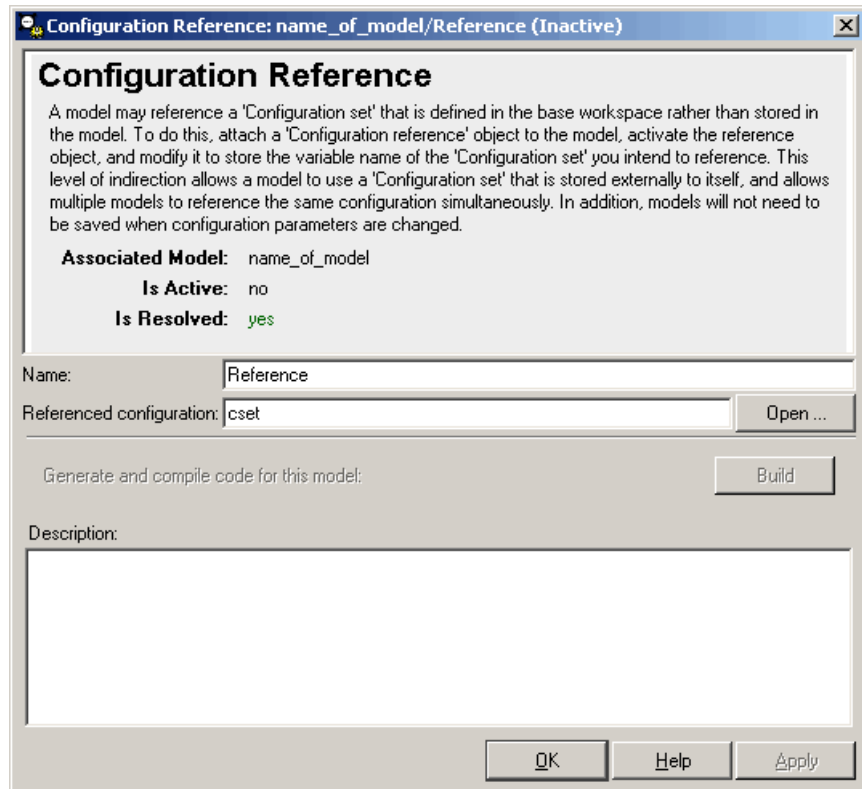
2 Click **Open** to the right of the **Referenced configuration** field.

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 14-56.

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.

```
% Create a new configuration set
cset1=Simulink.ConfigSet;
% Set a non-default stop time
set_param (cset1, 'StopTime', 500)
% Create a new config reference
cref1=Simulink.ConfigSetRef;
% Resolve the config ref to the set
cref1.WsVarName='cset1';
% Attach the config ref to a model
attachConfigSet('mdl1', cref1, true)
```

```
% Replace config set on base workspace
cset1=Simulink.ConfigSet;
% Call to refresh is commented out!
% cset1.refresh;
% Set a different stop time
set_param (cset, 'StopTime', 75)
```

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 14-62.

- 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.

Diagnosing Simulation Errors

In this section...
"Response to Run-Time Errors" on page 14-65
"Simulation Diagnostics Viewer" on page 14-65
"Creating Custom Simulation Error Messages" on page 14-67

Response to Run-Time Errors

If errors occur during a simulation, the Simulink® software halts the simulation, opens the subsystems that caused the error (if necessary), and displays the errors in the Simulation Diagnostics Viewer. The following sections explain how to use the viewer to determine the cause of the errors, and how to create custom error messages.

Simulation Diagnostics Viewer

The viewer comprises an Error Summary pane and an Error Message pane.
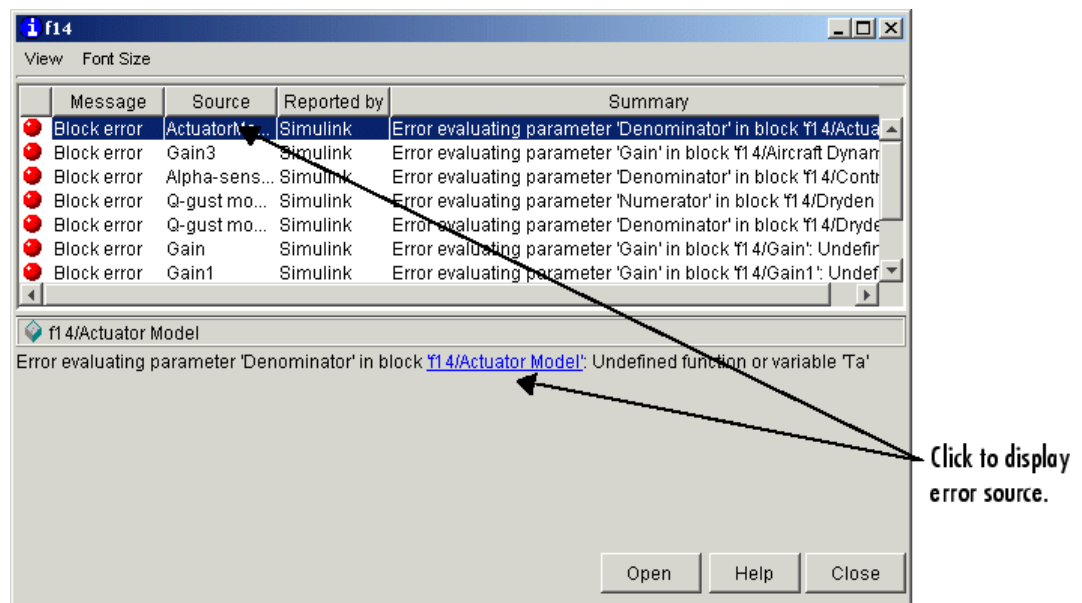
Error Summary Pane

The upper pane lists the errors that caused the simulation to terminate. The pane displays the following information for each error.

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

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

Reported by. Component that reported the error (for example, the Simulink product, the Stateflow® product , the Real-Time Workshop® product, 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, the Simulink software opens (if necessary) the subsystem that contains the first error source and highlights the source.
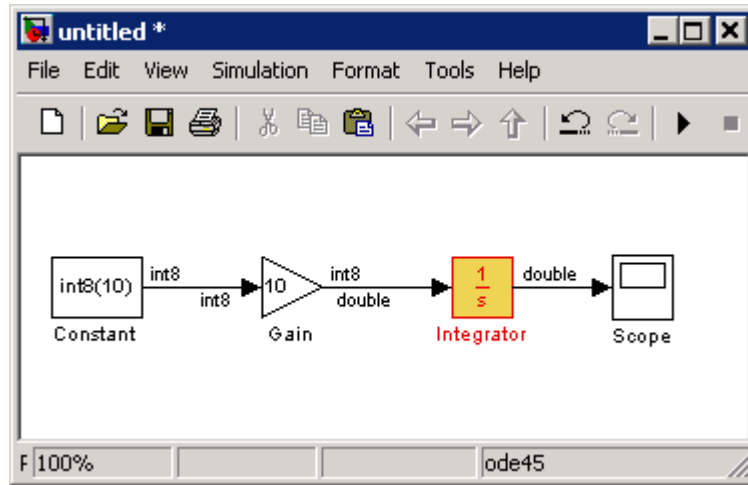
You can display the sources of other errors by clicking anywhere in the error message in the upper pane, by clicking the name of the error source in the error message (highlighted in blue), or by clicking the **Open** button on the viewer.

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.
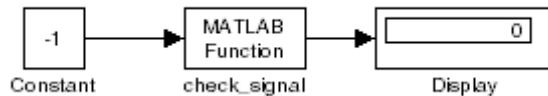
The MATLAB Fcn block invokes the following function:

```
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.
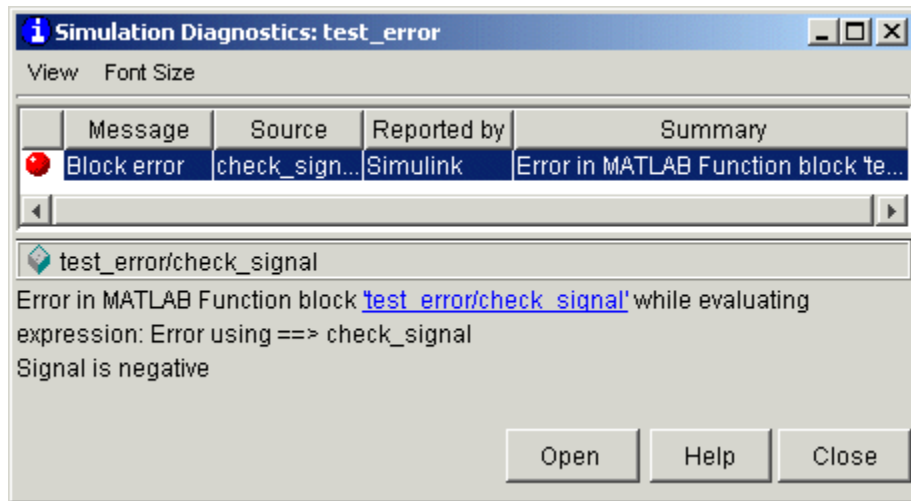
Including Hyperlinks

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

In this section...
"About Improving Performance and Accuracy" on page 14-71
"Speeding Up the Simulation" on page 14-71
"Improving Simulation Accuracy" on page 14-72

About Improving Performance and Accuracy

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

The solvers handle most model simulations accurately and efficiently with their default parameter values. However, some models yield better results if you adjust solver parameters. Also, if you know information about your model's behavior, your simulation results can be improved if you provide this information to the solver.

Speeding Up the Simulation

Slow simulation speed can have many causes. Here are a few:

- Your model includes a MATLAB® Fcn block. When a model includes a MATLAB Fcn block, the MATLAB interpreter is called at each time step, drastically slowing down the simulation. Use the built-in Fcn block or Math Function block whenever possible.

- Your model includes an M-file S-function. M-file S-functions also cause the MATLAB interpreter to be called at each time step. Consider either converting the S-function to a subsystem or to a C-MEX file S-function.

- Your model includes a Memory block. Using a Memory block causes the variable-order solvers (ode15s and ode113) to be reset back to order 1 at each time step.

- The maximum step size is too small. If you changed the maximum step size, try running the simulation again with the default value (auto).

- Did you ask for too much accuracy? The default relative tolerance (0.1% accuracy) is usually sufficient. For models with states that go to zero, if

the absolute tolerance parameter is too small, the simulation can take too many steps around the near-zero state values. See the discussion of error in "Maximum order".

- The time scale might be too long. Reduce the time interval.

- The problem might be stiff, but you are using a nonstiff solver. Try using `ode15s`.

- The model uses sample times that are not multiples of each other. Mixing sample times that are not multiples of each other causes the solver to take small enough steps to ensure sample time hits for all sample times.

- The model contains an algebraic loop. The solutions to algebraic loops are iteratively computed at every time step. Therefore, they severely degrade performance. For more information, see "Algebraic Loops" on page 2-31.

- Your model feeds a Random Number block into an Integrator block. For continuous systems, use the Band-Limited White Noise block in the Sources library.

- Your model contains a scope viewer that displays a large number of data points. Try adjusting the viewer parameter settings that can affect performance. For more information, see "How Scope Viewer Parameter Settings Can Affect Performance" on page 15-9.

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

In this section...
"About Simulating Programmatically" on page 14-74
"Using the sim Command" on page 14-74
"Using the set_param Command" on page 14-75

About Simulating 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

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 sim command reference page. 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. The configuration parameters are discussed in "Configuration Sets" on page 14-37.

Programmatic Symbol Resolution

When you use the sim command to run a simulation programmatically, you have two options that do not exist with interactive simulation: you can specify a workspace other than the MATLAB base workspace as the last workspace searched in hierarchical symbol resolution, and a workspace other than the MATLAB base workspace as the destination for any data logged or exported during simulation.

Most simulation is interactive, so most Simulink® documentation does not mention these possibilities. For information about substituting some other workspace for the base workspace during programmatic simulation, see the `sim` command reference page.

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')
```

The Simulink software returns `'stopped'`, `'initializing'`, `'running'`, `'paused'`, `'updating'`, `'terminating'`, and `'external'` (used with the Real-Time Workshop® product).

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.

15

Visualizing Simulation Results

About Scope Blocks, Viewers, Signal Logging, and Test Points

In this section...
"What are Scope Blocks, Signal Viewers, Test Points and Data Logging?" on page 15-2
"How Scope Blocks and Signal Viewers Differ" on page 15-3
"Why Use Generators and Signal Viewers Instead of Source and Scope Blocks?" on page 15-4

What are Scope Blocks, Signal Viewers, Test Points and Data Logging?

Scope blocks, signal viewers, test points, and data logging provide ways for you to display and capture results from your simulations.

These icons represent the various data display and data capture devices:
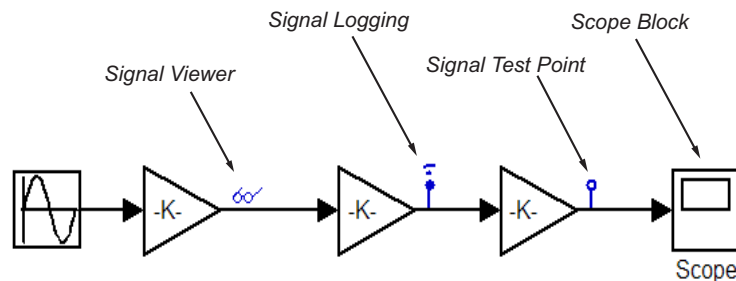
- To learn how to quickly perform basic signal viewer tasks, see "Performing Common Viewer Tasks" on page 15-17.

- For detailed information on signal viewers, see "Introducing the Signal and Scope Manager" on page 8-33.

- To learn how to add and change signal viewers, see "Using the Signal and Scope Manager" on page 8-39.

- For more information on signal logging, see "Logging Signals" on page 8-49.

- For more information on Signal Test Points, see "Working with Test Points" on page 8-70.

- For more information on Scope Blocks, see "Sinks".

How Scope Blocks and Signal Viewers Differ

You use Scope Blocks and signal viewers to display simulation results, but as shown in this table, their characteristics differ:

Characteristic	Signal Viewer	Scope Block
Interface	Attach to signal using Signal Selector or context menu See "The Signal Selector" on page 8-44 See "Scope Viewer Context Menu" on page 15-16	Drag from Library Browser
Scope of Control	All viewers centrally managed from Signal and Scope Manager See "Introducing the Signal and Scope Manager" on page 8-33	Each managed individually
Signals per axis	Multiple	One nonbus signal per axis
Axes per scope	Multiple "Displaying Multiple Axes" on page 15-19	One
Data handling	Save data to a signal logging object	Save variable data to workspace as structures or arrays
Data logging	Log data to model-wide data object See Simulink.ModelDataLogs	None

Characteristic	Signal Viewer	Scope Block
Scrolling display data capability	Yes	No
Display	• Data markers • Legends • Color and line codes distinguish signals	Color and line codes distinguish signals
Graph Refresh Period	Adjustable	Fixed

Why Use Generators and Signal Viewers Instead of Source and Scope Blocks?

You should use signal generators and viewers instead of Source and Scope blocks when:

- You want to navigate to and attach generators or viewers deep within a model hierarchy.

- You want to centrally manage all generators and viewers present in your model.

- You want to use the display features provided by signal viewers that are not available in Scope blocks.

- You want to reduce clutter in your block diagram. Because signal viewers attach directly to signals, it is not necessary to route them to a Scope block. This results in fewer signal routes in your block diagram.

- You want to easily view data from referenced models. See "Working with Test Points" on page 8-70 for more information.

Methods for Attaching a Generator or Viewer

Generators and viewers attach to signals in your model in one of two ways:

When...	Use...
You want to review all of the scopes and viewers, and the signals connected to them	Signal and Scope Manager
You want to quickly connect and disconnect a viewer or generator	Signal Context Menu

Displaying a Scope Viewer

Click the Scope Viewer icon to display a particular Scope Viewer:
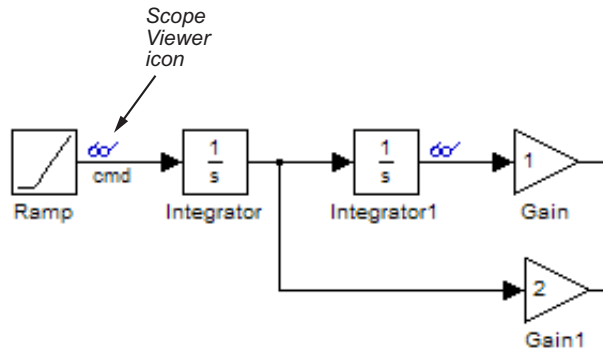
Two plots are displayed in this viewer example. Each is identified with a unique color, and the graph has a legend.
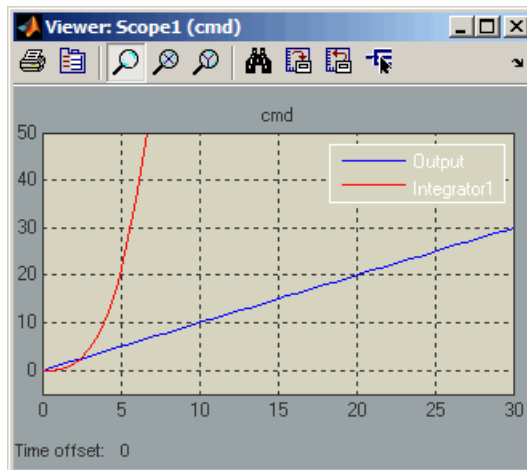
- To learn how to add a legend and to zoom into regions in your graph, see "Performing Common Viewer Tasks" on page 15-17.

- To learn how a viewer can display multiple signals, see "Adding Multiple Signals to a Scope Viewer" on page 15-17.

- To learn about the Scope Viewer controls, see "Changing Viewer Characteristics" on page 15-11.

- To learn how to attach a Scope Viewer to a signal, see "Attaching a Scope Viewer" on page 15-17.

Tip You must first attach a viewer for the Scope Viewer icon to be visible.

Things to Know When Using Viewers

In this section...
"About Viewers" on page 15-8
"How the Viewer Determines Trace Color Coding and Line Styles" on page 15-8
"How Scope Viewer Parameter Settings Can Affect Performance" on page 15-9

About Viewers

- The Scope Viewer is not the same as the Scope block. For an explanation of the differences, see "How Scope Blocks and Signal Viewers Differ" on page 15-3.

- The Scope Viewer does not show the signal label on the axis.

- The Scope Viewer does not work with the Report Generator.

- The Scope Viewer does not display the simulation minor time step values.

- Not all of the Scope Viewer features are supported when you simulate your model in Rapid Accelerator mode. For more information, see "Using Scopes and Viewers with Rapid Accelerator Mode" on page 19-16.

- The **Help** button for the Scope Viewer is located in the Scope Viewer's Parameters dialog box. For more information, see "Scope Viewer Parameters Dialog Box" on page 15-12.

How the Viewer Determines Trace Color Coding and Line Styles

The Scope Viewer displays each signal as a separate, color-coded trace, in the following order:

1 Blue

2 Red

3 Magenta
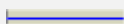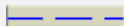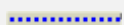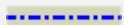
4 Cyan

5 Yellow

6 Green

The viewer cycles through the colors if the axis is displaying more than six signals.

If a signal contains multiple elements (such as a vector or matrix), the viewer distinguishes the elements with different line styles. If a signal has more than four elements, the viewer cycles through the line styles. The line styles retain the color of the signal.

Signal Element	Scope Viewer
1	
2	
3	
4	

How Scope Viewer Parameter Settings Can Affect Performance

In some cases, when a Scope Viewer needs to display a large number of data points, the simulation slows. When this happens, you can improve simulation performance by adjusting the settings of some of the viewer parameters. Try one or a combination of the following until you are satisfied with the simulation performance.

- Turn off scroll mode. See "**Scroll**" on page 15-13.

- Reduce the time range. See "**Time Range**" on page 15-13.

- Use decimation to reduce the number of data points. See "**Decimation**" on page 15-14.

- Increase the refresh period to decrease the refresh rate. See "**Refresh Period**" on page 15-15.

- Limit the number of data points that the viewer saves to the workspace. See "**Limit data points to last**" on page 15-14.

Changing Viewer Characteristics

In this section...
"The Scope Viewer Toolbar" on page 15-11
"Scope Viewer Parameters Dialog Box" on page 15-12

The Scope Viewer Toolbar

The Scope Viewer toolbar is attached to each Scope Viewer. It has the following controls:

Icon	Function
	Opens the Print dialog box so you can print the contents of a Scope Viewer window.
	Opens the Scope Parameters dialog for modifying display characteristics. For details, see "Scope Viewer Parameters Dialog Box" on page 15-12.
	Simultaneously zooms in on the *x* and *y* axes. The zoom feature is not active while the simulation is running. For more information, see "Zooming In On Graph Regions" on page 15-18.
	Use this button to zoom in on the *x* axis only. The zoom feature is not active while the simulation is running. For more information, see "Zooming In On Graph Regions" on page 15-18.
	Use this button to zoom in on the *y* axis only. The zoom feature is not active while the simulation is running. For more information, see "Zooming In On Graph Regions" on page 15-18.
	Automatically scales the axis to fully display all signals.

Icon	Function
	Stores the current axis settings so you can apply them to the next simulation.
	Restores the graph setting values saved by the most recent **Save axes settings** command.
	Activates the Signal Selector. For more information, see "The Signal Selector" on page 8-44.
	Docks and undocks the Scope Viewer. When you dock the Scope Viewer, it is placed within the MATLAB® Command Window and automatically resized.

Scope Viewer Parameters Dialog Box

Open the Scope Parameters dialog box by clicking [icon] on the scope toolbar, or by selecting **Scope parameters** from the context menu. There are three tabs:

- **General**, where you set the axis characteristics and the sampling decimation value (see "General Tab" on page 15-12).

- **History**, where you control the amount of stored and displayed data (see "History Tab" on page 15-14).

- **Performance**, where you control the scope refresh rate (see "Performance Tab" on page 15-15).

General Tab
With this tab you control the number of axes, the time range, and the appearance of your graph.

Number of axes. Set the number of axes in this data field. Each axis is displayed as a separate graph within a single Scope Viewer.

An example of this is shown in "Adding Multiple Signals to a Scope Viewer" on page 15-17.

Time Range. Change the *x*-axis limits by entering a number or `auto` in the **Time range** field.

Entering a number of seconds causes each screen to display the amount of data that corresponds to that number of seconds. Enter `auto` to set the *x*-axis to the duration of the simulation.

Note Do not enter variable names in these fields.

Tick labels. Specifies whether to label axes ticks. The options are:

Option	Effect
all	Places ticks on the outside of all axes
inside	Places tick labels inside all axes (available only on signal viewers)
bottom axis only	Places tick labels outside the bottom axes

Scroll. When you select this option, the scope continuously scrolls the displayed signals to the left to keep as much data in view as will fit on the screen at any one time.

In contrast, when this option is not selected, the scope draws a screen full of data from left to right until the screen is full, erases the screen, and draws the next screen full of data. This loop is repeated until the end of simulation time. The effects of this option are discernible only when drawing is slow, for example, when the model is very large or has a very small step size.

Note In some cases, the simulation slows when the simulation runs with the scroll option selected. See "How Scope Viewer Parameter Settings Can Affect Performance" on page 15-9.

Data markers. Displays a marker at each data point on the Scope Viewer screen.

Legends. Displays a legend on the scope that indicates the line style used to display each signal.

Decimation. Logs every Nth data point, where N is the number entered in the edit field.

For example, suppose that your model uses a fixed-step solver with a step size of 0.1 s. if you enter a value of 2, data points for this viewer will be recorded at times 0.0, 0.2, 0.4....

History Tab

With this tab you control the amount of data that the Scope Viewer stores, displays and stores to the workspace. The values that appear in these fields are the values that are used in the next simulation.

Limit data points to last. Limits the number of data points saved to the workspace. Select the **Limit data points to last** check box and enter a value in its data field.

The Scope relies on its data history for zooming and autoscaling operations. If the number of data points is limited to 1,000 and the simulation generates 2,000 data points, only the last 1,000 are available for regenerating the display.

Save to model signal logging object. At the end of the simulation, this option saves the data displayed on the Scope Viewer . The data is saved in the `Simulink.ModelDataLogs` object used to log data for the model (see "Logging Signals" on page 8-49 for more information).

For this option to take effect, you must also enable signal logging for the model as a whole. To do this, check the **Signal logging** option on the **Data Import/Export** pane of the model's **Configuration Parameters** dialog box.

Logging Name. Specifies the name under which to store the viewer's data in the model's `Simulink.ModelDataLogs` object. The name must be different from the log names specified by other signal viewers or for other signals, subsystems, or model references logged in the model's `Simulink.ModelDataLogs` object.

Performance Tab

Controls how frequently the Scope Viewer is refreshed. Reducing the refresh rate can speed up the simulation in some cases.

Note For information about additional Scope Viewer parameters that can affect performance, see "How Scope Viewer Parameter Settings Can Affect Performance" on page 15-9.

This tab contains the following controls.

Refresh Period. Select the units in which the refresh period is expressed. Options are either seconds or frames, where a frame is the width of the scope's screen in seconds. This is the value of the scope's **Time range** parameter.

Refresh Slider. Sets the refresh rate.

Drag the slider button to the right to increase the refresh period and hence decrease the refresh rate.

Freeze Button. Controls refresh.

Click the button to freeze (stop refreshing) or unfreeze the Scope Viewer.

Scope Viewer Context Menu

The Scope Viewer context menu is a convenient way to make simple changes to a Scope Viewer without navigating to the Scope Parameters dialog box.

Right-click within a Scope Viewer to display the context menu. It contains the following controls:

Control	Function
Legends	Adds a legend to your viewer.
Autoscale	Autoscales the viewer axis.
Signal selection	Displays the Signal Selector dialog. For information, see "The Signal Selector" on page 8-44.
Axes properties	Displays the Axis Properties dialog. You can manually set the minimum and maximum range for the *y* axis here.
Scope parameters	Displays the Scope parameters dialog. For information, see "Scope Viewer Parameters Dialog Box" on page 15-12.
Tick labels	Displays the Tick Labels dialog. From here you can turn on and off various tick options.

Performing Common Viewer Tasks

In this section...
"Viewing Scope Viewer Help" on page 15-17
"Attaching a Scope Viewer" on page 15-17
"Adding Multiple Signals to a Scope Viewer" on page 15-17
"Adding a Legend" on page 15-18
"Zooming In On Graph Regions" on page 15-18
"Displaying Multiple Axes" on page 15-19

Viewing Scope Viewer Help

The Scope Viewer **Help** button is in the Scope Viewer's Parameters dialog.

Access the dialog by clicking on the Scope Viewer toolbar.

Attaching a Scope Viewer

1 In your block diagram, right-click a signal, and from the context menu, select **Create & Connect Viewer**.

2 From the list that appears, select the type of scope to be attached.

An empty Scope Viewer will be displayed. You must run the simulation after the viewer has been attached for the information to be plotted.

Tip You can also attach signal viewers with the Signal and Scope Manager. To learn how to do this, see "Using the Signal and Scope Manager" on page 8-39

Adding Multiple Signals to a Scope Viewer

To add additional traces to an existing Scope Viewer:

1 In your block diagram, right-click the signal to be added.

15-17

2 From the signal context menu, select **Connect to existing viewer**.

3 A list of Signal Viewers that have already been created is displayed. From the list, select the scope to which the new signal will be added.

You can also add signals to Signal Viewers with the Signal and Scope Manager. To learn how to do this, see "Using the Signal and Scope Manager" on page 8-39.

Note The simulation must be run again for the new trace to be displayed in the viewer.

Adding a Legend

To add a legend (a box identifying each signal in a graph):

1 Right-click in the viewer to display the context menu.

2 Click on **Legends**.

Tip Left-click and drag the legend box to reposition it.

The color and line styles used in the display are described in "How the Viewer Determines Trace Color Coding and Line Styles" on page 15-8.

Zooming In On Graph Regions

- To zoom in on a region (simultaneously zooming in on the x and y

  directions), select ⌕ from the Scope Viewer toolbar, and left-click within the graph. While holding down the mouse button, define the region of interest.

  To zoom, click and hold the left mouse button, and drag the mouse to define the zoom region bounding box. Release the mouse button to effect the change.

- Select ![icon] and use the mouse to zoom only in the x direction.

  Define the zoom region by positioning the pointer at one end of the region, pressing and holding down the left mouse button, then moving the pointer to the other end of the region. Release the mouse button to effect the change. To zoom, click and hold the left mouse button, and drag the mouse to define the zoom region bounding box. Release the mouse button to effect the change.

- Select ![icon] and use the mouse to zoom only the y direction.

Tip Use Autoscale (![icon]) to restore the display if you mistakenly zoom in too much.

Displaying Multiple Axes

You can add multiple plots (called *axes*) to a Scope Viewer. Each axes can have different *y*-axis settings. This Scope Viewer has three axes, each displaying a separate signal with their own *y* axis settings.
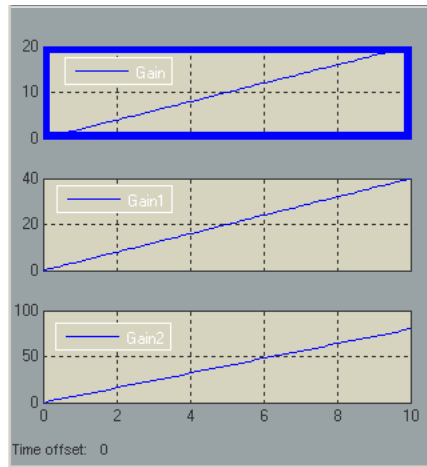
To add axes to an existing Scope Viewer:

1 Open the Scope Viewer Signal Properties dialog.

Access this dialog from the Scope Viewer toolbar or by right-clicking within a viewer.

For information on the Scope Viewer toolbar, see "The Scope Viewer Toolbar" on page 15-11.

2 In the **Number of axes** field, enter the total number of axes for the graph.

3 Click OK to accept the change and dismiss the dialog.

4 In your block diagram, right-click on the signal to be added.

The signal context menu appears.

5 Select **Connect to existing viewer**.

A list of signal viewers is displayed.

6 From the list, select the scope to which the new signal will be added.

7 From the list of Axes, select the pane to which the signal will be plotted. The panes are numbered from top to bottom.

Repeat this step until signals for each of the axes have been assigned.

Note Run the simulation again to display the new traces.

Performing Common Generator Tasks

In this section...
"Attaching a Generator" on page 15-21
"Removing a Generator" on page 15-21

Attaching a Generator

1 Right-click the input to a block (such as a gain block), and from the signal context menu select **Create & Connect Signal Generator**.

2 From the list that appears, select the type of scope to be attached.

The generator will appear in the block diagram as a small rectangle that lists the generator type you have chosen. Double-click this region to display a dialog from where you can change the generators properties.

You can also attach generators with the Signal and Scope Manager. To learn how to do this, see "Using the Signal and Scope Manager" on page 8-39.

Removing a Generator

To remove the generator from the block diagram:

1 Right-click a generator.

2 From the context menu, select **Disconnect Generator**.

16

Analyzing Simulation Results

Viewing Output Trajectories

In this section...
"Using the Scope Block" on page 16-2
"Using Return Variables" on page 16-2
"Using the To Workspace Block" on page 16-3

Using the Scope Block

You can display output trajectories on a Scope block during simulation as illustrated by the following model.

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

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

Using Return Variables

By returning time and output histories, you can use the plotting commands provided in the MATLAB® software to display and annotate the output trajectories.
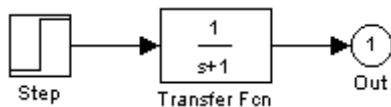
The block labeled Out is an Outport block from the Ports & Subsystems library. The output trajectory, yout, is returned by the integration solver. For more information, see "Data Import/Export Pane".

You can also run this simulation from the **Simulation** menu by specifying variables for the time, output, and states on the **Data Import/Export** pane of the **Configuration Parameters** dialog box. You can then plot these results using

```
plot(tout,yout)
```

Using the To Workspace Block

The To Workspace block can be used to return output trajectories to the workspace. The following model illustrates this use:
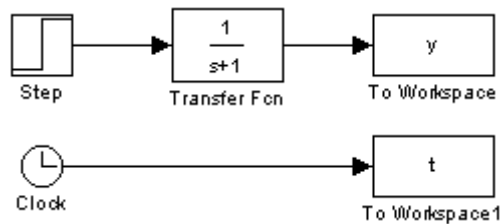
The variables y and t appear in the workspace when the simulation is complete. You store the time vector by feeding a Clock block into a To Workspace block. You can also acquire the time vector by entering a variable name for the time on the **Data Import/Export** pane of the **Configuration Parameters** dialog box, for menu-driven simulations, or by returning it using the sim command (see "Data Import/Export Pane" for more information).

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

Linearizing Models

In this section...
"About Linearizing Models" on page 16-4
"Linearization with Referenced Models" on page 16-6
"Linearization Using the 'v5' Algorithm" on page 16-8

About Linearizing Models

The Simulink® product provides the `linmod`, `linmod2`, and `dlinmod` functions to extract linear models in the form of the state-space matrices A, B, C, and D. State-space matrices describe the linear input-output relationship as

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where x, u, and y are state, input, and output vectors, respectively. For example, the following model is called `lmod`.
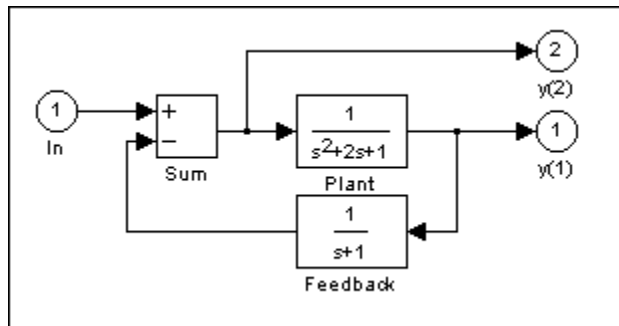
To extract the linear model of this system, enter this command.

```
[A,B,C,D] = linmod('lmod')

A =
    -2    -1    -1
     1     0     0
     0     1    -1
B =
     1
     0
     0
C =
     0     1     0
     0     0    -1
D =
     0
     1
```

Inputs and outputs must be defined using Inport and Outport blocks from the Ports & Subsystems library. Source and sink blocks do not act as inputs and outputs. Inport blocks can be used in conjunction with source blocks, using a Sum block. Once the data is in the state-space form or converted to an LTI object, you can apply functions in the Control System Toolbox™ product for further analysis:

- Conversion to an LTI object

```
sys = ss(A,B,C,D);
```

- Bode phase and magnitude frequency plot

```
bode(A,B,C,D) or bode(sys)
```

- Linearized time response

```
step(A,B,C,D) or step(sys)
impulse(A,B,C,D) or impulse(sys)
lsim(A,B,C,D,u,t) or lsim(sys,u,t)
```

You can use other functions in the Control System Toolbox and the Robust Control Toolbox™ products for linear control system design.

When the model is nonlinear, an operating point can be chosen at which to extract the linearized model. Extra arguments to linmod specify the operating point.

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

For discrete systems or mixed continuous and discrete systems, use the function dlinmod for linearization. This function has the same calling syntax as linmod except that the second right-hand argument must contain a sample time at which to perform the linearization.

Linearization with Referenced Models

You can use linmod to extract a linear model from a Simulink environment that contains Model blocks.

Note In normal simulation mode, the linmod command applies the block-by-block linearization algorithm on blocks inside the referenced model. If the model block is in accelerated mode, the linmod command uses numerical perturbation to linearize the referenced model. Due to limitations on linearizing multirate model blocks in accelerator mode, you should use normal mode simulation for all models referenced by model blocks when linearizing with referenced models. See the Control Design documentation for an explanation of the block-by-block linearization algorithm.

For example, consider the f14 model mdlref_f14.mdl. The Aircraft Dynamics Model block refers to the model mdlref_dynamics.mdl.
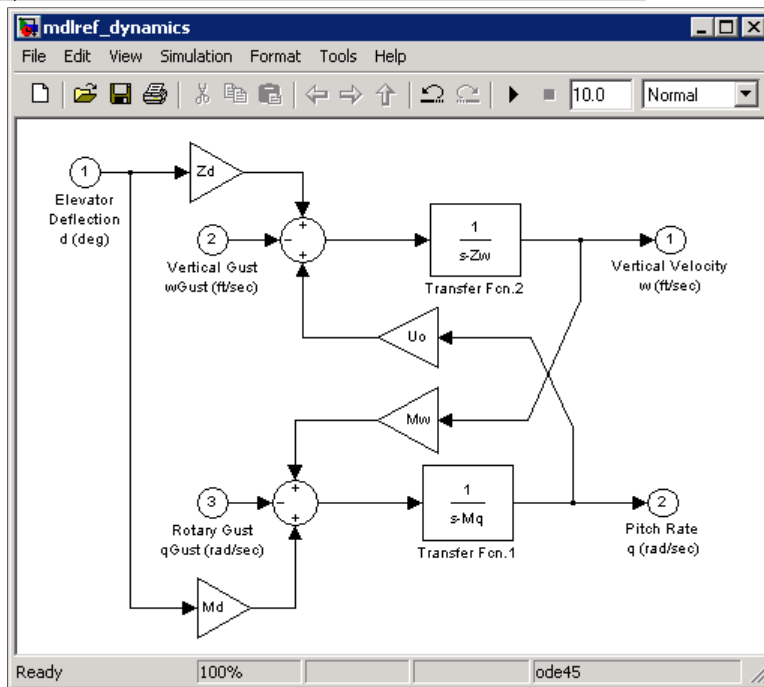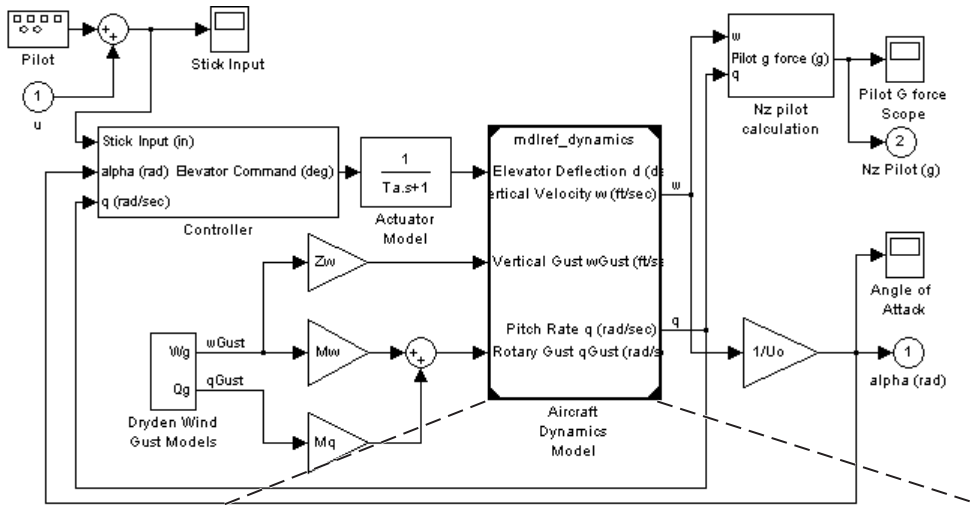
To linearize the mdlref_f14 model, call the linmod command on the top mdlref_f14 model as follows.

```
[A,B,C,D] = linmod('mdlref_f14')
```

The resulting state-space model corresponds to the complete f14 model, including the referenced model.

You can call linmod with a state and input operating point for models that contain Model blocks. When using operating points, the state vector x refers to the total state vector for the top model and any referenced models. You must enter the state vector using the structure format. To get the complete state vector, call

```
x = Simulink.BlockDiagram.getInitialState(topModelName)
```

Linearization Using the 'v5' Algorithm

Calling the linmod command with the 'v5' argument invokes the perturbation algorithm created prior to MATLAB® software version 5.3. This algorithm also allows you to specify the perturbation values used to perform the perturbation of all the states and inputs of the model.

```
[A,B,C,D]=linmod('sys',x,u,para,xpert,upert,'v5')
```

Using linmod with the 'v5' option to linearize a model that contains Derivative or Transport Delay blocks can be troublesome. Before linearizing, replace these blocks with specially designed blocks that avoid the problems. These blocks are in the Simulink Extras library in the Linearization sublibrary.

You access the Extras library by opening the Blocksets & Toolboxes icon:

- For the Derivative block, use the Switched derivative for linearization.

- For the Transport Delay block, use the Switched transport delay for linearization. (Using this block requires that you have the Control System Toolbox product.)

When using a Derivative block, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, it is better implemented (although this is not always possible) with a single Transfer Fcn block of the form

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

In this example, the blocks on the left of this figure can be replaced by the block on the right.
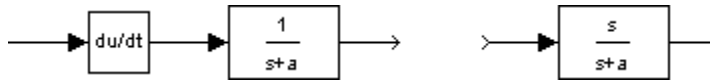
Finding Steady-State Points

The Simulink® `trim` function uses a model to determine steady-state points of a dynamic system that satisfy input, output, and state conditions that you specify. Consider, for example, this model, called `lmod`.
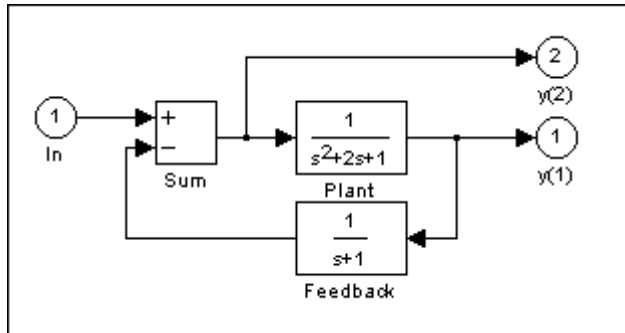
You can use the `trim` function to find the values of the input and the states that set both outputs to 1. First, make initial guesses for the state variables (x) and input values (u), then set the desired value for the output (y).

```
x = [0; 0; 0];
u = 0;
y = [1; 1];
```

Use index variables to indicate which variables are fixed and which can vary.

```
ix = [];      % Don't fix any of the states
iu = [];      % Don't fix the input
iy = [1;2];   % Fix both output 1 and output 2
```

Invoking `trim` returns the solution. Your results might differ because of roundoff error.

```
[x,u,y,dx] = trim('lmod',x,u,y,ix,iu,iy)

x =
    0.0000
    1.0000
    1.0000
u =
    2
y =
    1.0000
    1.0000
dx =
    1.0e-015 *
    -0.2220
    -0.0227
     0.3331
```

Note that there might be no solution to equilibrium point problems. If that is the case, `trim` returns a solution that minimizes the maximum deviation from the desired result after first trying to set the derivatives to zero. For a description of the `trim` syntax, see `trim` in the *Simulink Reference*.

17

Creating Block Masks

About Masks

What are 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 the Simulink® software 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.

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. The Simulink product uses MATLAB® code that you supply to draw the custom icon. You can use any of the MATLAB drawing commands in the icon code. This gives you great flexibility in designing an icon for a masked subsystem.

Mask Parameters

You can define a set of user-settable parameters for a masked subsystem. The value of a parameter is stored in the mask workspace (see "Mask Workspace" on page 17-4) 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 17-39).

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 17-31 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 the Simulink software 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 17-33). You can use the initialization code to set the initial values of the masked subsystem's mask parameters.

Mask Workspace

A workspace is associated with each masked subsystem that you create. The current values of the subsystem's parameters are stored in the workspace as well as any variables created by the block's initialization code and parameter callbacks. 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 must be set to All or ExplicitOnly. See "Resolving Symbols" on page 3-69, "Hierarchical Symbol Resolution" on page 3-70, and Permit Hierarchical Resolution for more information.

- 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.

Hierarchical Symbol Resolution in Mask Workspaces. When the Simulink software tries to resolve a symbol defined in a masked block, it searches the mask workspaces in hierarchical order, then proceeds upward to the model and base workspaces. See "Resolving Symbols" on page 3-69 for information about symbol resolution.

Creating Masks

See "Masking a Subsystem" on page 17-14 for an overview of the process of creating a masked subsystem. See "Masked Subsystem Example" on page 17-6 for an example of the process.

Masked Subsystem Example

Introduction to Example

This simple subsystem (masking_example) models the equation for a line, y = mx + b.
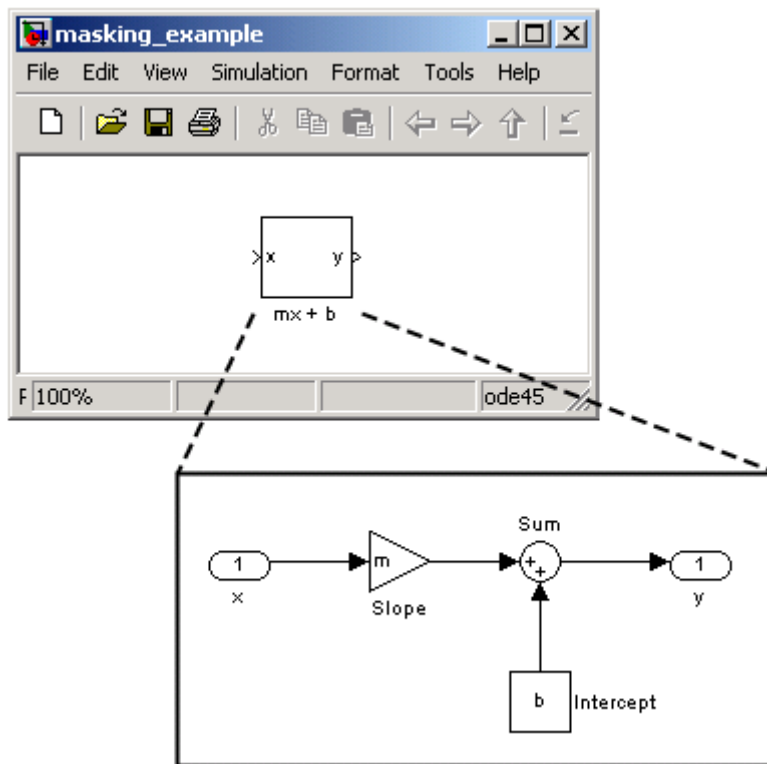
Ordinarily, when you double-click a Subsystem block, the Subsystem block opens, displaying its blocks in a separate window. The mx + b subsystem contains a Gain block, named Slope, whose **Gain** parameter is specified as m, and a Constant block, named Intercept, whose **Constant value** parameter is 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:
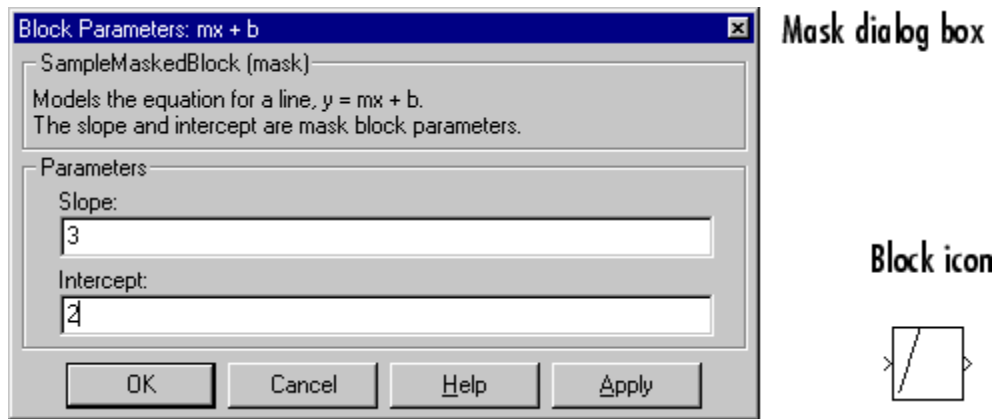
A user enters values for **Slope** and **Intercept** in the mask dialog box. These values are available to all the blocks in the underlying subsystem. Masking this subsystem creates a self-contained functional unit with its own application-specific parameters, **Slope** and **Intercept**. The mask maps 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 MATLAB® 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.

17-7

- Enter the documentation of the block, consisting of the block description and the block help text.

- Specify the drawing command that creates the block icon.

- Specify the commands that provide the variables needed by the drawing command (there are none in this example).

Creating Mask Dialog Box Prompts

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

This example primarily uses the Mask Editor's **Parameters** pane to create the masked subsystem's dialog box (see "Parameters Pane" on page 17-24).
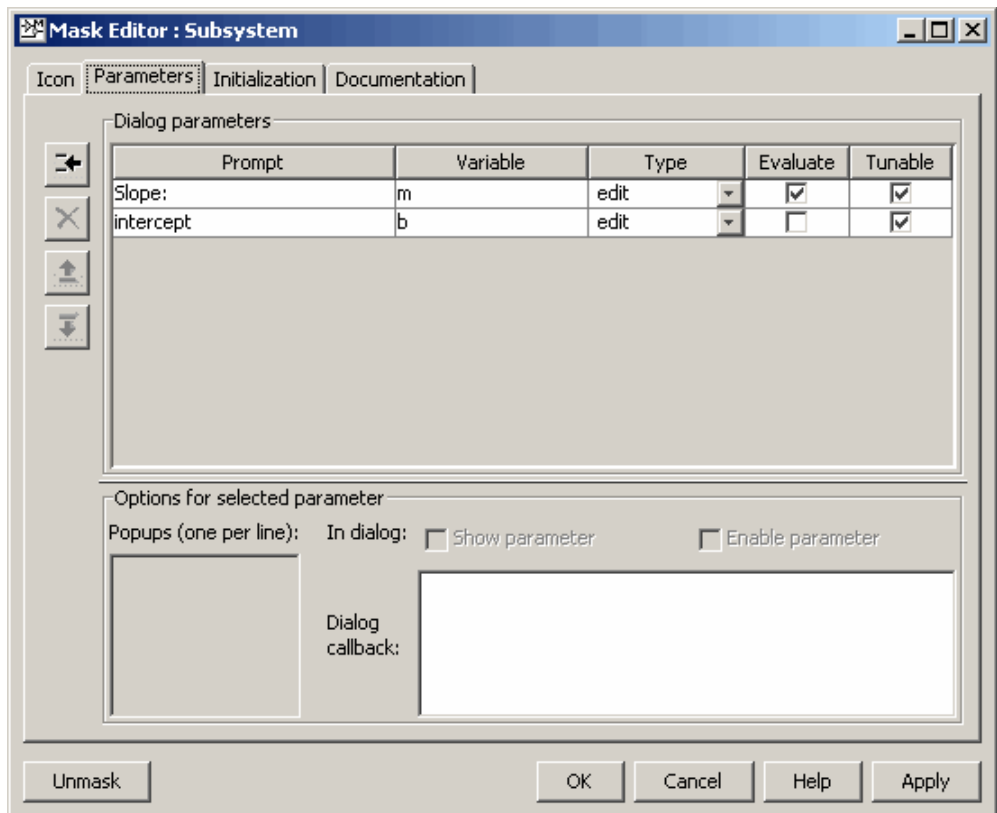
The Mask Editor enables you to specify these attributes of a mask parameter:

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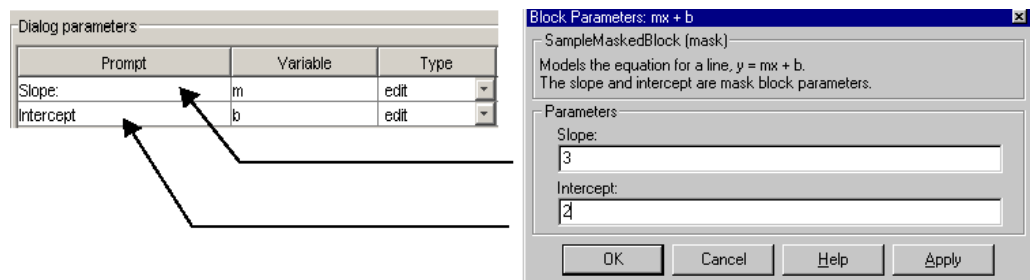
After you create the mask parameters for slope and intercept, click **OK**. Then double-click the Subsystem block to open the newly constructed dialog box. Enter 3 for the **Slope** and 2 for the **Intercept** parameter.

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 17-36). For this sample masked block, the pane looks like this.
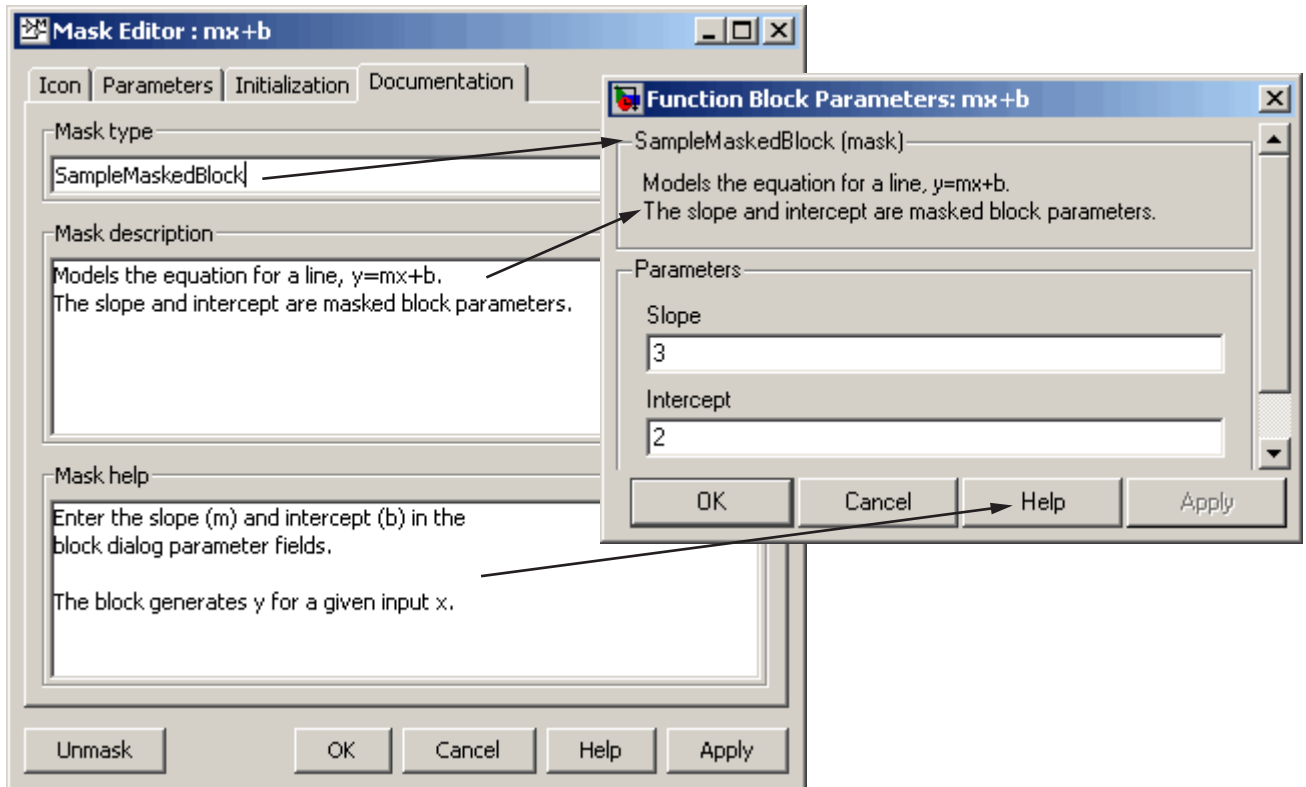
Creating the Block Icon

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

The block icon is defined on the **Icon** pane. For this block, the **Icon** pane looks like this.
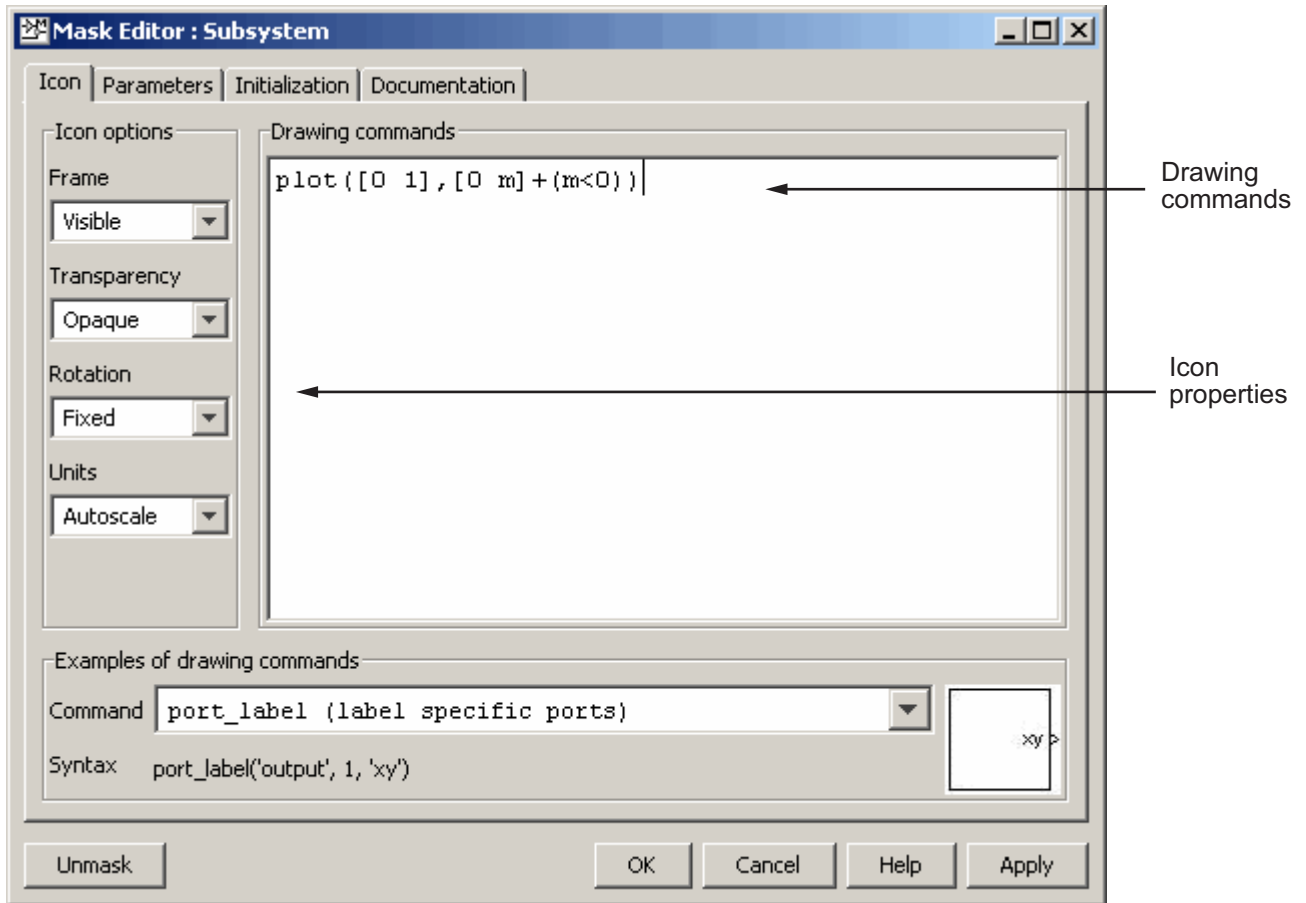
The drawing command

```
plot([O 1],[O m]+(m<O))
```

plots a line from `(0,0)` to `(1,m)`. If the slope is negative, the line is shifted 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 17-20 for more information.

Masking a Block

In this section...

About Block Masking

You use the model editor to mask Subsystem, Model, and S-function blocks, and the set_param command to mask built-in blocks.

Note The Simulink® product 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. Doing so will result in an error message being displayed.

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.
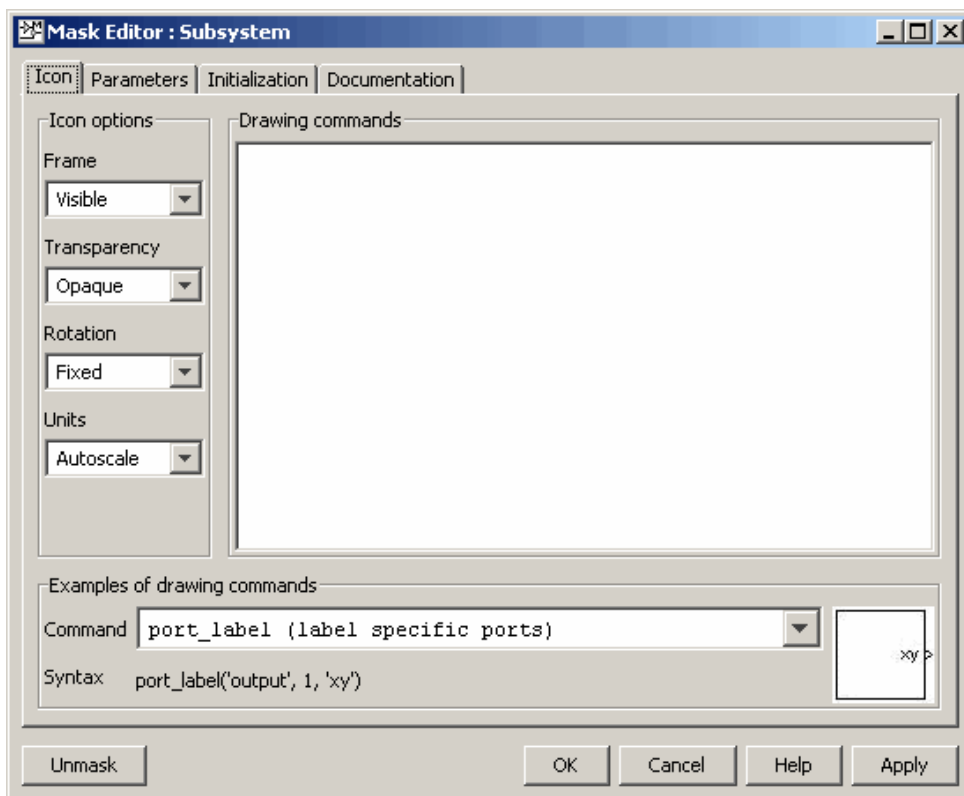
See "Mask Editor" on page 17-17 for a detailed description of the Mask Editor.

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 17-20).

- Create parameters that allow a user to set subsystem options (see "Mask Editor" on page 17-17).

- 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 17-17 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 17-20).

- Create a custom mask parameter dialog box for the block (see "Parameters Pane" on page 17-24).

- 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

Note The Mask Editor requires that MATLAB® start with Java™ enabled. This means you cannot use the Mask Editor if you start the MATLAB product with the -nojvm option.

Creating a Subsystem 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 17-16.

- 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.
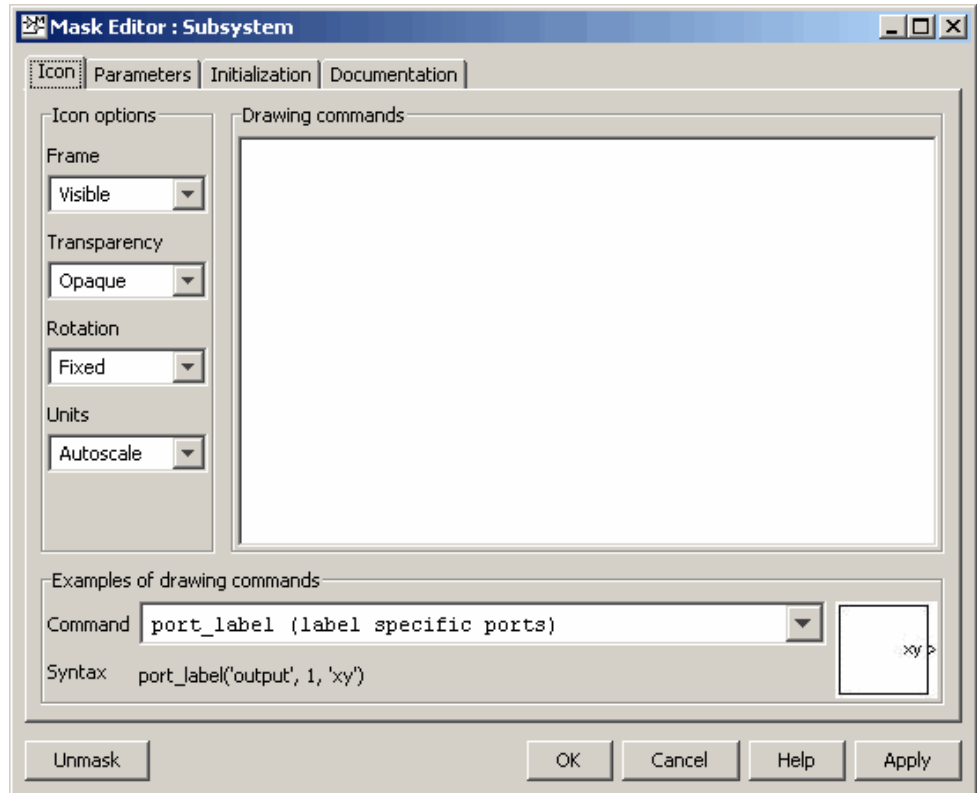
The Mask Editor contains a set of tabbed panes, each of which enables you to define a feature of the mask:

- The **Icon** pane enables you to define the block icon (see "Icon Pane" on page 17-20).

- 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 17-24).

- The **Initialization** pane enables you to specify initialization commands (see "Initialization Pane" on page 17-33).

- The **Documentation** pane enables you to define the mask type and specify the block description and the block help (see "Documentation Pane" on page 17-36).

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, the mask information is still retained 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, the inactive mask information is discarded. 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.
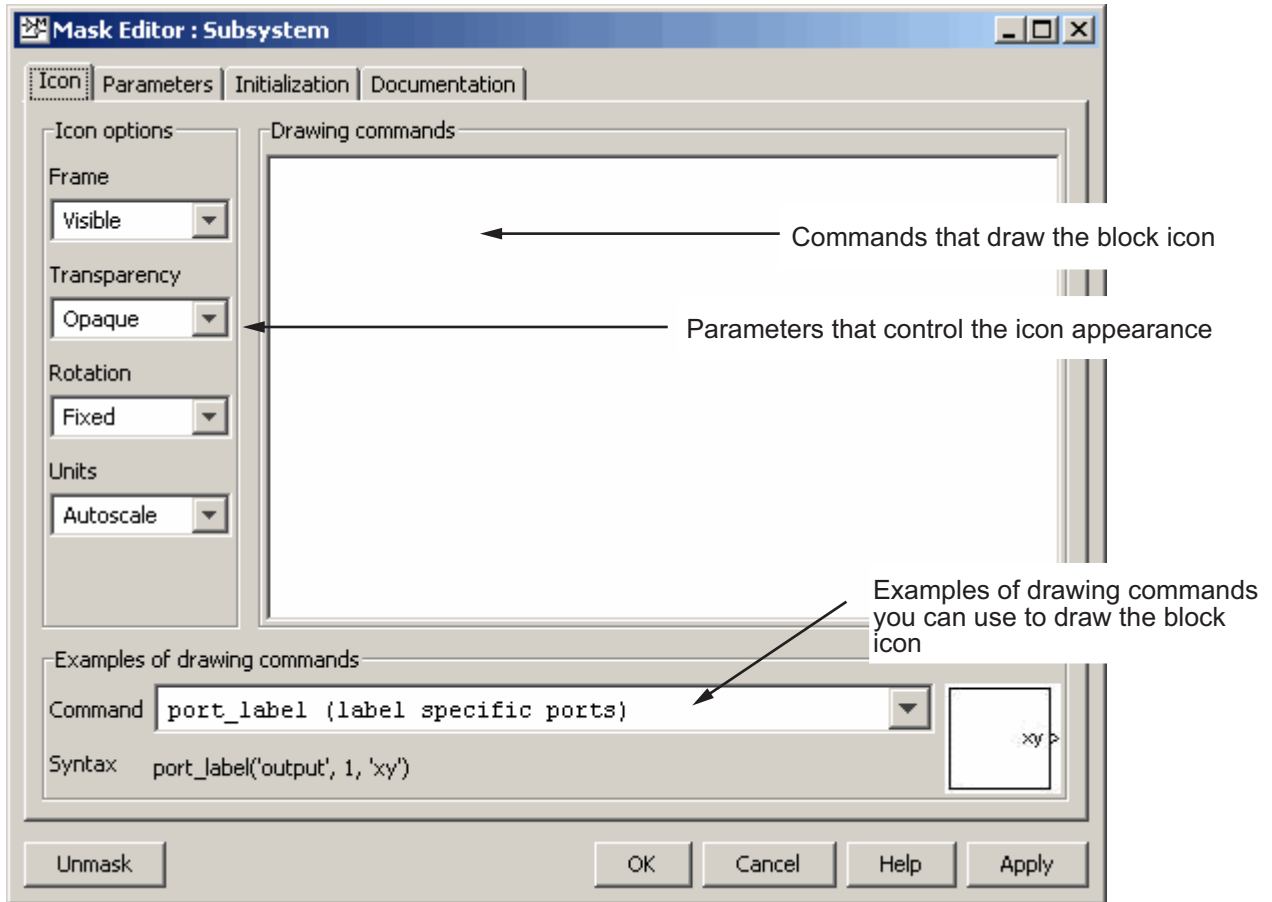
The **Icon** pane contains the following controls.

Drawing commands

This field allows you to enter commands that draw the block's icon. A set of commands is provided 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. The drawing commands are executed in the order in which they appear in this field. Drawing commands have access to all variables in the mask workspace. If the drawing commands cannot be successfully executed, 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).

The drawing commands are executed 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 the Simulink product. To determine the syntax of a command, select the command from the **Command** list. An example of the selected

command is displayed 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.
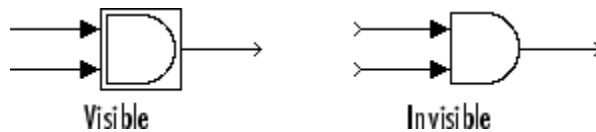
Transparency. The icon can be set to Opaque or Transparent, either hiding or showing what is underneath the icon. Opaque, the default, covers information the Simulink software draws, such as port labels. This figure shows opaque and transparent icons for an AND gate block. Notice the text on the transparent icon.
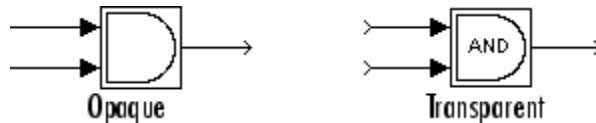
Rotation. When the block is rotated or flipped, you can choose whether to rotate or flip the icon or to have it remain fixed in its original orientation. The default is not to rotate the icon. The icon rotation is consistent with block port rotation. This figure shows the results of choosing Fixed and 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`.
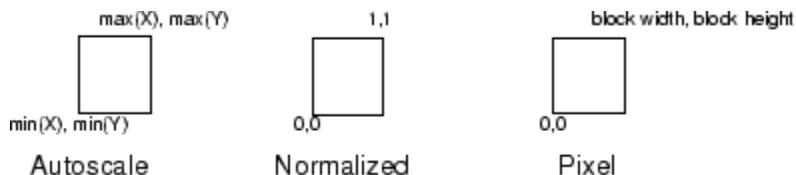
- `Autoscale` scales the icon to fit the block frame. When the block is resized, the icon is also resized. For example, this figure shows the icon drawn using these vectors:

  ```
  X = [0 2 3 4 9]; Y = [4 6 3 5 8];
  ```
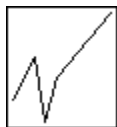

  The lower-left corner of the block frame is (0,3) and the upper-right corner is (9,8). The range of the *x*-axis is 9 (from 0 to 9), while the range of the *y*-axis is 5 (from 3 to 8).

- `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.
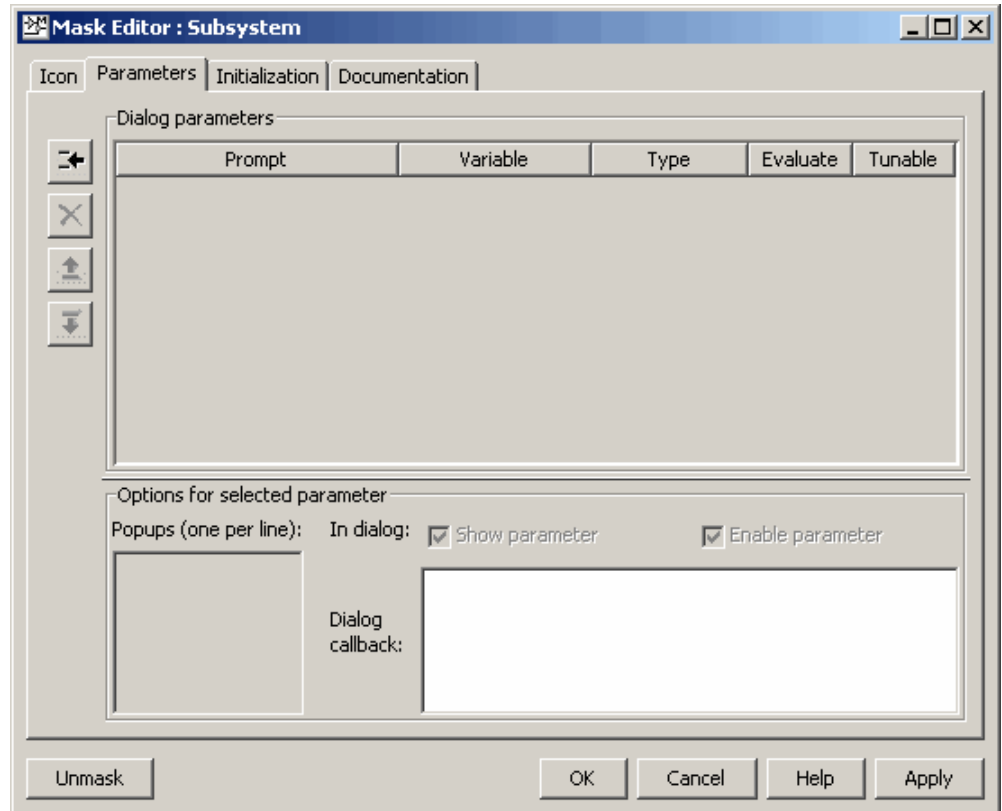
The **Parameters** pane contains the following elements:

- The **Dialog parameters** panel allows you to select and change the major properties of the mask's parameters (see "Dialog Parameters Panel" on page 17-25).

- 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 17-25).

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.
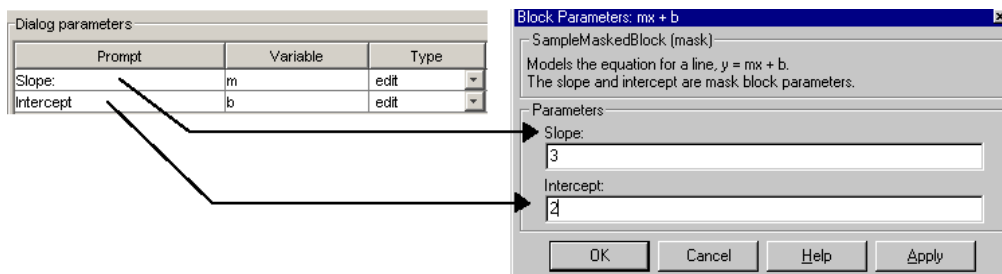
Variable. Name of the variable that stores the parameter's value in the mask's workspace (see "Mask Workspace" on page 17-4). You can use this variable as the value of parameters of blocks inside the masked subsystem, thereby allowing the user to set the parameters via the mask dialog box.

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 supported commands (see "Control Types" on page 17-30). To change the current control type, select another type from the list.

Evaluate. If checked, this option causes the Simulink software to evaluate the expression entered by the user before it is assigned to the variable. Otherwise, the expression itself is treated as a string value and is assigned to the variable. For example, if the user enters the expression `gain` in an edit field and the **Evaluate** option is checked, `gain` is evaluated and the result is assigned to the variable. Otherwise, the string `'gain'` is assigned to the variable. See "Check Box Control" on page 17-31 and "Pop-Up Control" on page 17-31 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 This setting is ignored 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 6-17 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 the MATLAB code that you want the Simulink software to execute when a user applies a change to the selected parameter, i.e., selects the **Apply** or **OK** button on the mask 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 17-40 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 17-33).

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
```

The Simulink software 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 17-33.)

- 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 17-52.

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.
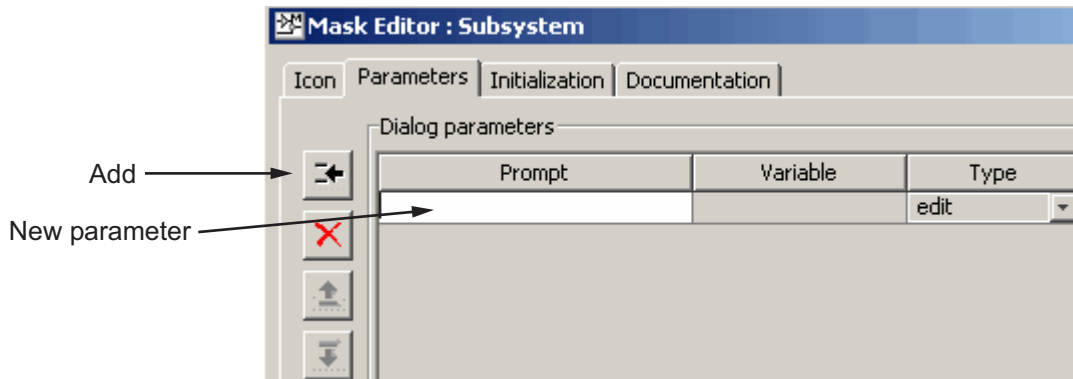
Delete Button. Deletes the parameter currently selected in the **Dialog parameters** table.

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 17-3) 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

You can 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).
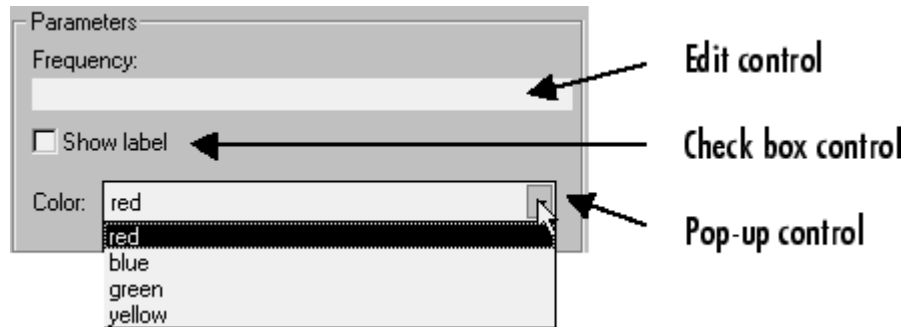
Edit Control

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.

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.

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 17-27). The following example shows a pop-up parameter.
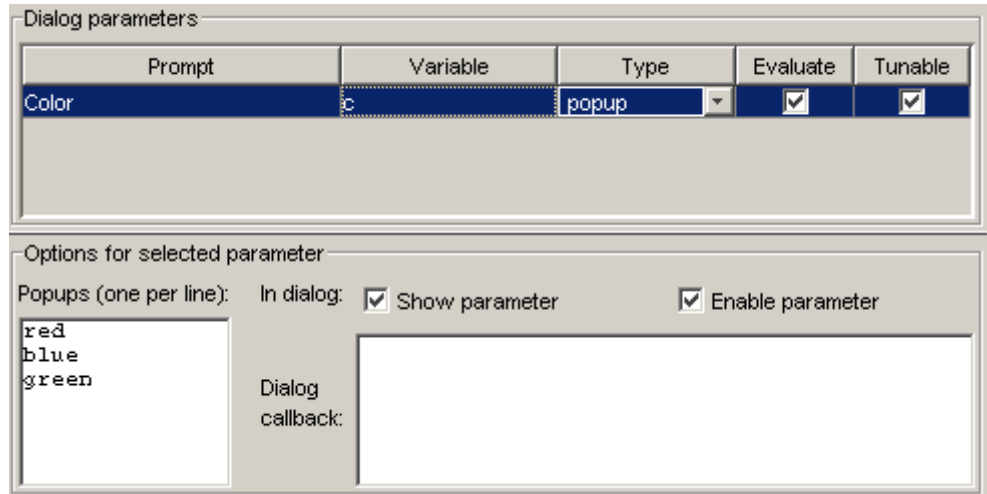
The value of the variable associated with the parameter (Color) depends on the item selected from the pop-up list and whether the **Evaluate** option is checked (on).

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.
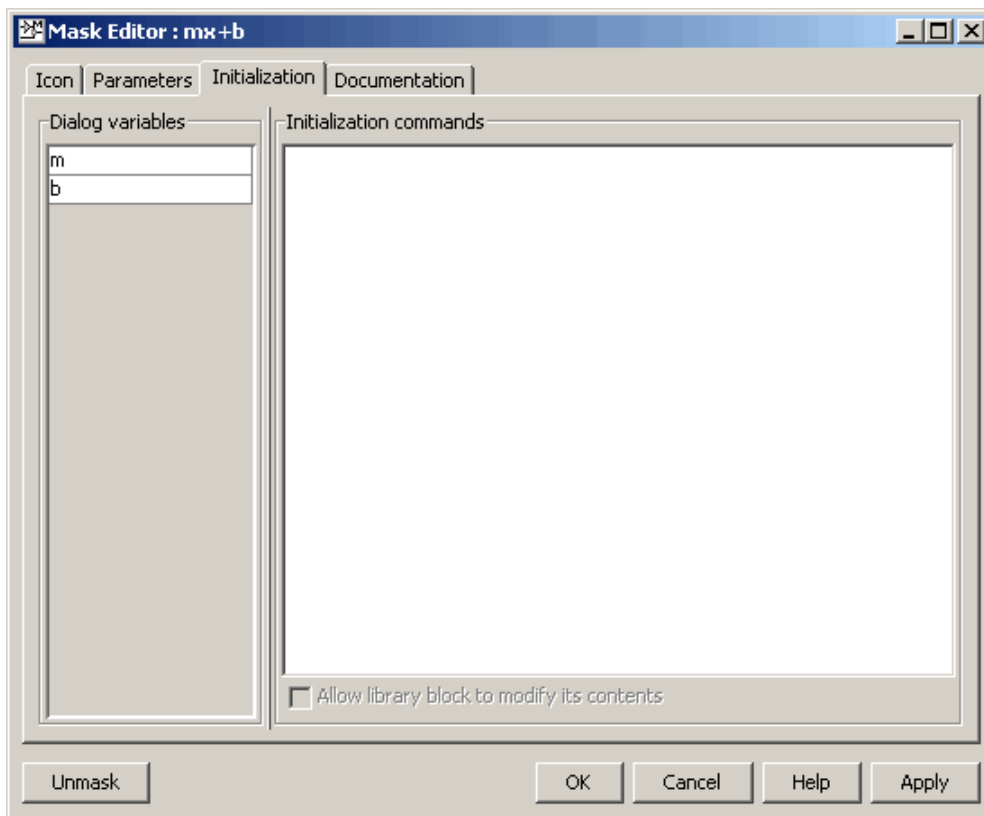
The initialization commands are executed when you

- 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 17-52.

Allow library block to modify its contents

This check box is enabled only if the masked subsystem resides in a library. Checking this option allows the block's initialization code to modify the contents of the masked subsystem, i.e., it lets the code add or delete blocks and set the parameters of those blocks. Otherwise, an error is generated when a masked library block tries to modify its contents in any way. To set this option at the MATLAB prompt, select the self-modifying block and enter the following command.

```
set_param(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 17-40 for more information).

- Avoid prefacing variable names in initialization commands with L_ and M_ to prevent undesirable results. These specific prefixes are reserved for use with internal variable names.

- Avoid using set_param commands to set parameters of blocks residing in masked subsystems that reside in the masked subsystem being initialized. Trying to set parameters of blocks in lower-level masked subsystems can trigger unresolvable symbol errors if lower-level masked subsystems reference symbols defined by higher-level masked subsystems. Suppose, for example, a masked subsystem A contains masked subsystem B, which contains Gain block C, whose Gain parameter references a variable defined by B. Suppose also that subsystem A's initialization code contains the command

```
set_param([gcb '/B/C'], 'SampleTime', '-1');
```

Simulating or updating a model containing A causes an unresolvable symbol error.

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.
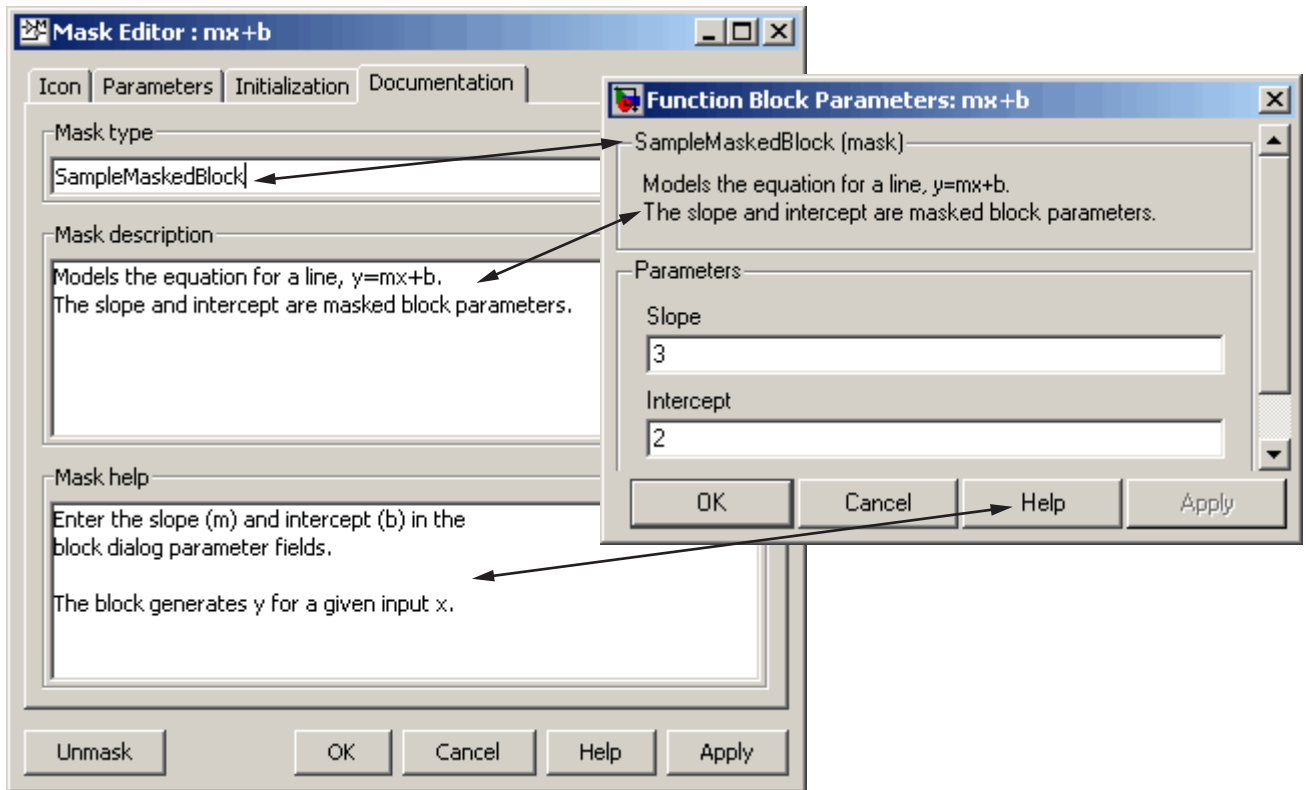
Mask Type Field

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 the Simulink product 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.

Long lines of text are automatically wrapped. 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

The first line of the masked block help text is examined. If a URL specification is detected, for example,

```
http://www.mathworks.com
```

or

```
file:///c:/mydir/helpdoc.html
```

the specified file is displayed in the browser. If a `web` command is detected, for example,

```
web([docroot '/My Blockset Doc/' get_param(gcb,'MaskType')...
'.html'])
```

or an `eval` command, for example,

```
eval('!Word My_Spec.doc')
```

the specified command is executed. Otherwise, the contents of the **Block Help** field is displayed, which can include HTML tags, in the browser.

Note, if you enter HTML-tagged text, the Simulink software 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.
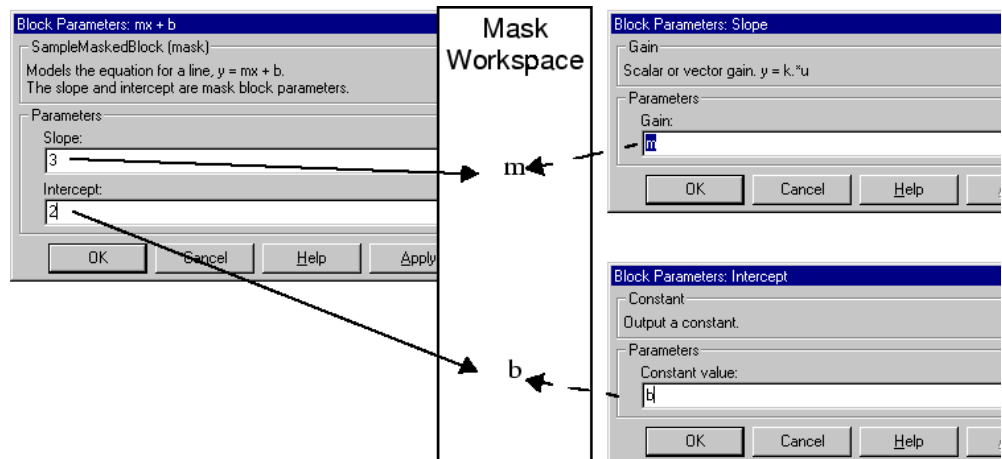
You can use a masked block's initialization code to link mask parameters indirectly to block parameters. In this approach, the initialization code creates variables in the mask workspace whose values are functions of the mask parameters and that appear in expressions that set the values of parameters of blocks concealed by the mask.

Creating Dynamic Mask Parameter Dialog Boxes

In this section...
"About Creating Masked Dialog Boxes" on page 17-40
"Setting Masked Block Dialog Parameters" on page 17-41
"Predefined Masked Dialog Parameters" on page 17-44

About Creating Masked Dialog Boxes

You can create dialogs for masked blocks whose appearance changes in response to user input. Features of masked dialog boxes that can change in this way include

- Visibility of parameter controls

  Changing a parameter can cause the control for another parameter to appear or disappear. The dialog expands or shrinks when a control appears or disappears, respectively.

- Enabled state of parameter controls

  Changing a parameter can cause the control for another parameter to be enabled or disabled for input. A disabled control is grayed to indicate visually that it is disabled.

- Parameter values

  Changing a 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 17-27). 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 17-41). 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

The Simulink® software defines a set of masked block parameters that define the current state of the masked block's dialog. You can use the mask editor to inspect and set many of these parameters. The Simulink `get_param` and `set_param` commands also let you inspect and set mask dialog parameters. The 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:
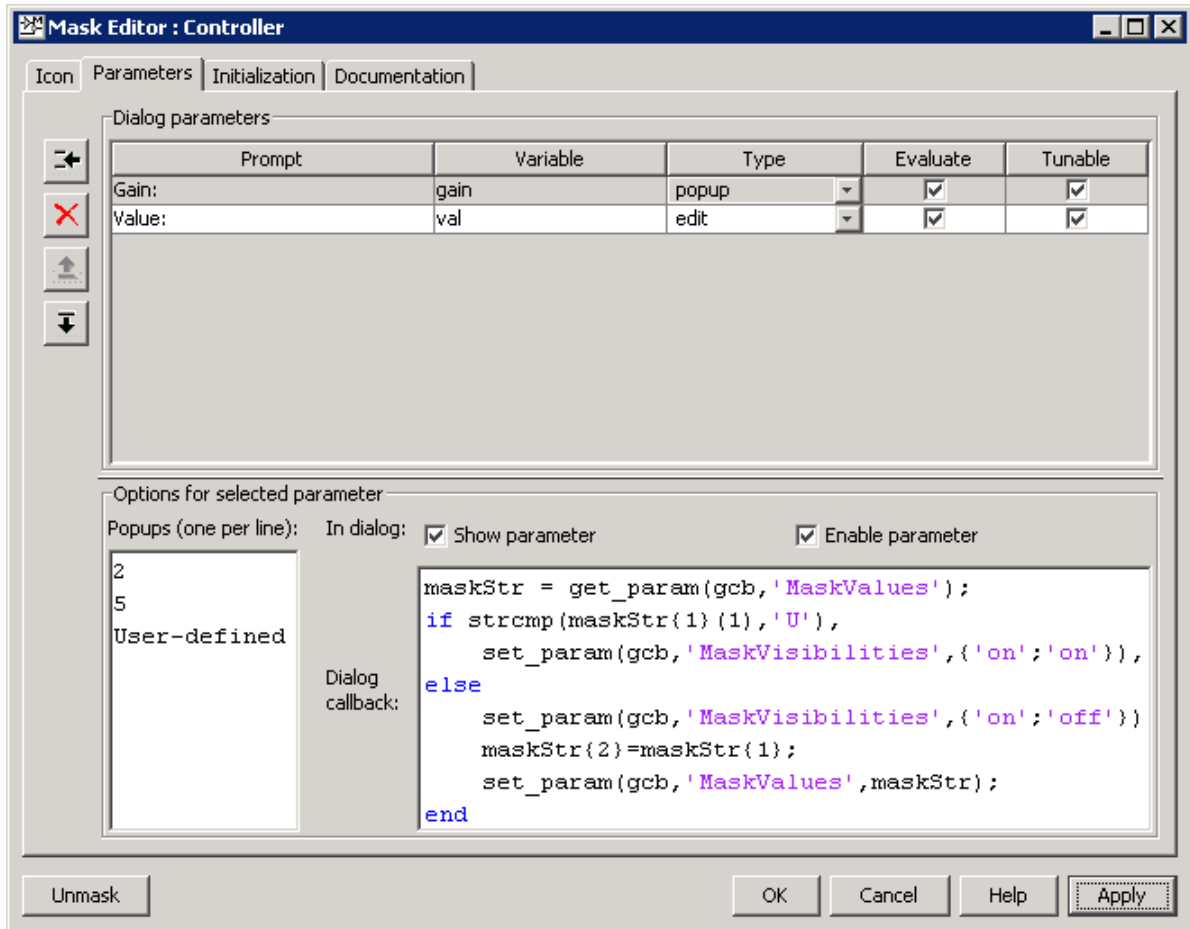
Double-clicking on the new masked subsystem opens the Mask Parameters dialog box. Selecting 2 or 5 for the **Gain** parameter hides the **Value** parameter, while selecting User-defined makes the **Value** parameter visible. Note that any blocks in the masked subsystem that need the gain value should 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

The following predefined parameters are associated with masked dialogs.

MaskCallbacks

The value of this parameter is a cell array of strings that specify callback expressions for the dialog's user-defined parameter controls. The first cell defines the callback for the first parameter's control, the second for the second parameter control, etc. The callbacks can be any valid MATLAB® expressions, including expressions that invoke M-file commands. This means that you can implement complex callbacks as M-files.

You can use either the mask editor or the MATLAB command line to specify mask callbacks. To use the mask editor to enter a callback for a parameter, enter the callback in the **Callback** field for the parameter.

The easiest way to set callbacks for a mask dialog at the MATLAB command is to first select the corresponding masked dialog in a model or library window and then to issue a set_param command at the MATLAB command line. For example, the following code

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

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

MaskDescription

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

MaskDisplay

The value of this parameter is string that specifies the drawing commands for the block's icon.

MaskEnables

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

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

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

would disable the third control of the currently open masked block's dialog. Disabled controls are colored gray to indicate visually that they are disabled.

MaskInitialization

The value of this parameter is string that specifies the initialization commands for the mask workspace.

MaskPrompts

The value of this parameter is a cell array of strings that specify prompts for user-defined parameters. The first cell defines the prompt for the first parameter, the second for the second parameter, etc.

MaskType

The value of this parameter is the mask type of the block associated with this dialog.

MaskValues

The value of this parameter is a cell array of strings that specify the values of user-defined parameters for this dialog. The first cell defines the value for the first parameter, the second for the second parameter, etc.

MaskVisibilities

The value of this parameter is a cell array of strings that specify the visibility of the user-defined parameter controls for this dialog. The first cell defines the visibility of the control for the first parameter, the second for the second parameter, etc. A value of `'on'` indicates that the corresponding control is visible; a value of `'off'` indicates that the control is hidden.

You can hide or show user-defined parameter controls dynamically by setting this parameter in the callback for a control. For example, the following command in a callback

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

would hide the control for the currently selected block's second user-defined mask parameter. The Simulink software expands or shrinks a dialog to show or hide a control, respectively.

Note For a full list of predefined masked block parameters see the Mask Parameters reference page.

Masking Library Blocks

In this section...
"Why Mask Blocks?" on page 17-47
"Specifying Default Values for Library Block Mask Parameters" on page 17-47
"Creating Self-Modifying Masks for Library Blocks" on page 17-48

Why Mask Blocks?

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. You mask blocks in a library (see Chapter 7, "Working with Block Libraries") just as you would mask blocks in a Simulink® model (see "Masking a Block" on page 17-14).

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).

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).

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.
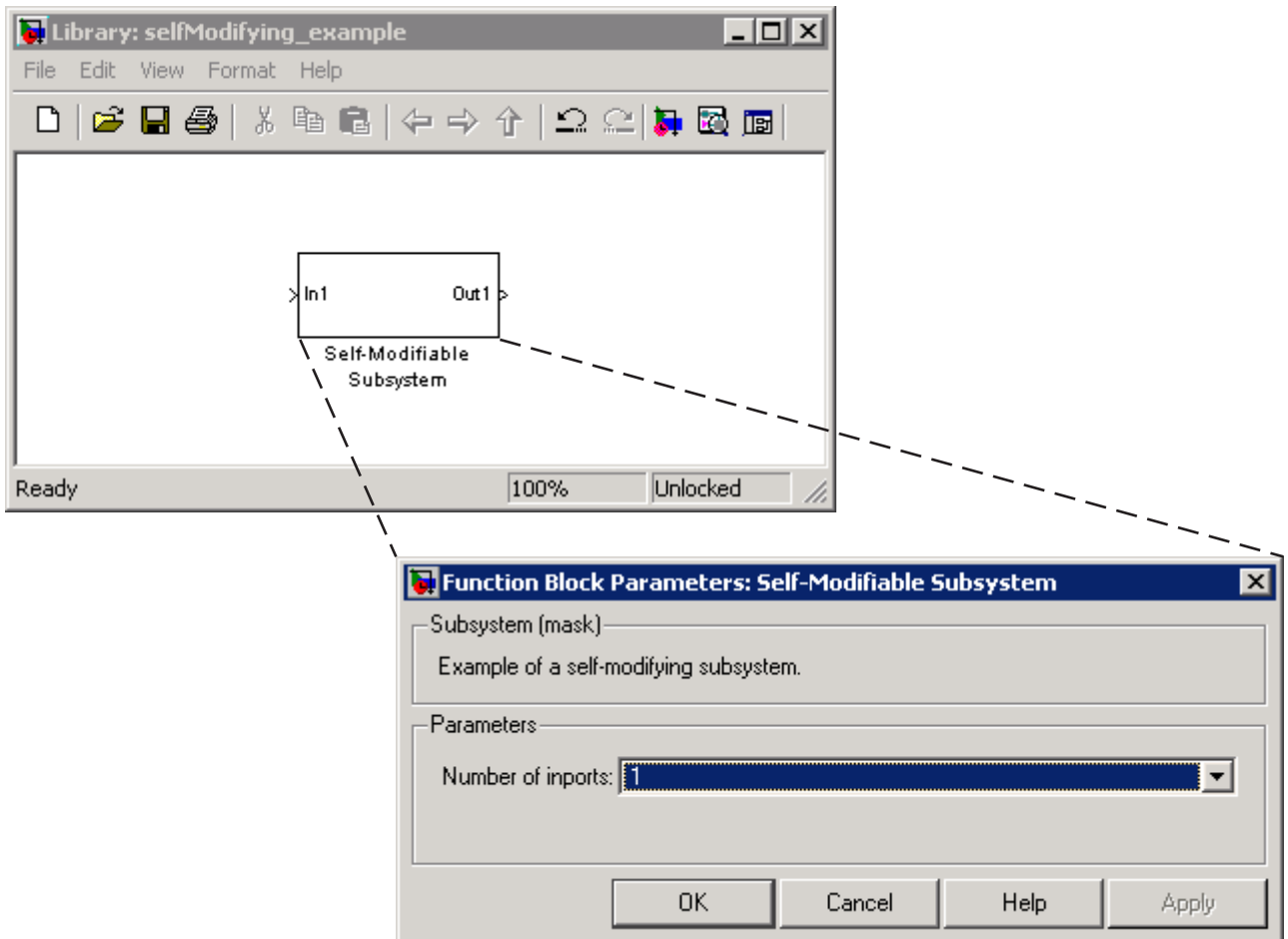
Select the subsystem then select **View Mask** from the model editor's **Edit** menu or from the block's context menu. (Right-click the block to display its context menu.) The Mask Editor opens. The Mask Editor's **Parameters** pane defines one mask parameter variable numIn that stores the value for 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.
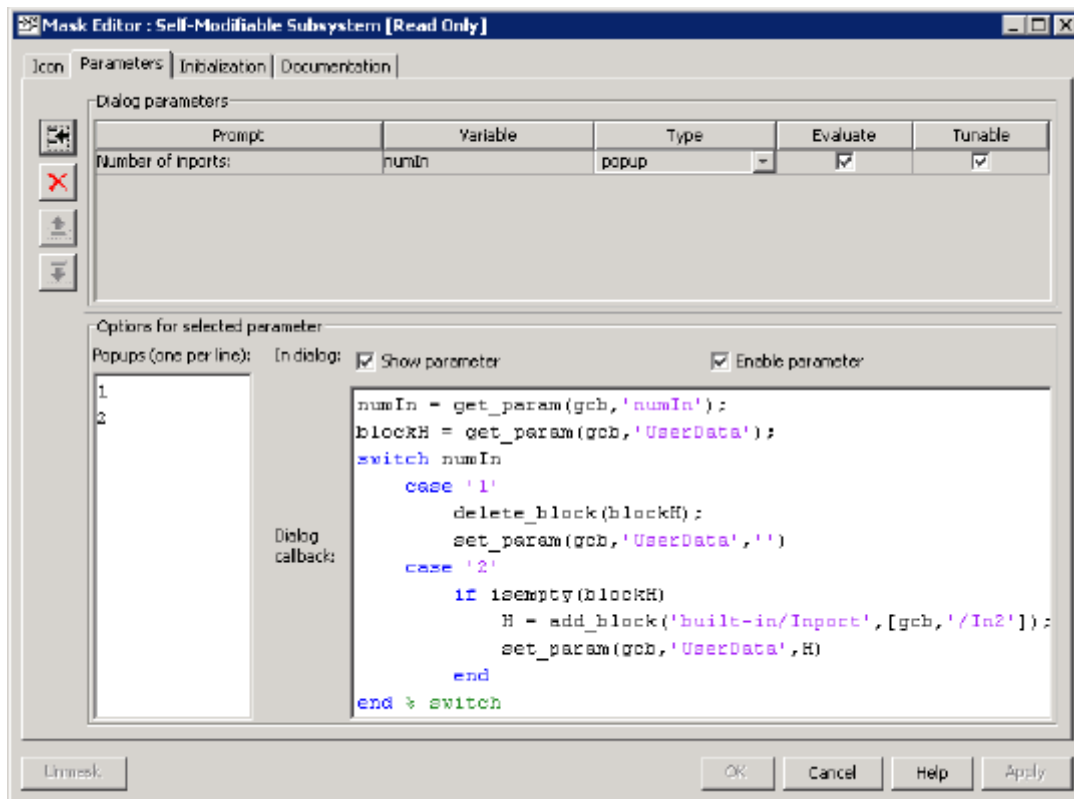
To allow the dialog callback to function properly, the **Allow library block to modify its contents** option on the Mask Editor's **Initialization** pane is selected. If this option was not selected, copies of the library block could not modify their structural contents and changing the selection in the **Number of inports** list would produce an error.
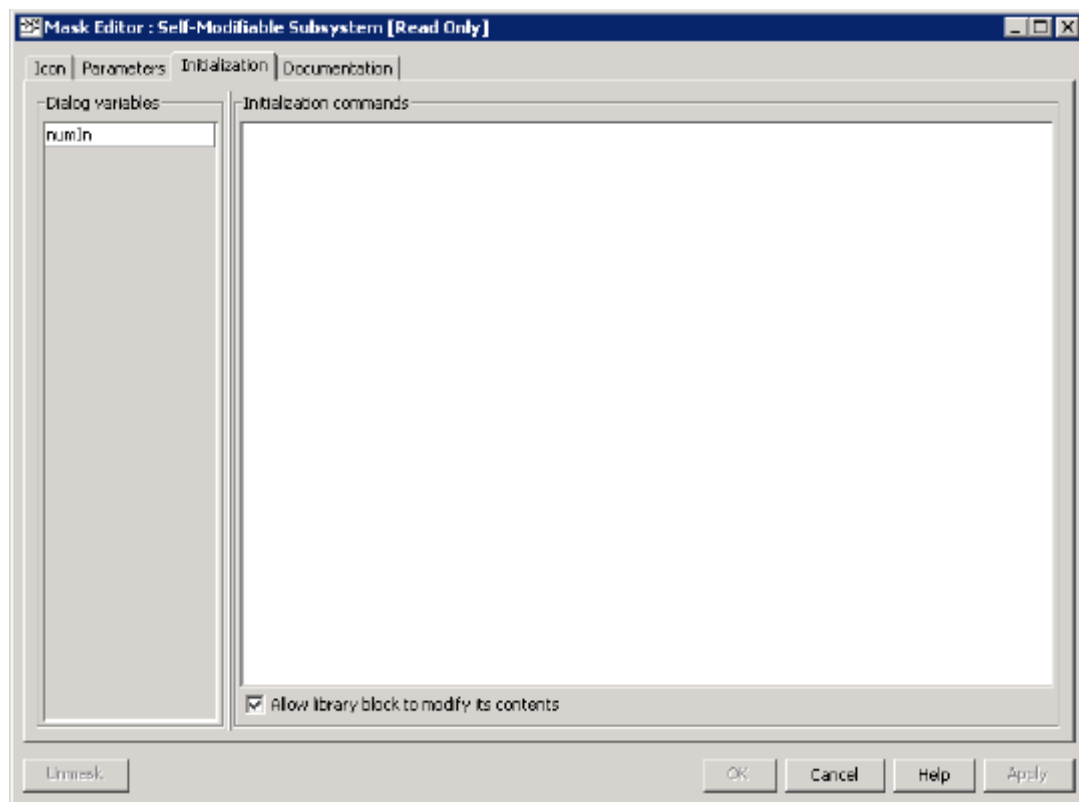
Debugging Masks

In this section...
"About Debugging Masks" on page 17-52
"Debugging Masks Using the Mask Editor" on page 17-52
"Debugging Masks Using the MATLAB® Editor/Debugger" on page 17-52

About Debugging Masks

You can use the 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.

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.
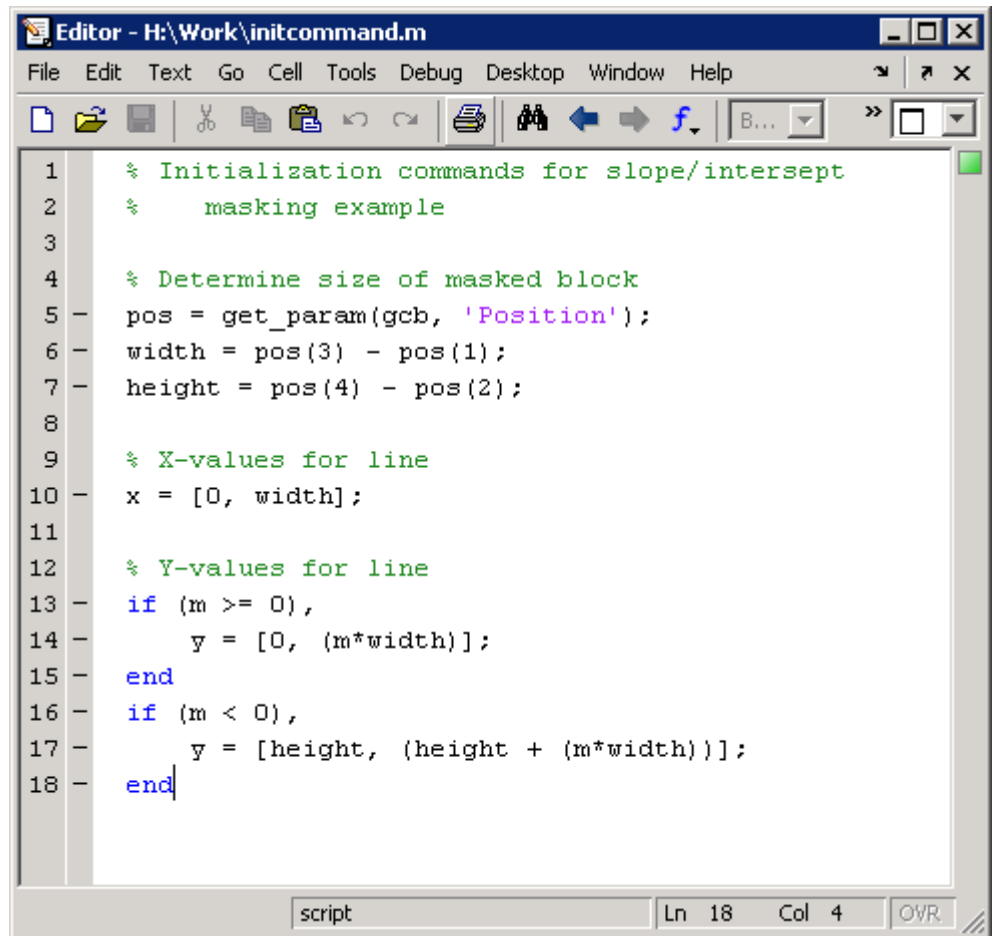
Debugging Masks Using the MATLAB® Editor/Debugger

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 17-6 for information on creating this example.
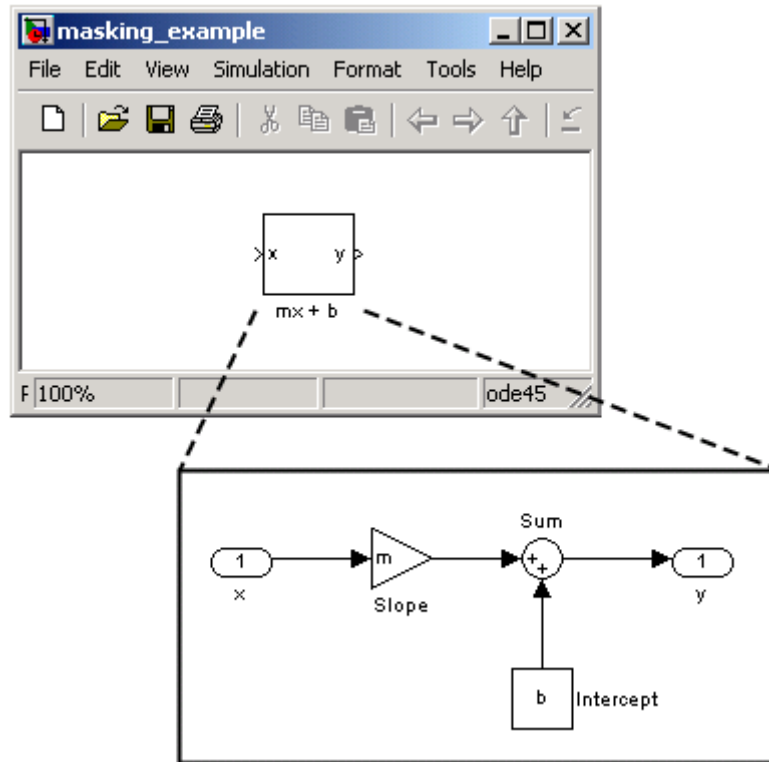
1 Write an M-file script `initcommand.m` that contains initialization commands to calculate the x and y values of the line.

```
Editor - H:\Work\initcommand.m                              _ □ ×

File   Edit   Text   Go   Cell   Tools   Debug   Desktop   Window   Help      ↘ ↗ ×

1      % Initialization commands for slope/intersept
2      %    masking example
3
4      % Determine size of masked block
5  –   pos = get_param(gcb, 'Position');
6  –   width = pos(3) – pos(1);
7  –   height = pos(4) – pos(2);
8
9      % X-values for line
10 –   x = [0, width];
11
12     % Y-values for line
13 –   if (m >= 0),
14 –       y = [0, (m*width)];
15 –   end
16 –   if (m < 0),
17 –       y = [height, (height + (m*width))];
18 –   end

                            script              Ln  18    Col  4   OVR
```

2 Enter the name of the script into the **Initialization commands** field in the Mask Editor's **Initialization** pane.
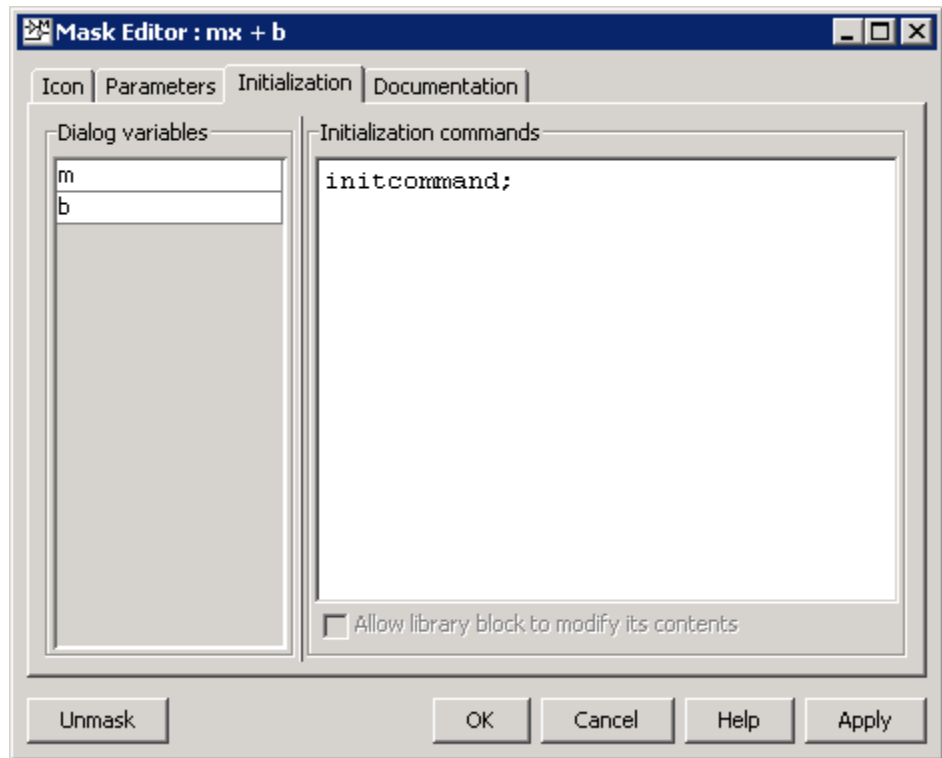
3 Use the MATLAB Editor/Debugger to place breakpoints in the M-file and step through the code. See "Editing and Debugging M-Files" in the MATLAB Desktop Tools and Development Environment for more information on editing and debugging M-files.

Note The Simulink® software 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.

18

Simulink® Debugger

Introduction to the Debugger

With the debugger you run your simulation method by method. You can stop after each method to examine the execution results. In this way you can pinpoint problems in your model to specific blocks, parameters, or interconnections.

Note Methods are functions that the Simulink® software uses to solve a model at each time step during the simulation. Blocks are made up of multiple methods. "Block execution" in this documentation is shorthand for "block methods execution." Block diagram execution is a multi-step operation that requires execution of the different block methods in all the blocks in a diagram at various points during the process of solving a model at each time step during simulation, as specified by the simulation loop.

The debugger has both a graphical and a command-line user interface. The graphical interface allows you to access the most commonly used features of the debugger. The command-line interface gives you access all of the capabilities in the debugger. If you can use either to perform a task, the documentation shows you first how to use the graphical interface, then the command-line interface.

Using the Debugger's Graphical User Interface

In this section...

Displaying the Graphical Interface

Select **Simulink Debugger** from a model window's **Tools** menu to display the debugger graphical interface.
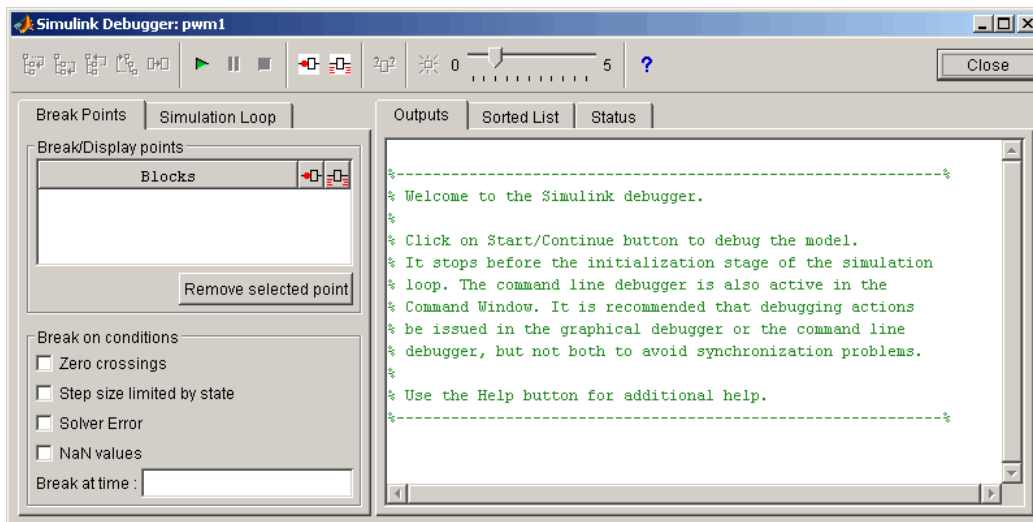
Note The debugger graphical user interface does not display state or solver information. The command line interface does provide this information. See "Displaying System States" on page 18-36 and "Displaying Solver Information" on page 18-36.

Toolbar
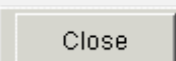
The debugger toolbar appears at the top of the debugger window.
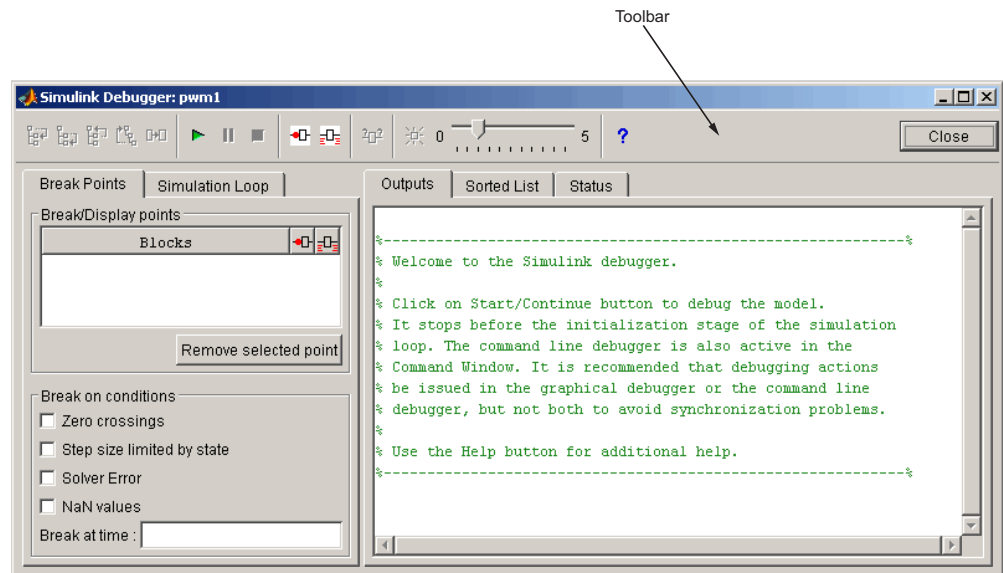
Toolbar

From left to right, the toolbar contains the following command buttons:

Button	Purpose
	Step into the next method (see "Stepping Commands" on page 18-21 for more information on this command, and the following stepping commands).
	Step over the next method.

Button	Purpose
	Step out of the current method.
	Step to the first method at the start of next time step.
	Step to the next block method.
	Start or continue the simulation.
	Pause the simulation.
	Stop the simulation.
	Break before the selected block.
	Display inputs and outputs of the selected block when executed (same as `trace gcb`).
	Display the current inputs and outputs of selected block (same as `probe gcb`).
	Toggle animation mode on or off (see "Animation Mode" on page 18-23). 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.
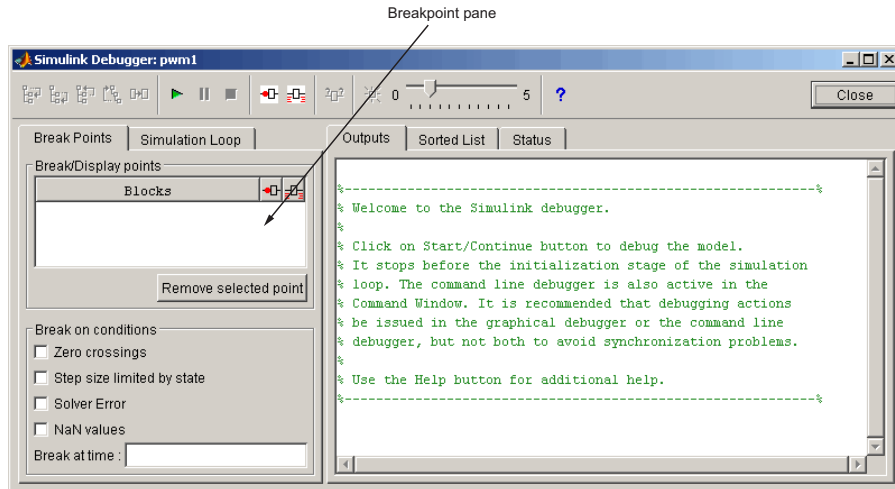
Breakpoint pane

The **Breakpoints** pane allows you to specify block methods or conditions at which to stop a simulation. See "Setting Breakpoints" on page 18-27 for more information.

Note The debugger grays out and disables the **Breakpoints** pane when you select animation mode. (see "Animation Mode" on page 18-23). This prevents you from setting breakpoints and indicates that animation mode ignores existing breakpoints.

Simulation Loop Pane

To display the **Simulation Loop** pane, select the **Simulation Loop** tab on the debugger window.

Simulation Loop pane
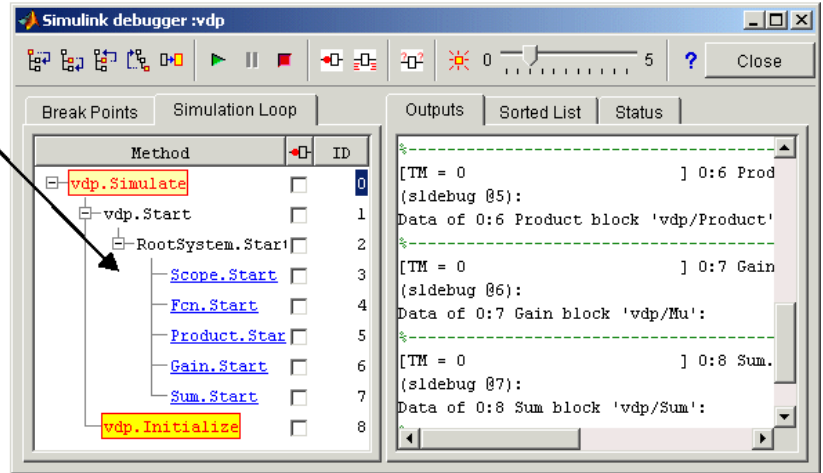
The **Simulation Loop** pane contains three columns:

- Method
- Breakpoints
- ID

Method Column

The **Method** column lists the methods that have been called thus far in the simulation as a method tree with expandable/collapsible nodes. Each node of the tree represents a method that calls other methods. Expanding a node shows the methods that the block method calls. Clicking a block method name highlights the corresponding block in the model diagram.

Whenever the simulation stops, the debugger highlights the name of the method where the simulation has stopped as well as the methods that invoked it. The highlighted method names indicate the current state of the method call stack.

Breakpoints Column

The breakpoints column consists of check boxes. Selecting a check box sets a breakpoint at the method whose name appears to the left of the check box. See "Setting Breakpoints from the Simulation Loop Pane" on page 18-29 for more information.

Note This column is disabled when its animation mode is selected (see "Animation Mode" on page 18-23), because in animation mode you can't set breakpoints.

ID Column

The ID column lists the IDs of the methods listed in the **Methods** column. See "Method ID" on page 18-11 for more information.

Outputs Pane

To display the **Outputs** pane, select the **Outputs** tab on the debugger window.
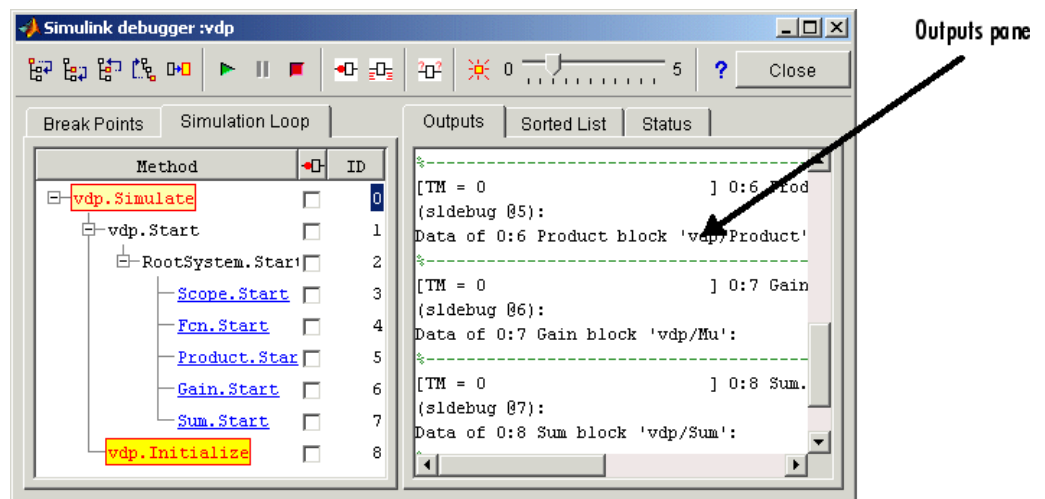
The Outputs pane displays the same debugger output that would appear in the MATLAB® command window if the debugger were running in command-line

mode. The output includes the debugger command prompt and the inputs, outputs, and states of the block at whose method the simulation is currently paused (see "Block Data Output" on page 18-20). The command prompt displays current simulation time and the name and index of the method in which the debugger is currently stopped (see "Block ID" on page 18-11).

Sorted List Pane

To display the **Sorted List** pane, select the **Sorted List** tab on the debugger window.
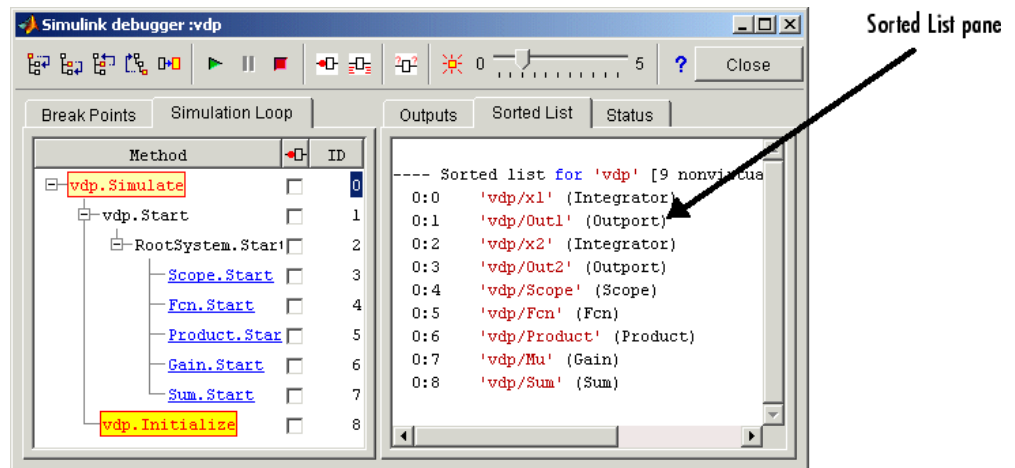
The **Sorted List** pane displays the sorted lists for the model being debugged. See "Displaying a Models Sorted Lists" on page 18-38 for more information.

Status Pane

To display the **Status** pane, select the **Status** tab on the debugger window.
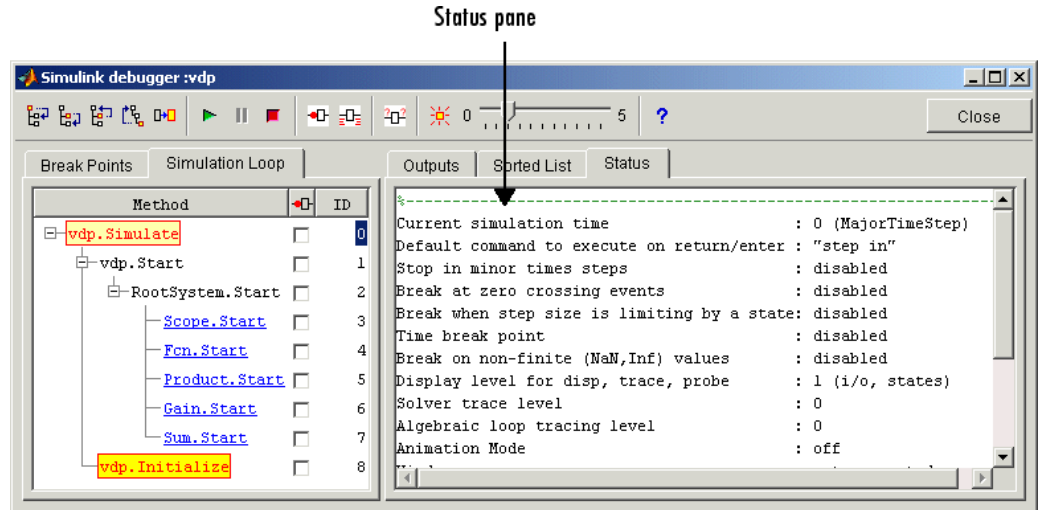
Status pane

The **Status** pane displays the values of various debugger options and other status information.

Using the Debugger's Command-Line Interface

In this section...
"Controlling the Debugger" on page 18-11
"Method ID" on page 18-11
"Block ID" on page 18-11
"Accessing the MATLAB® Workspace" on page 18-12

Controlling the Debugger

In command-line mode, you control the debugger by entering commands at the debugger command line in the MATLAB® Command Window. The debugger accepts abbreviations for debugger commands. See "Simulink® Debugger Commands — Alphabetical List" for a list of command abbreviations and repeatable commands.

Note You can repeat some commands by entering an empty command (i.e., by pressing the **Enter** key) at the command line.

Method ID

Some of the Simulink software commands and messages use method IDs to refer to methods. A method ID is an integer assigned to a method the first time the method is invoked. The debugger assigns method indexes sequentially, starting with 0.

Block ID

Some of the debugger commands and messages use block IDs to refer to blocks. Block IDs are assigned to blocks while generating the model's sorted lists during the compilation phase of the simulation. A block ID has the form sid:bid, where sid is an integer identifying the system that contains the block (either the root system or a nonvirtual subsystem) and bid is the position of the block in the system's sorted list. For example, the block index 0:1 refers to the first block in the model's root system. The slist command shows the block ID for each debugged block in the model.

Accessing the MATLAB® Workspace

You can enter any MATLAB expression at the sldebug prompt. For example, suppose you are at a breakpoint and you are logging time and output of your model as tout and yout. The following command creates a plot.

```
(sldebug ...) plot(tout, yout)
```

You cannot display the value of a workspace variable whose name is partially or entirely the same as that of a debugger command by entering it at the debugger command prompt. You can, however, use the eval command to work around this problem. For example, use eval('s') to determine the value of s rather 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 toolbar. Clicking the **Help** button displays help for the debugger in the MATLAB® product Help browser.

Help button
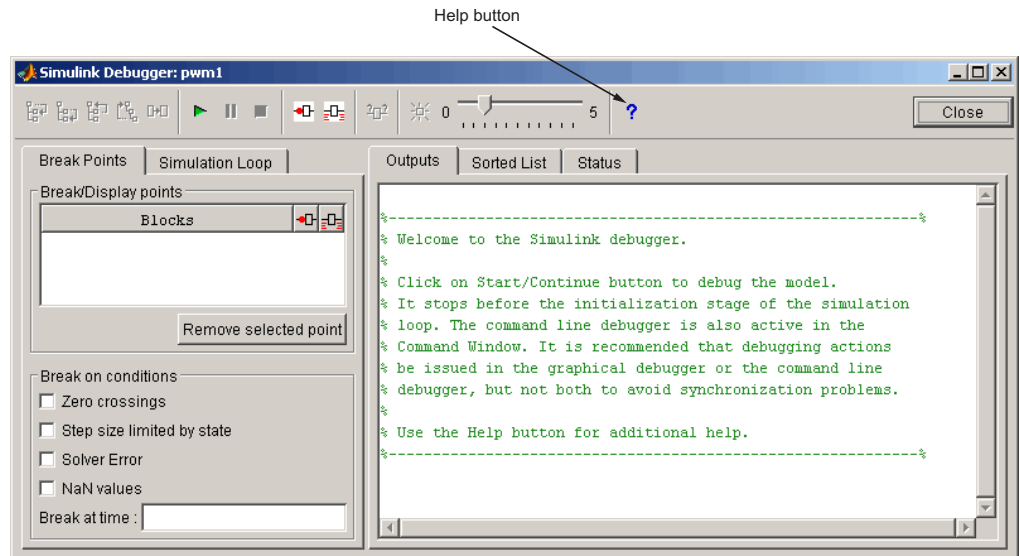
In command-line mode, you can get a brief description of the debugger commands by typing help at the debug prompt.

Starting the Debugger

You can start the debugger either from a model window or from the command line. To start the debugger from a model window, select **Simulink Debugger** from the model window **Tools** menu. The debugger graphical user interface appears (see "Using the Debugger's Graphical User Interface" on page 18-3).

To start the debugger from the MATLAB® Command Window, enter either a `sim` command or the `sldebug` command. For example the following commands load the demo model `vdp` into memory, starts the simulation, and stops the simulation at the first block in the model's execution list

```
sim('vdp',[0,10],simset('debug','on'))
```

or

```
sldebug 'vdp'
```

Note When running the debugger in graphical user interface (GUI) mode, you must explicitly start the simulation. See "Starting a Simulation" on page 18-15 for more information.

Starting a Simulation

To start the simulation, click the **Start/Continue** button on the debugger toolbar.

Start/Continue button
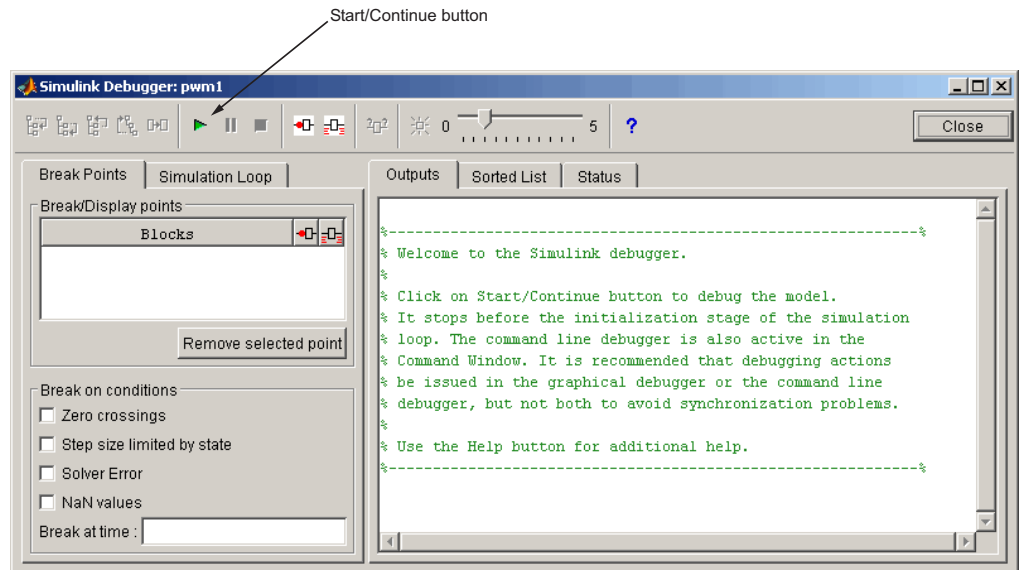
The simulation starts and stops at the first simulation method that is to be executed. It displays the name of the method in its **Simulation Loop** pane and in the debug pointer on the block diagram. The debug pointer indicates on the block diagram which block method is being executed at each step. At this point, you can

- Set breakpoints.
- Run the simulation step by step.
- Continue the simulation to the next breakpoint or end.
- Examine data.
- Perform other debugging tasks.

As the simulation progresses, the block diagram updates with debug pointers.

The debugger displays the name of the method in the Simulation Loop pane, as shown in the following figure:
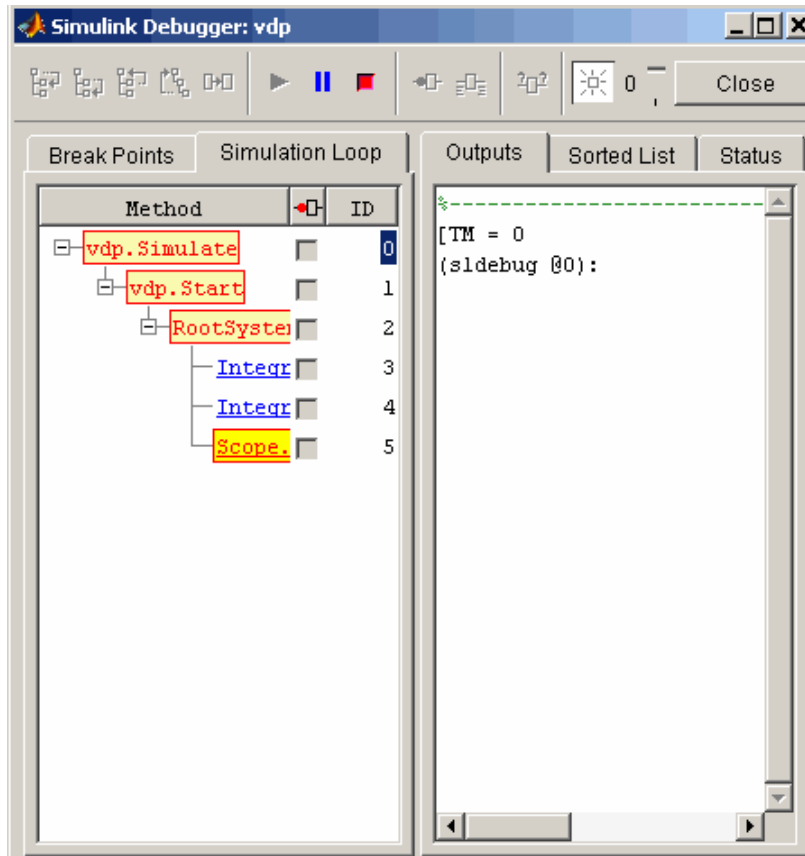
The debugger also displays a graphical debug pointer (see "Debug Pointer" on page 18-25) in the block diagram of the model that you are debugging. The debug pointer points to the first block method to be executed.

debug pointer
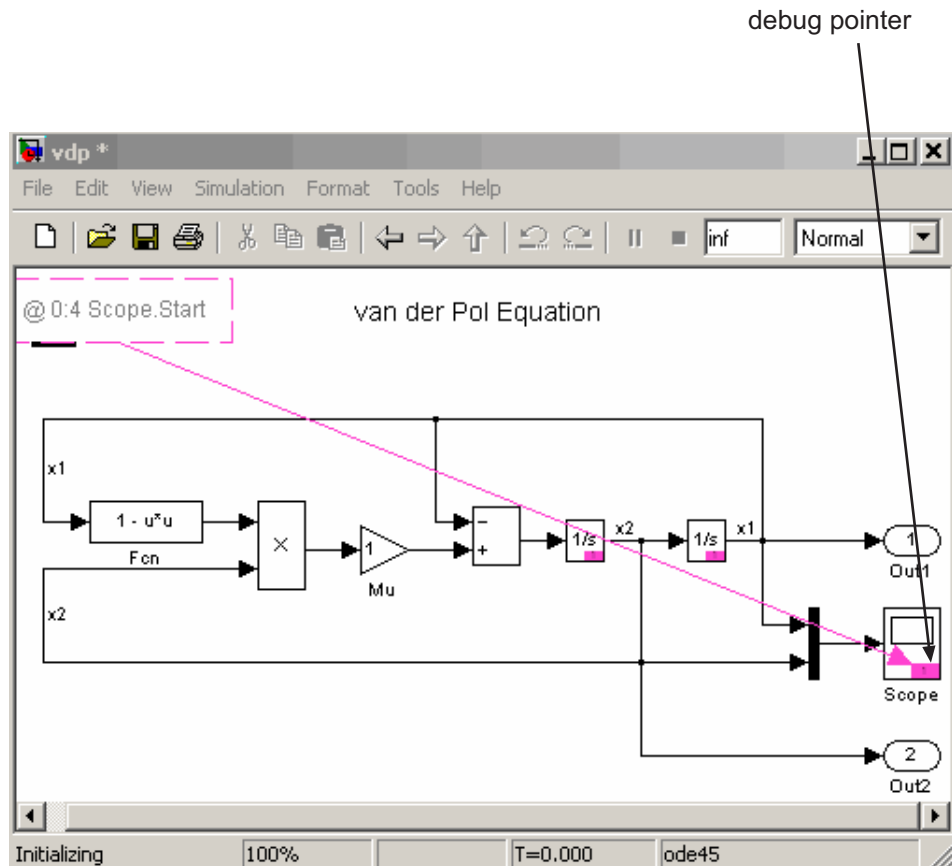
The following sections explain how to use the debugger controls to perform these debugging tasks.

Note When you start the debugger in GUI mode, the debugger command-line interface is also active in the Command Window of the MATLAB® product. However, to prevent synchronization errors between the graphical and command-line interfaces you should avoid using the command-line interface.

Running a Simulation Step by Step

In this section...

Introduction

The debugger provides various commands that let you advance a simulation from the method where it is currently suspended (the next method) by various increments (see "Stepping Commands" on page 18-21). For example, you can advance the simulation

- Into or over the next method

- Out of the current method

- To the top of the simulation loop.

After each advance, the debugger displays information that enables you to determine the point to which the simulation has advanced and the results of advancing the simulation to that point.

For example, in GUI mode, after each step command, the debugger highlights the current method call stack in the **Simulation Loop** pane. The call stack comprises the next method and the methods that invoked the next method either directly or indirectly. The debugger highlights the call stack by highlighting the names of the methods that make up the call stack in the **Simulation Loop** pane.
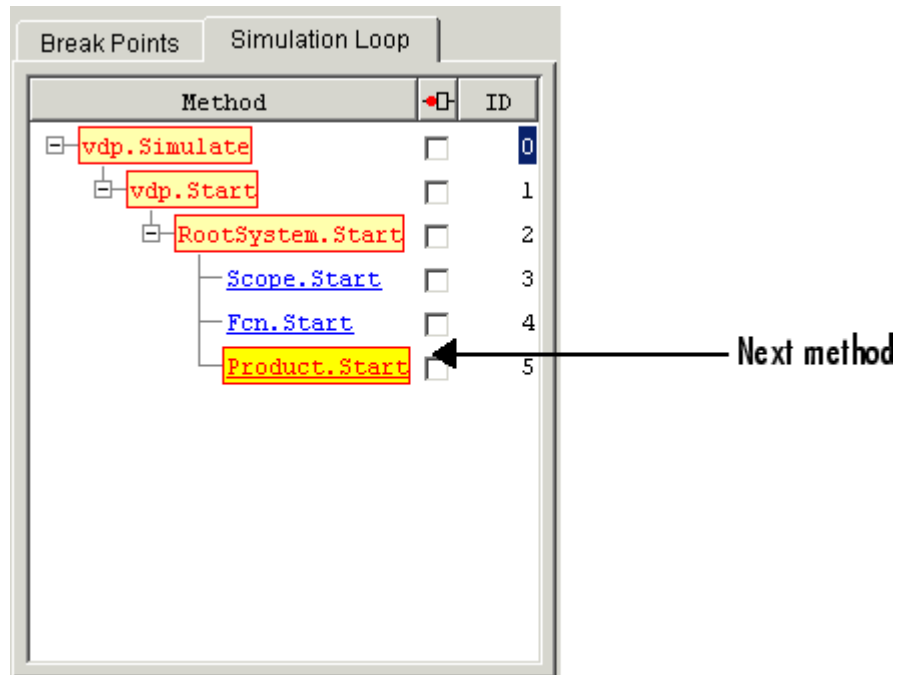
In command-line mode, you can use the `where` command to display the method call stack. If the next method is a block method, the debugger points the debug pointer at the block corresponding to the method (see "Debug Pointer" on page 18-25 for more information). If the block of the next method to be executed resides in a subsystem, the debugger opens the subsystem and points to the block in the subsystem block diagram.

Block Data Output

After executing a block method, the debugger prints any or all of the following block data in the debugger **Output** panel (in GUI mode) or, if in command line mode, the MATLAB® Command Window:

- `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 nth output.

- CSTATE = v

  where v is the value of the block's continuous state vector.

- DSTATE = v

  where v is the value of the block's discrete state vector.

The debugger also displays the current time, the ID and name of the next method to be executed, and the name of the block to which the method applies in the MATLAB Command Window. The following example illustrates typical debugger outputs after a step command.

```
                    Current time    Next method

%-------------------------------------------------------------%
[Tm = 2.009509145207664e-005 ] 0:2 Integrator.Outputs 'vdp/x2'
(sldebug @44):
Data of 0:2 Integrator block 'vdp/x2':
U1     = [-2]
Y1     = [-4.0190182904153282e-005]
CSTATE = [-4.0190182904153282e-005]
%-------------------------------------------------------------%
[Tm = 2.009509145207664e-005 ] 0:3 Outport.Outputs 'vdp/Out2'
```

Stepping Commands

Command-line mode provides the following commands for advancing a simulation incrementally:

This command...	Advances the simulation...
step [in into]	Into the next method, stopping at the first method in the next method or, if the next method does not contain any methods, at the end of the next method
step over	To the method that follows the next method, executing all methods invoked directly or indirectly by the next method

This command...	Advances the simulation...
step out	To the end of the current method, executing any remaining methods invoked by the current method
step top	To the first method of the next time step (i.e., the top of the simulation loop)
step blockmth	To the next block method to be executed, executing all intervening model- and system-level methods
next	Same as step over

Buttons in the debugger toolbar allow you to access these commands in GUI mode.
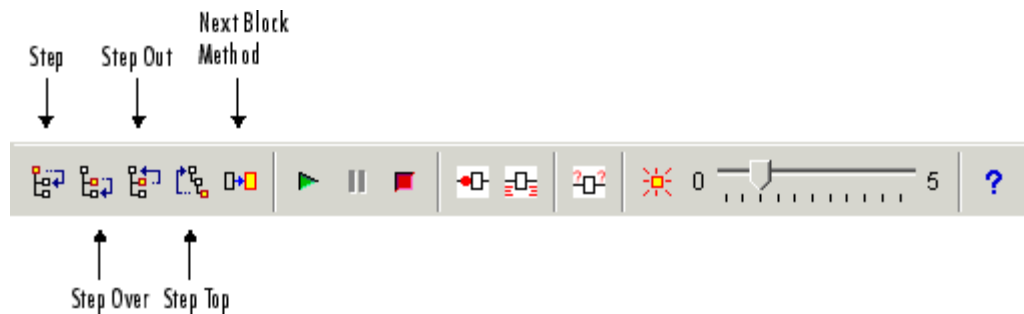
Clicking a button has the same effect as entering the corresponding command at the debugger command line.

Continuing a Simulation

In GUI mode, the **Stop** button turns red when the debugger suspends the simulation for any reason. To continue the simulation, click the **Start/Continue** button. In command-line mode, enter continue to continue the simulation. By default, the debugger runs the simulation to the next breakpoint (see "Setting Breakpoints" on page 18-27) or to the end of the simulation, whichever comes first.

Animation Mode

In *animation mode*, the **Start/Continue** button or the `continue` command advances the simulation method by method, pausing after each method, to the first method of the next major time step. While running the simulation in animation mode, the debugger uses its debug pointer (see "Debug Pointer" on page 18-25) to indicate on the block diagram which block method is being executed at each step. The moving pointer shows the simulation progress.

Note In animation mode, the debugger does not allow you to set breakpoints and ignores any breakpoints that you set when animating the simulation.

To enable animation when running the debugger in GUI mode, click the **Animation Mode** button on the debugger toolbar.
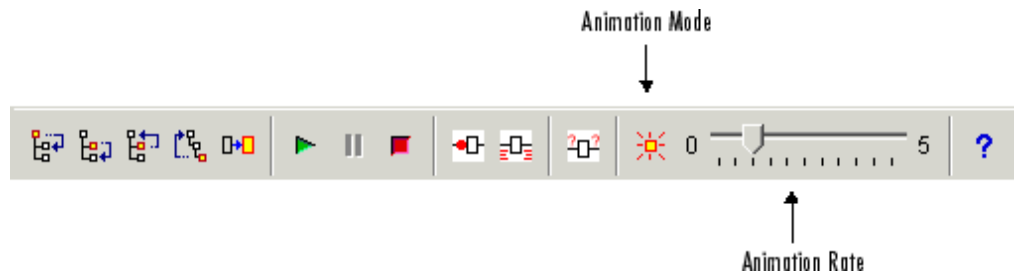
Use the slider on the debugger toolbar to increase or decrease the delay between method invocations, and so slow down or speed up the animation rate. To disable animation mode when running the debugger in GUI mode, toggle the **Animation Mode** button on the toolbar.

To enable animation when running the debugger in command-line mode, enter the `animate` command at the command line. The `animate` command has an optional delay parameter for you to specify the length of the pause between method invocations (1 second by default), and thereby accelerate or slow down the animation. For example, the command

```
animate 0.5
```

causes the animation to run at twice its default rate. To disable animation mode when running the debugger in command-line mode, at the command line enter:

```
animate stop
```

Running a Simulation Nonstop

The run command lets you run a simulation to the end of the simulation, skipping any intervening breakpoints. At the end of the simulation, the debugger returns you to the command line. To continue debugging a model, you must restart the debugger.

Note The GUI mode does not provide a graphical version of the run command. To run the simulation to the end, you must first clear all breakpoints and then click the **Start/Continue** button.

Debug Pointer

The debugger displays a debug pointer on the block diagram whenever it stops the simulation at a method.

Next method box   Block pointer   Method tile
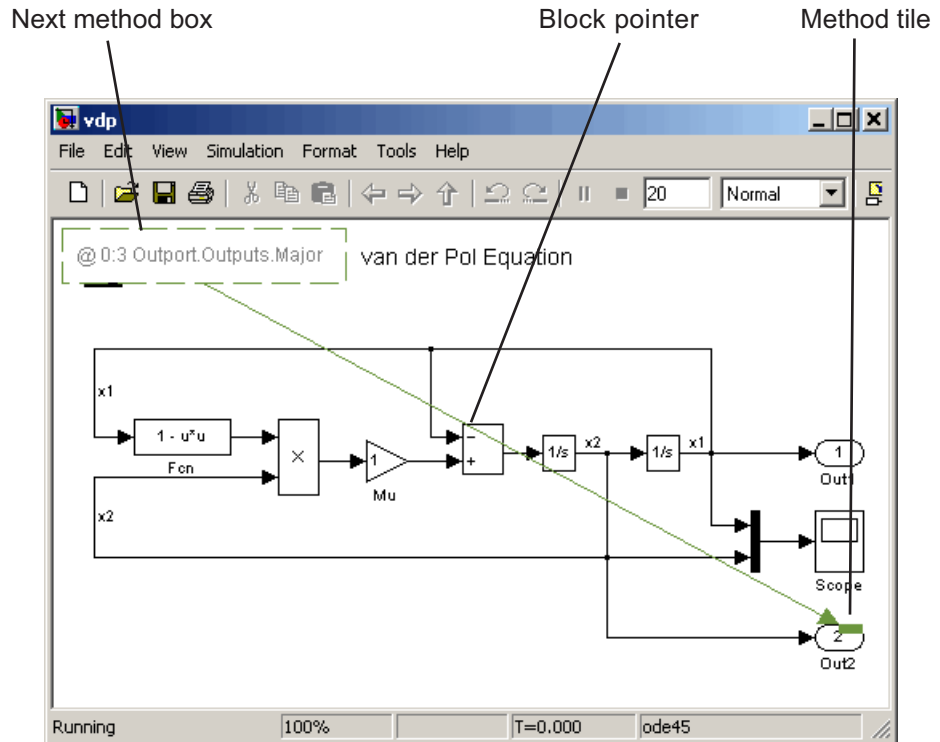
The debug pointer is an annotation that indicates the next method to be executed when simulation resumes. It consists of the following elements:

- Next method box
- Block pointer
- Method tile

18-25

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

In this section...

About Breakpoints

The debugger allows you to define stopping points called breakpoints in a simulation. You can then run a simulation from breakpoint to breakpoint, using the debugger `continue` command. The debugger lets you define two types of breakpoints: unconditional and conditional. An unconditional breakpoint occurs whenever a simulation reaches a method that you specified previously. A conditional breakpoint occurs when a condition that you specified in advance arises in the simulation.

Breakpoints are useful when you know that a problem occurs at a certain point in your program or when a certain condition occurs. By defining an appropriate breakpoint and running the simulation via the `continue` command, you can skip immediately to the point in the simulation where the problem occurs.

Setting Unconditional Breakpoints

You can set unconditional breakpoints from the:

- Debugger toolbar
- **Simulation Loop** pane
- MATLAB® product Command Window (command-line mode only)

Setting Breakpoints from the Debugger Toolbar

To set a breakpoint on a block's methods, select the block and then click the **Breakpoint** button on the debugger toolbar. If you set a break point on a block, the debugger stops at any method that the execution reaches in the block.
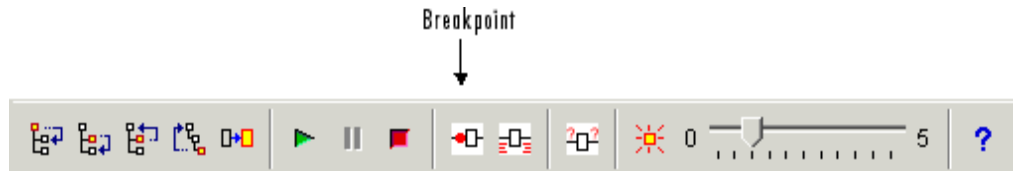
Breakpoint

The debugger displays the name of the selected block in the **Break/Display points** panel of the **Breakpoints** pane.
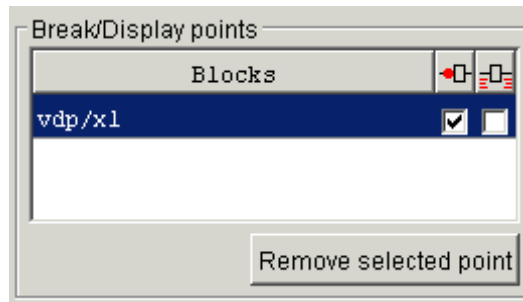
Note Clicking the **Breakpoint** button on the toolbar sets breakpoints on the invocations of a block's methods in major time steps.

You can temporarily disable the breakpoints on a block by deselecting the check box in the breakpoints column of the panel. To clear the breakpoints on a block and remove its entry from the panel,

 1 Select the entry.

 2 Click the **Remove selected point** button on the panel.

Note You cannot set a breakpoint on a virtual block. A virtual block is purely graphical: it indicates a grouping or relationship among a model's computational blocks. The debugger warns you if you try to set a breakpoint on a virtual block. You can get a listing of a model's nonvirtual blocks, using the `slist` command (see "Displaying a Model's Nonvirtual Blocks" on page 18-40).

Setting Breakpoints from the Simulation Loop Pane

To set a breakpoint at a particular invocation of a method displayed in the Simulation Loop pane, select the check box next to the method's name in the breakpoint column of the pane.
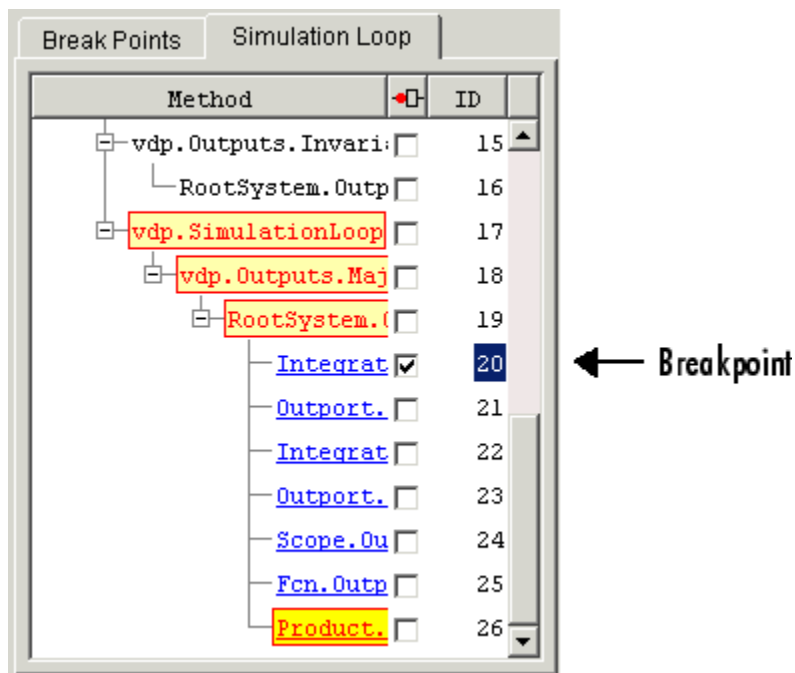
To clear the breakpoint, deselect the check box.

Setting Breakpoints from the Command Window

In command-line mode, use the break and bafter commands to set breakpoints before or after a specified method, respectively. Use the clear command to clear breakpoints.

Setting Conditional Breakpoints

You can use either the **Break on conditions** controls group on the debugger **Breakpoints** pane

```
┌─ Break on conditions ──────────────────┐
│  ☐ Zero crossings                       │
│  ☐ Step size limited by state           │
│  ☐ Solver Error                         │
│  ☐ NaN values                           │
│  Break at time : [            ]          │
└─────────────────────────────────────────┘
```
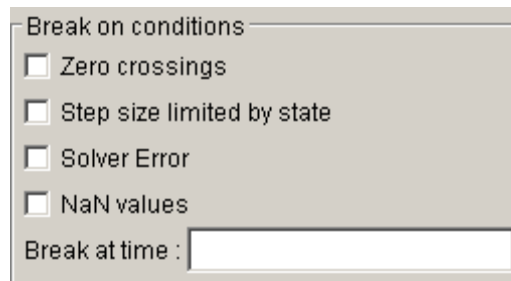
or the following commands (in command-line mode) to set conditional breakpoints.

This command...	Causes the 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 **Break at time** field (GUI mode) or enter the time using the `tbreak` command. This causes the debugger to stop the simulation at the `Outputs.Major` method of the model at the first time step that follows the specified time. For example, starting `vdp` in debug mode and entering the commands

```
tbreak 2
continue
```

causes the debugger to halt the simulation at the `vdp.Outputs.Major` method of time step `2.078` as indicated by the output of the `continue` command.

```
%------------------------------------------------------------
%
[TM = 2.078784598291364     ] vdp.Outputs.Major
(sldebug @18):
```

Breaking on Nonfinite Values

Selecting the debugger **NaN values** option or entering the `nanbreak` command causes the simulation to stop when a computed value is infinite or outside the range of values that is supported by the machine running the simulation. This option is useful for pinpointing computational errors in a model.

Breaking on Step-Size Limiting Steps

Selecting the **Step size limited by state** option or entering the `xbreak` command causes the debugger to stop the simulation when the model uses a variable-step solver and the solver encounters a state that limits the size of the steps that it can take. This command is useful in debugging models that appear to require an excessive number of simulation time steps to solve.

Breaking at Zero Crossings

Selecting the **Zero crossings** option or entering the `zcbreak` command causes the simulation to halt when a nonsampled zero crossing is detected in a model that includes blocks where zero crossings can arise. After halting, the ID, type, and name of the block in which the zero crossing was detected is

displayed. The block ID (s:b:p) consists of a system index s, block index b, and port index p separated by colons (see "Block ID" on page 18-11).

For example, setting a zero-crossing break at the start of execution of the zeroxing demo model,

```
>> sldebug zeroxing
%------------------------------------------------------------
%
[TM = 0                          ] zeroxing.Simulate
(sldebug @0): >> zcbreak
Break at zero crossing events               : enabled
```

and continuing the simulation

```
(sldebug @0): >> continue
```

results in a zero-crossing break at

```
2 Zero crossings detected at the following locations
  6  0:5:1  Saturate  'zeroxing/Saturation'
  7  0:5:2  Saturate  'zeroxing/Saturation'
ZeroCrossing Events detected. Interrupting model execution
%------------------------------------------------------------%
[Tm = 0.4                 ] zeroxing.zc.SearchLoop
(sldebug @55): >>
```

If a model does not include blocks capable of producing nonsampled zero crossings, the command prints a message advising you of this fact.

Breaking on Solver Errors

Selecting the debugger **Solver Errors** option or entering the ebreak command causes the simulation to stop if the solver detects a recoverable error in the model. If you do not set or disable this breakpoint, the solver recovers from the error and proceeds with the simulation without notifying you.

Displaying Information About the Simulation

In this section...

Displaying Block I/O

The debugger allows you to display block I/O by clicking the appropriate buttons on the debugger toolbar
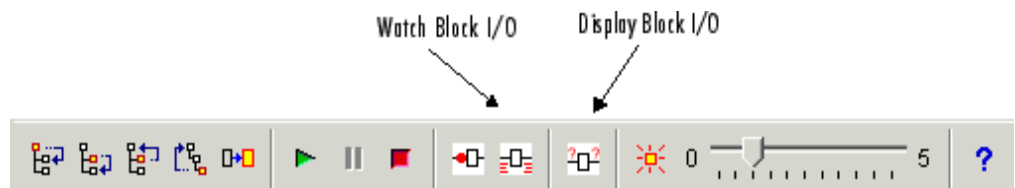
or by entering the appropriate debugger command.

This command...	Displays a Blocks I/O...
probe	Immediately
disp	At every breakpoint any time execution stops
trace	Whenever the block executes

Note The two debugger toolbar buttons, Watch Block I/O ( ) and Display Block I/O ( ) correspond, respectively, to `trace gcb` and `probe gcb`. The `probe` and `disp` commands do not have a one-to-one correspondence with the debugger toolbar buttons.

Displaying I/O of a Selected Block

To display the I/O of a block, select the block and click 🔲 in GUI mode or enter the probe command in command-line mode. In the following table, the probe gcb command has a corresponding toolbar button. The other commands do not.

Command	Description
probe	Enter or exit probe mode. In probe mode, the debugger displays the current inputs and outputs of any block that you select in the model's block diagram. Typing any command causes the debugger to exit probe mode.
probe gcb	Display I/O of selected block. Same as 🔲.
probe s:b	Print the I/O of the block specified by system number s and block number b.

The debugger prints the current inputs, outputs, and states of the selected block in the debugger **Outputs** pane (GUI mode) or the Command Window of the MATLAB® product.

The probe command is useful when you need to examine the I/O of a block whose I/O is not otherwise displayed. For example, suppose you are using the step command to run a model method by method. Each time you step the simulation, the debugger displays the inputs and outputs of the current block. The probe command lets you examine the I/O of other blocks as well.

Displaying Block I/O Automatically at Breakpoints

The disp command causes the debugger to display a specified block's inputs and outputs whenever it halts the simulation. You can specify a block either by entering its block index or by selecting it in the block diagram and entering gcb as the disp command argument. You can remove any block from the debugger list of display points, using the undisp command. For example, to remove block 0:0, either select the block in the model diagram and enter undisp gcb or simply enter undisp 0:0.

Note Automatic display of block I/O at breakpoints is not available in the debugger GUI mode.

The disp command is useful when you need to monitor the I/O of a specific block or set of blocks as you step through a simulation. Using the disp command, you can specify the blocks you want to monitor and the debugger will then redisplay the I/O of those blocks on every step. Note that the debugger always displays the I/O of the current block when you step through a model block by block, using the step command. You do not need to use the disp command if you are interested in watching only the I/O of the current block.

Watching Block I/O

To watch a block, select the block and click in the debugger toolbar or enter the trace command. In GUI mode, if a breakpoint exists on the block, you can set a watch on it as well by selecting the check box for the block in the watch column of the **Break/Display points** pane. In command-line mode, you can also specify the block by specifying its block index in the trace command. You can remove a block from the debugger list of trace points using the untrace command.

The debugger displays a watched block's I/O whenever the block executes. Watching a block allows you obtain a complete record of the block's I/O without having to stop the simulation.

Displaying Algebraic Loop Information

The atrace command causes the debugger to display information about a model's algebraic loops (see "Algebraic Loops" on page 2-31) each time they are solved. The command takes a single argument that specifies the amount of information to display.

This command...	Displays for each algebraic loop...
atrace 0	No information

This command...	Displays for each algebraic loop...
atrace 1	The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error
atrace 2	Same as level 1
atrace 3	Level 2 plus the Jacobian matrix used to solve the loop
atrace 4	Level 3 plus intermediate solutions of the loop variable

Displaying System States

The states debug command lists the current values of the system's states in the MATLAB product Command Window. For example, the following sequence of commands shows the states of the bouncing ball demo (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 models differential equations that can slow down simulation performance. Executing this command causes the debugger to display solver-related information at the command line of the MATLAB product when you run or step through a simulation. The information includes the sizes of the steps taken by the solver, the estimated integration error resulting from the step size, whether a step size succeeded (i.e., met the accuracy requirements that the model specifies), the times at which solver resets occur, etc. If you are concerned

about the time required to simulate your model, this information can help you to decide whether the solver you have chosen is the culprit and hence whether choosing another solver might shorten the time required to solve the model.

Displaying Information About the Model

In this section...

Displaying a Models Sorted Lists

In GUI mode, the debugger **Sorted List** pane displays lists of blocks for a models root system and each nonvirtual subsystem. Each list lists the blocks that the subsystems contains sorted according to their computational dependencies, alphabetical order, and other block sorting rules. In command-line mode, you can use the slist command to display a model's sorted lists.

```
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
   0:0    'vdp/Integrator1' (Integrator)
   0:1    'vdp/Out1' (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 use them to determine the block IDs of the models blocks. Some debugger commands accept block IDs as arguments.

Identifying Blocks in Algebraic Loops

If a block belongs to an algebraic list, the slist command displays an algebraic loop identifier in the entry for the block in the sorted list. The identifier has the form

```
algId=s#n
```

where s is the index of the subsystem containing the algebraic loop and n is the index of the algebraic loop in the subsystem. For example, the following entry for an Integrator block indicates that it participates in the first algebraic loop at the root level of the model.

```
0:1 'test/ss/I1' (Integrator, tid=0) [algId=0#1, discontinuity]
```

You can use the debugger ashow command to highlight the blocks and lines that make up an algebraic loop. See "Displaying Algebraic Loops" on page 18-41 for more information.

Displaying a Block

To determine the block in a models diagram that corresponds to a particular index, enter bshow s:b at the command prompt, where s:b is the block index. The bshow command opens the system containing the block (if necessary) and selects the block in the systems window.

Displaying a Models Nonvirtual Systems

The systems command displays a list of the nonvirtual systems in the model that you are debugging. For example, the clutch demo (clutch) contains the following systems:

```
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. That is, subsystems that the model diagram represents as Subsystem blocks but that are solved as part of a parent system. are not listed. In Simulink® models, the root system and triggered or enabled subsystems are true systems. All other subsystems are virtual (that is, graphical) and do not appear in the listing from the systems command.

Displaying a Model's Nonvirtual Blocks

The slist command displays a list of the nonvirtual blocks in a model. The listing groups the blocks by system. For example, the following sequence of commands produces a list of the nonvirtual blocks in the Van der Pol (vdp) demo model.

```
sldebug vdp
[Tm=0                    ] **Start** of system 'vdp' outputs
(sldebug @0:0 'vdp/Integrator1'): slist
---- Sorted list for 'vdp' [12 blocks, 9 nonvirtual blocks,
directFeed=0]
  0:0    'vdp/Integrator1' (Integrator)
  0:1    'vdp/Out1' (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. That is, blocks that indicate relationships between or groupings among computational blocks.

Displaying Blocks with Potential Zero Crossings

The zclist command displays a list of blocks in which nonsampled zero crossings can occur during a simulation. For example, zclist displays the following list for the clutch sample model:

```
(sldebug @0:0 'clutch/Clutch Pedal'): zclist
  2:3    'clutch/Unlocked/Sign' (Signum)
  0:4    'clutch/Lockup Detection/Velocities Match' (HitCross)
  0:10   'clutch/Lockup Detection/Required Friction
            for Lockup/Abs' (Abs)
  0:11   'clutch/Lockup Detection/Required Friction for
            Lockup/ Relational Operator' (RelationalOperator)
```

```
0:18    'clutch/Break Apart Detection/Abs' (Abs)
0:20    'clutch/Break Apart Detection/Relational Operator'
         (RelationalOperator)
0:24    'clutch/Unlocked' (SubSystem)
0:27    'clutch/Locked' (SubSystem)
```

Displaying Algebraic Loops

The ashow command highlights a specified algebraic loop or the algebraic loop that contains a specified block. To highlight a specified algebraic loop, enter ashow s#n, where s is the index of the system (see "Identifying Blocks in Algebraic Loops" on page 18-38) 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.
```

19

Accelerating Models

What Is Acceleration?

Acceleration is a mode of operation in the Simulink® product that you can use to speed up the execution of your model. The Simulink software includes two modes of acceleration: *Accelerator* mode and the *Rapid Accelerator* mode. Both modes replace the normal interpreted code with compiled target code. Using compiled code speeds up simulation of many models, especially those where run time is long compared to the time associated with compilation and checking to see if the target is up to date.

The Accelerator mode works with any model that does not include "Algebraic Loops" on page 2-31, but performance decreases if a model contains blocks that do not support acceleration. The Accelerator mode supports the Simulink debugger and profiler. These tools assist in debugging and determining relative performance of various parts of your model. For more information, see "Using the Accelerator Mode with the Simulink® Debugger" on page 19-31 and "Capturing Performance Data" on page 19-33.

The Rapid Accelerator mode works with only those models containing blocks that support code generation of a standalone executable. For this reason, Rapid Accelerator mode does not support the debugger or profiler. However, this mode generally results in faster execution than the Accelerator mode. When used with dual-core processors, the Rapid Accelerator mode runs Simulink and the MATLAB® technical computing environment from one core while the rapid accelerator target runs as a separate process on a second core.

For more information about the performance characteristics of the Accelerator and Rapid Accelerator modes, and how to measure the difference in performance, see "Comparing Performance" on page 19-24.

How the Acceleration Modes Work

In this section...

Overview

The Accelerator and Rapid Accelerator modes use portions of the Real-Time Workshop® product to create an executable. These modes replace the interpreted code normally used in Simulink® simulations, shortening model run time.

Although the acceleration modes use some Real-Time Workshop code generation technology, you do not need the Real-Time Workshop software installed to accelerate your model.

Note The code generated by the Accelerator and Rapid Accelerator modes is suitable only for speeding the simulation of your model. You must use the Real-Time Workshop product if you want to generate code for other purposes.

Normal Mode

In Normal mode, the MATLAB® technical computing environment is the foundation on which the Simulink software is built.Simulink controls the solver and model methods used during simulation. Model methods include such things as computation of model outputs. Normal mode runs in one process.
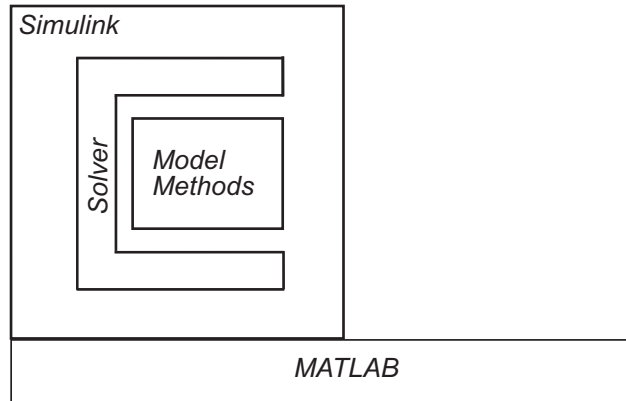
One Process

Accelerator Mode

The Accelerator mode generates and links code into a C-MEX S-function. Simulink uses this *acceleration target code* to perform the simulation, and the code remains available for use in later simulations.

Simulink checks that the acceleration target code is up to date before reusing it. As explained in "Code Regeneration in Accelerated Models" on page 19-7, the target code regenerates if it is not up to date.

In Accelerator mode, the model methods are separate from the Simulink software and are part of the Acceleration target code. A C-MEX S-function API communicates with the Simulink software, and a MEX API communicates with MATLAB. The target code executes in the same process as MATLAB and Simulink.

One Process
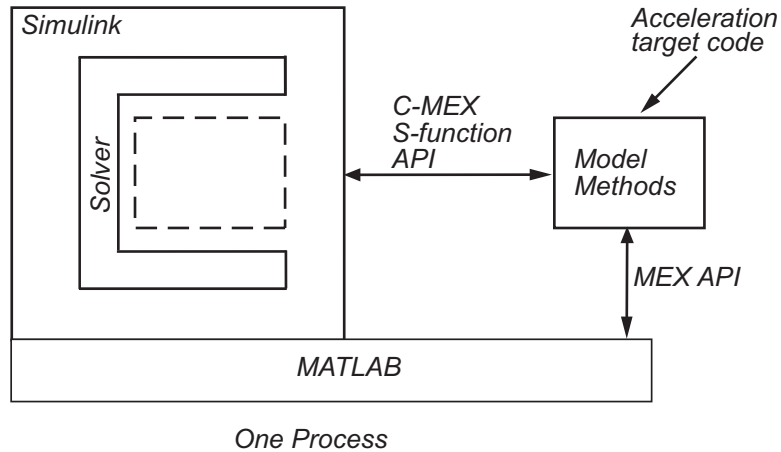
Rapid Accelerator Mode

The Rapid Accelerator mode creates a *Rapid Accelerator standalone executable* from your model. This executable includes the solver and model methods, but it resides outside of MATLAB and Simulink. It uses External mode (see "External Mode") to communicate with Simulink.

Rapid Accelerator
Standalone executable

Simulink

External
Mode

Solver

Model
Methods

MATLAB

One Process   *Second Process*
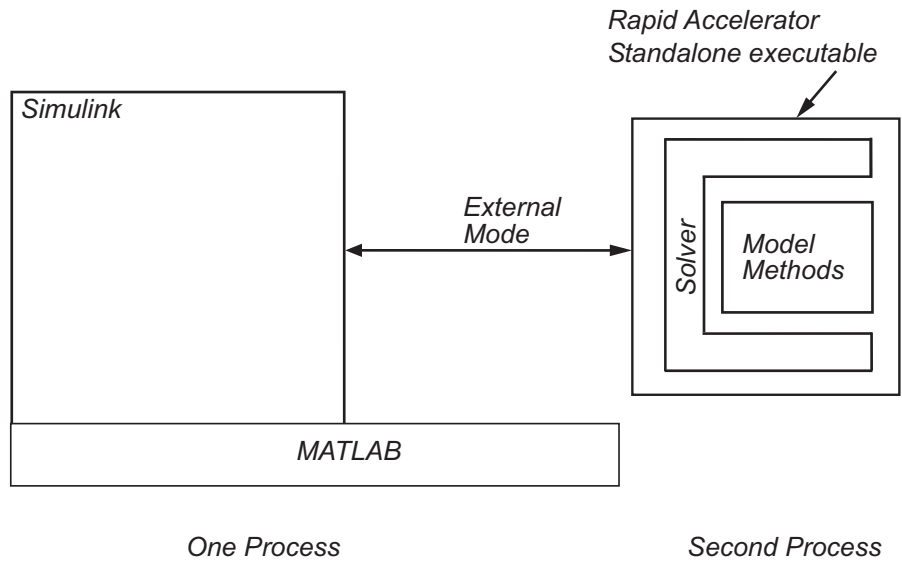
MATLAB and Simulink run in one process, and if a second processing core is available, the standalone executable runs there.

Code Regeneration in Accelerated Models

In this section...
"Structural Changes That Cause Rebuilds" on page 19-7
"Determining If the Simulation Will Rebuild" on page 19-7

Structural Changes That Cause Rebuilds

Changing the structure of your model causes the Rapid Accelerator mode to regenerate the standalone executable, and for the Accelerator mode to regenerate the target code and update (overwrite) the existing MEX-file.

Examples of model structure changes that result in a rebuild include:

- Changing the solver type, for example from `Variable-step` to `Fixed-step`
- Adding or deleting blocks or connections between blocks
- Changing the values of nontunable block parameters, for example, the **Initial seed** parameter of the Random Number block
- Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
- Changing the number of states in the model
- Selecting a different function in the Trigonometric Function block
- Changing signs used in a Sum block
- Adding a Target Language Compiler (TLC) file to inline an S-function
- Changing the `sim` command `output` argument when using the Rapid Accelerator mode
- Changing solver parameters such as stop time, or `rel tol` when using the Rapid Accelerator mode

Determining If the Simulation Will Rebuild

The Accelerator and Rapid Accelerator modes use a checksum to determine if the model has changed, indicating that the code should be regenerated. The

checksum is an array of four integers computed using an MD5 checksum algorithm based on attributes of the model and the blocks it contains.

Use the `Simulink.BlockDiagram.getChecksum` command to obtain the checksum for your model. For example:

```
cs1 = Simulink.BlockDiagram.getChecksum('myModel');
```

Obtain a second checksum after you have altered your model. The code regenerates if the new checksum does not match the previous checksum. You can use the information in the checksum to determine why the simulation target rebuilt. For a detailed explanation of this procedure, see the demo model `slAccelDemoWhyRebuild`.

Choosing a Simulation Mode

In this section...
"Tradeoffs" on page 19-9
"Comparing Modes" on page 19-10
"Decision Tree" on page 19-11

Tradeoffs

In general, you must trade off simulation speed against flexibility when choosing either Accelerator mode or Rapid Accelerator mode instead of Normal mode.
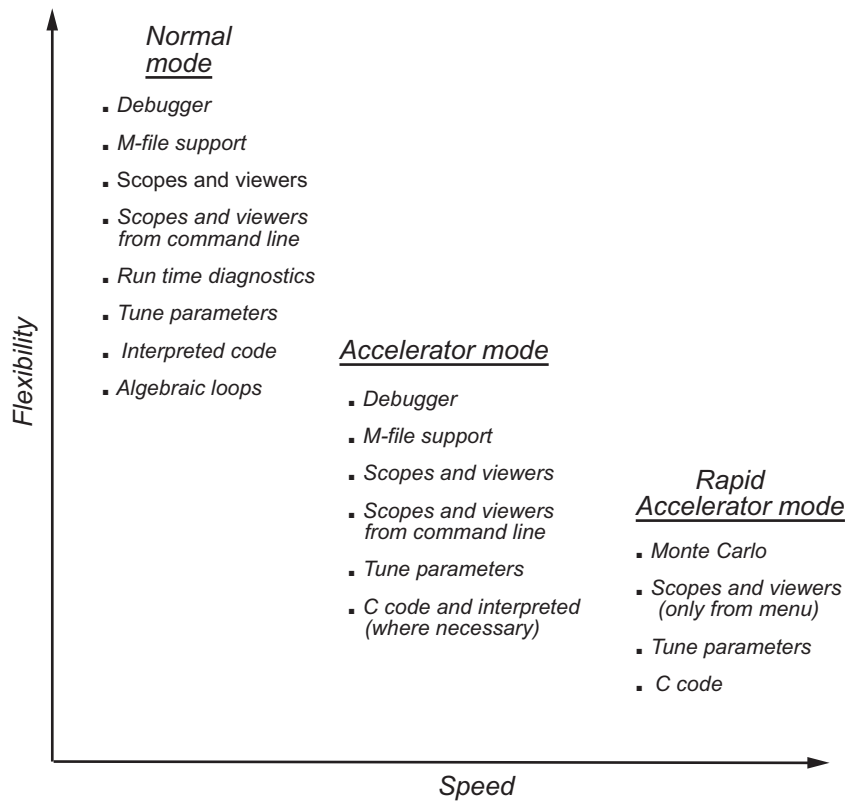
Normal mode offers the greatest flexibility for making model adjustments and displaying results, but it runs the slowest. Rapid Accelerator mode runs the fastest, but this mode does not support the debugger or profiler, and works only with those models for which C code is available for all of the blocks in the model. Accelerator mode lies between these two in performance and in interaction with your model.

Comparing Modes

The following table compares the characteristics of Normal mode, Accelerator mode, and Rapid Accelerator mode.

If you want to...	Then use this mode...		
	Normal	**Accelerator**	**Rapid Accelerator**
Performance			
Run your model in a separate address space			✓
Efficiently run batch and Monte Carlo simulations			✓
Model Adjustment			
Change model parameters such as solver type, stop time without rebuilding	✓	✓	
Change block tunable parameters such as gain	✓	✓	✓
Model Requirement			
Accelerate your model even if C code is not used for all blocks		✓	
Support M-file S-function blocks	✓	✓	
Permit algebraic loops in your model	✓		
Have your model work with the debugger or profiler	✓	✓	
Have your model include C++ code	✓	✓	

If you want to...	Then use this mode...		
	Normal	**Accelerator**	**Rapid Accelerator**
Data Display			
Use scopes and signal viewers	✓	✓	See "Using Scopes and Viewers with Rapid Accelerator Mode" on page 19-16
Use scopes and signal viewers when running your model from the command line	✓	✓	

Note Scopes and viewers do not update if you run your model from the command line in Rapid Accelerator mode.

Decision Tree

The following decision tree can help you select between Normal mode, Accelerator mode, or Rapid Accelerator mode.

See "Comparing Performance" on page 19-24 to understand how effective the accelerator modes will be in improving the performance of your model.
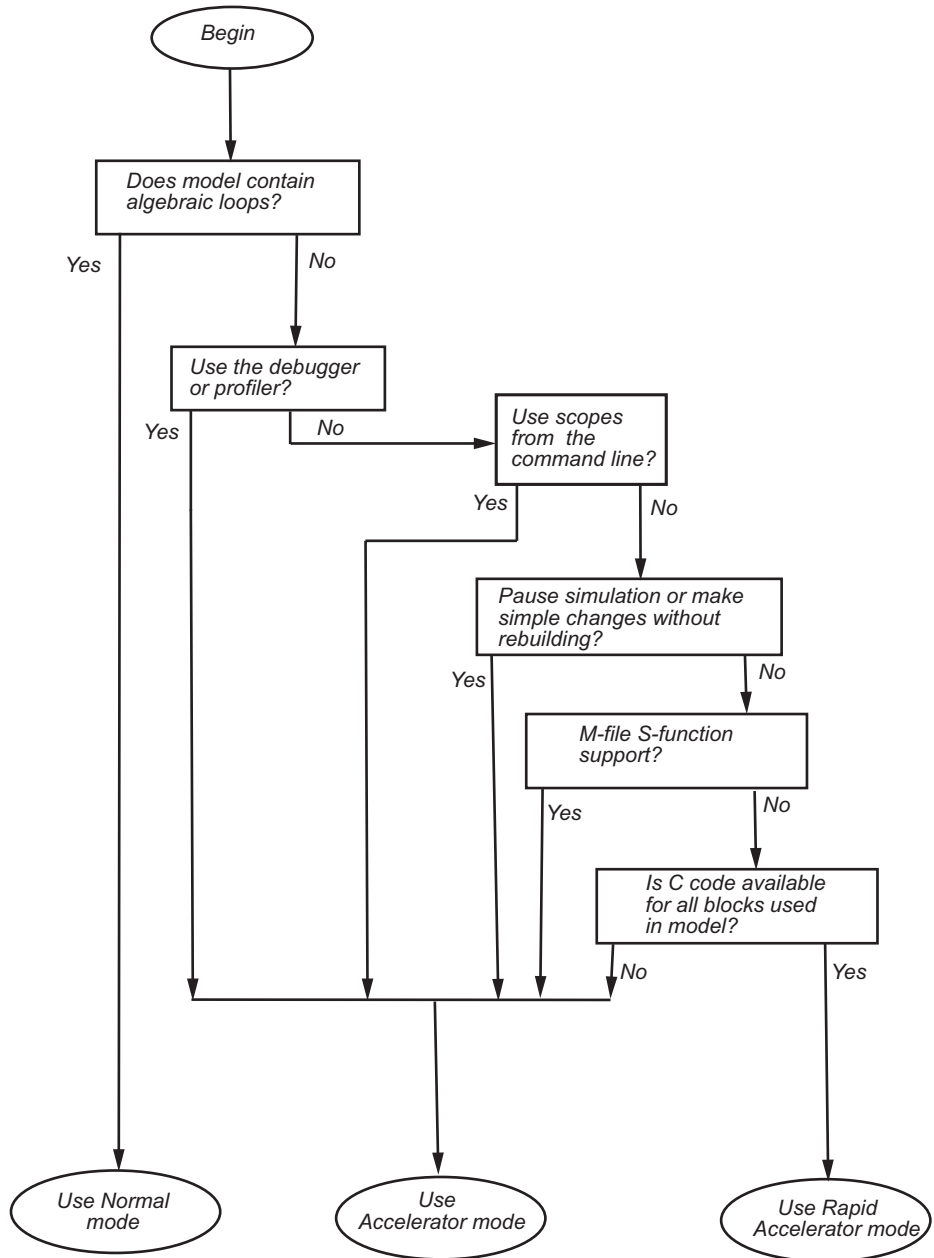
Designing Your Model for Effective Acceleration

In this section...
"Selecting Blocks for Accelerator Mode" on page 19-13
"Selecting Blocks for Rapid Accelerator Mode" on page 19-14
"Controlling S-Function Execution" on page 19-14
"Accelerator and Rapid Accelerator Mode Fixed-Point Considerations" on page 19-15
"Using Scopes and Viewers with Rapid Accelerator Mode" on page 19-16
"Factors Inhibiting Acceleration" on page 19-16

Selecting Blocks for Accelerator Mode

For some blocks, the Accelerator mode uses normal interpreted code rather than C code. The Accelerator mode runs even if these blocks are in your model, but with degraded performance.

These Simulink® blocks use interpreted code:

- Display
- Embedded MATLAB Function
- From File
- From Workspace
- Inport (root level only)
- MATLAB Fcn
- Outport (root level only)
- Scope
- To File
- To Workspace
- Transport Delay
- Variable Transport Delay

• XY Graph

Note In some instances, Normal mode output might not precisely match the output from Accelerator mode because of slight differences in the numerical precision between the interpreted and compiled versions of a model.

Selecting Blocks for Rapid Accelerator Mode

Blocks that do not support code generation (such as SimEvents®), or blocks that generate code only for a specific target (such as vxWorks), cannot be simulated in Rapid Accelerator mode.

Additionally, Rapid Accelerator mode does not work if your model contains any of the following blocks:

• MATLAB Fcn

• Device driver S-functions, such as blocks from the XPC Target product, or those targeting Freescale™ MPC555

Note In some instances, Normal mode output might not precisely match the output from Rapid Accelerator mode because of slight differences in the numerical precision between the interpreted and compiled versions of a model.

Controlling S-Function Execution

Inlining S-functions using the Target Language Compiler increases performance with the Accelerator mode by eliminating unnecessary calls to the Simulink application program interface (API). By default, however, the Accelerator mode ignores an inlining TLC file for an S-function, even though the file exists. The Rapid Accelerator mode always uses the TLC file if one is available.

A device driver S-Function block written to access specific hardware registers on an I/O board is one example of why this behavior was chosen as the default. Because the Simulink software runs on the host system rather than

the target, it cannot access the targets I/O registers and so would fail when attempting to do so.

To direct the Accelerator mode to use the TLC file instead of the S-function MEX-file, specify SS_OPTION_USE_TLC_WITH_ACCELERATOR in the mdlInitializeSizes function of the S-function, as in this example:

```
static void mdlInitializeSizes(SimStruct *S)
{
/* Code deleted */
ssSetOptions(S, SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}
```

Accelerator and Rapid Accelerator Mode Fixed-Point Considerations

- Fixed-point signals or vectors greater than 32 bits are not supported by the Accelerator or Rapid Accelerator mode.

- The Accelerator mode and Rapid Accelerator mode store integers as compactly as possible:

Word Size (Bits)	Data Type Used
1 to 8	int8 uint8
9 to 16	int16 uint16
17 to 32	int32 uint32

- Simulink® Fixed Point™ does not collect data in the Accelerator or Rapid Accelerator mode.

- The fixed-point options min, max, and overflow are not supported by either the Accelerator or Rapid Accelerator modes.

Using Scopes and Viewers with Rapid Accelerator Mode

Running the simulation from the command line or the menu determines behavior of scopes and viewers in Rapid Accelerator mode.

Scope or Viewer Type	Simulation Run from Menu	Simulation Run from Command Line
Simulink Scope blocks	Same support as Normal mode	• Logging is supported • Scope window is not updated
Simulink signal viewer scopes	Graphics are updated, but logging is not supported	Not supported
Other signal viewer scopes	Support limited to that available in External mode	Not supported
Signal logging	Not supported	Not supported
Multirate signal viewers	Not supported	Not supported

Rapid Accelerator mode does not support multirate signal viewers such as the Signal Processing Blockset™ spectrum scope or the Communications Blockset™ scatterplot, signal trajectory, or eye diagram scopes.

Note Although scopes and viewers do not update when you run Rapid Accelerator mode from the command line, they do update when you use the menu. "Running Acceleration Mode from the User Interface" on page 19-20 shows how to run Rapid Accelerator mode from the menu. "Interacting with the Acceleration Modes Programmatically" on page 19-28 shows how to run the simulation from the command line.

Factors Inhibiting Acceleration

You cannot use the Accelerator or Rapid Accelerator mode if your model:

- Passes array parameters to M-file S-functions that are not numeric, logical, or character arrays, are sparse arrays, or that have more than two dimensions
- Contains algebraic loops
- Uses Fcn blocks containing trigonometric functions having complex inputs

Rapid Accelerator mode does not support targets written in C++.

For Rapid Accelerator mode, model parameters must be one of these data types:

- `boolean`
- `uint8` or `int8`
- `uint16` or `int16`
- `uint32` or `int32`
- `single` or `double`

Reserved Keywords

Certain words are reserved for use by the Real-Time Workshop® code language and by Accelerator mode and Rapid Accelerator mode. These keywords must not appear as function or variable names on a subsystem, or as exported global signal names. Using the reserved keywords results in the Simulink software reporting an error, and the model cannot be compiled or run.

The keywords reserved for the Real-Time Workshop product are listed in "Configuring Generated Identifiers". Additional keywords that apply only to the Accelerator and Rapid accelerator modes are:

muDoubleScalarAbs	muDoubleScalarCos	muDoubleScalarMod
muDoubleScalarAcos	muDoubleScalarCosh	muDoubleScalarPower
muDoubleScalarAcosh	muDoubleScalarExp	muDoubleScalarRound
muDoubleScalarAsin	muDoubleScalarFloor	muDoubleScalarSign
muDoubleScalarAsinh	muDoubleScalarHypot	muDoubleScalarSin

muDoubleScalarAtan,	muDoubleScalarLog	muDoubleScalarSinh
muDoubleScalarAtan2	muDoubleScalarLog10	muDoubleScalarSqrt
muDoubleScalarAtanh	muDoubleScalarMax	muDoubleScalarTan
muDoubleScalarCeil	muDoubleScalarMin	muDoubleScalarTanh

Performing Acceleration

In this section...

Customizing the Build Process

Compiler optimizations are off by default. This results in faster build times. To optimize acceleration of your model, set the compiler optimization level from the Optimization pane in the Configuration Parameters dialog box.
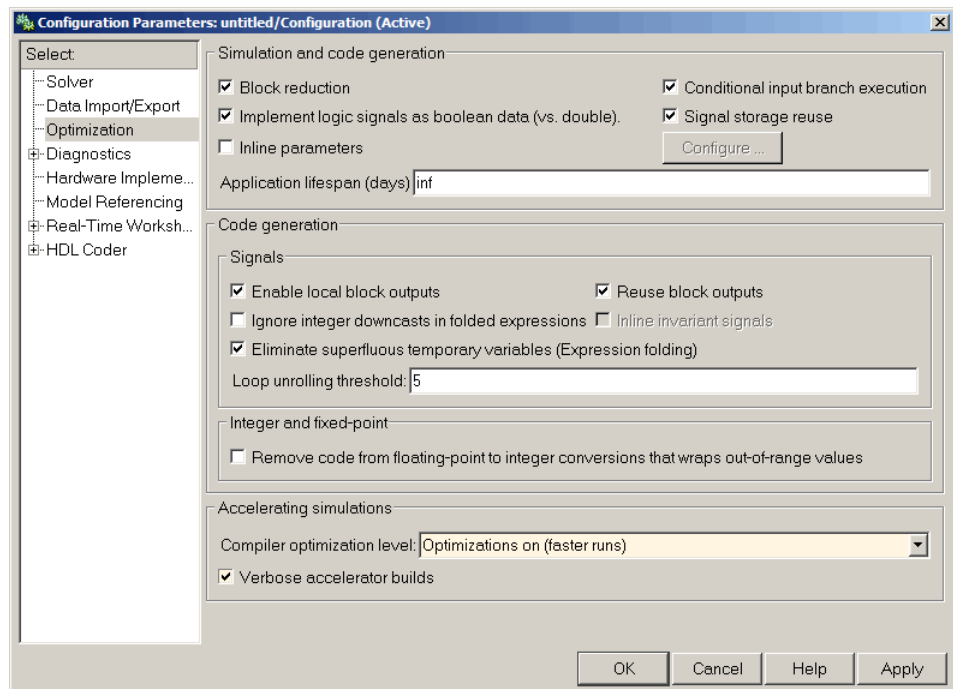
Select `Optimizations on (faster runs)` when you want to create optimized code. Code generation takes longer with this option, but the model runs faster.

Select **Verbose accelerator builds** to display progress information using code generation, and to see the compiler options in use.

Running Acceleration Mode from the User Interface

To accelerate a model, first open it, and then from the **Simulation** menu, select either **Accelerator** or **Rapid Accelerator**. Then start the simulation.

The following example shows how to accelerate the already opened f14 model using the Accelerator mode:

1 From the **Simulation** menu, select **Accelerator**.

Alternatively, you can select Accelerator from the model editor's toolbar.
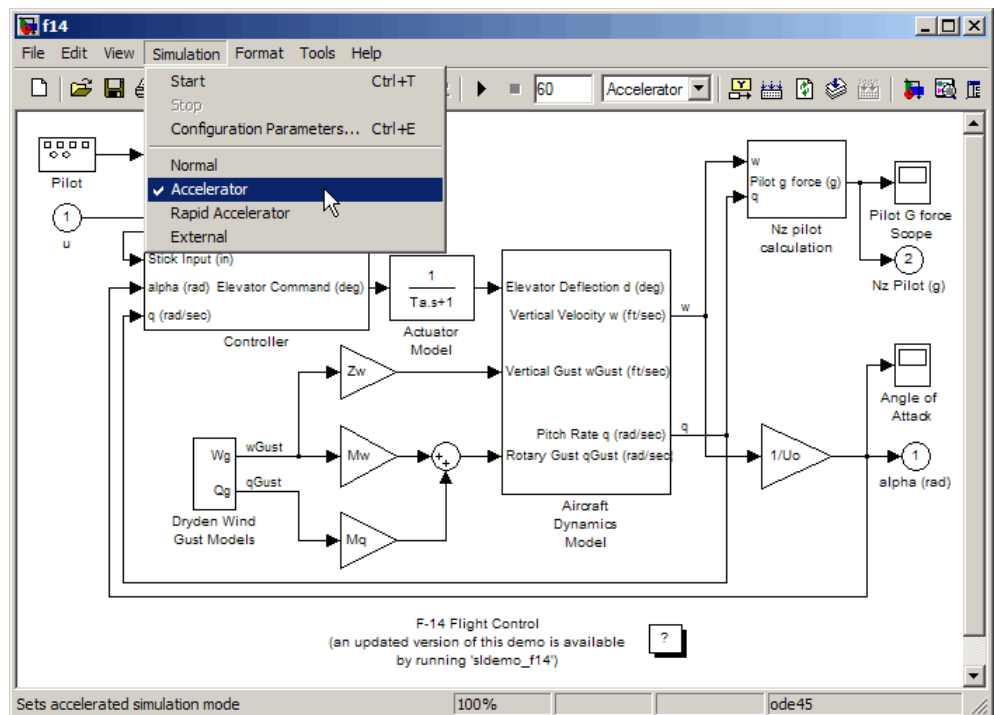
2 From the **Simulation** menu, select **Start**.

The Accelerator and Rapid Accelerator modes first check to see if code was previously compiled for your model. If code was created previously, the Accelerator or Rapid Accelerator modes run the model. If code was not previously built, they first generate and compile the C code, and then run the model.

For reasons why these modes rebuild your model, see "Code Regeneration in Accelerated Models" on page 19-7.

The Accelerator mode places the generated code in a subdirectory called *modelname_accel_rtw* (for example, f14_accel_rtw), and places a compiled MEX-file in the current working directory.

The Rapid Accelerator mode places the generated code in a subdirectory called *modelname_raccel_rtw* (for example, f14_raccel_rtw).

Note The warnings that blocks generate during simulation (such as divide-by-zero and integer overflow) are not displayed when your model runs in Accelerator or Rapid Accelerator modes.

Making Run-Time Changes

A feature of the Accelerator and Rapid Accelerator modes is that simple adjustments (such as changing the value of a Gain or Constant block) can be made to the model while the simulation is still running. More complex changes (for example, changing from a sin to tan function) are not allowed during run time.

The Simulink® software issues a warning if you attempt to make a change that is not permitted. The absence of a warning indicates that the change was accepted. The warning does not stop the current simulation, and the simulation continues with the previous values. If you wish to alter the model in ways that are not permitted during run time, you must first stop the simulation, make the change, and then restart the simulation.

In general, simple model changes are more likely to result in code regeneration when in Rapid Accelerator mode than when in Accelerator mode. For instance, changing the stop time in Rapid Accelerator mode causes code to regenerate, but does not cause Accelerator mode to regenerate code.

Improving Acceleration Mode Performance

In this section...
"Techniques" on page 19-22
"C Compilers" on page 19-23

Techniques

To get the best performance when accelerating your models:

- Verify that the Configuration Parameters dialog box settings are as follows:

On this pane...	Set...	To...
Solver Diagnostics	**Solver data inconsistency**	none
Data Validity Diagnostics	**Array bounds exceeded**	none
Optimization	**Signal storage reuse**	selected

- Disable Stateflow® debugging and animation.

- Inline user-written S-functions (these are TLC files that direct the Real-Time Workshop® software to create C code for the S-function). See "Controlling S-Function Execution" on page 19-14 for a discussion on how the Accelerator mode and Rapid Accelerator mode work with inlined S-functions.

  For information on how to inline S-functions, consult "Writing S-Functions for Real-Time Workshop Code Generation".

- When logging large amounts of data (for instance, when using the Workspace I/O, To Workspace, To File, or Scope blocks), use decimation or limit the output to display only the last part of the simulation.

- Customize the code generation process to improve simulation speed. See "Customizing the Build Process" on page 19-19 for details.

C Compilers

On computers running the Microsoft® Windows® operating system, the
Accelerator and Rapid Accelerator modes use the default 32-bit C compiler
supplied by The MathWorks to compile your model. If you have a C compiler
installed on your PC, you can configure the `mex` command to use it instead.
You might choose to do this if your C compiler produces highly optimized
code since this would further improve acceleration, or if you wish to use a
64-bit compiler.

Note Technical Note 1601 (which is available at `www.mathworks.com`) lists
the 32- and 64-bit C compilers that are compatible with the MATLAB®
software for all supported computing platforms.

Comparing Performance

In this section...
"Performance of the Simulation Modes" on page 19-24
"Measuring Performance" on page 19-26

Performance of the Simulation Modes

The Accelerator and Rapid Accelerator modes give the best speed improvement compared to Normal mode when simulation execution time exceeds the time required for code generation. For this reason, the Accelerator and Rapid Accelerator modes generally perform better than Normal mode when simulation execution times are several minutes or more. However, models with a significant number of Stateflow® or Embedded MATLAB™ blocks might show only a small speed improvement over Normal mode because in Normal mode these blocks also simulate through code generation.

Including tunable parameters in your model can also increase the simulation time.

The figure shows in general terms the performance of a hypothetical model simulated in Normal, Accelerator, and Rapid Accelerator modes.

Simulation Stop Time or Number of Major Time Steps
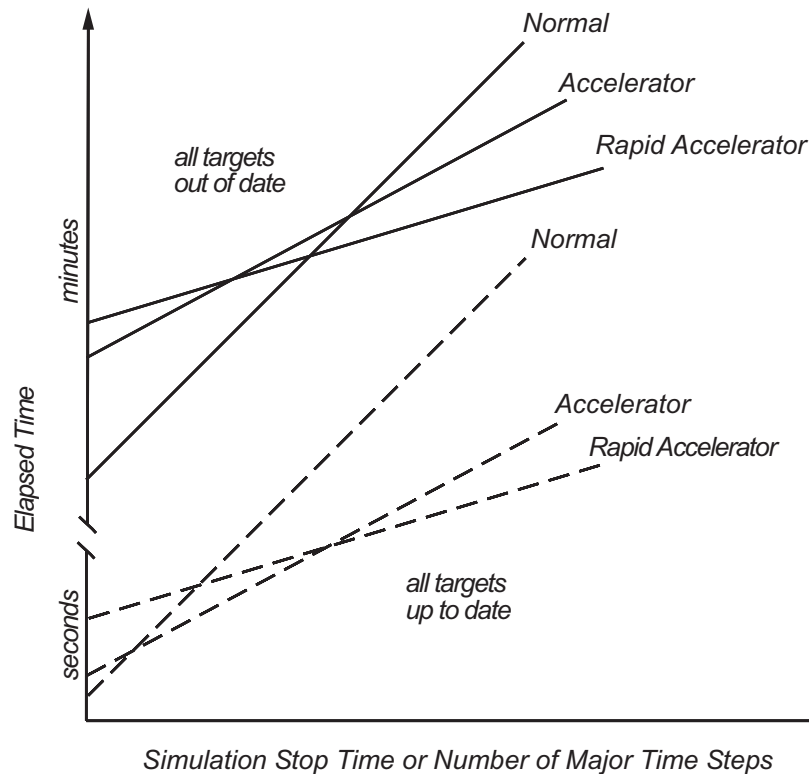
Performance When the Target Must Be Rebuilt

The solid lines in the figure show performance when the target code must be rebuilt ("all targets out of date"). For this hypothetical model, the time scale is on the order of minutes, but it could be longer for more complex models.

As generalized in the figure, the time required to compile the model in Normal mode is less than the time required to build either the Accelerator target or Rapid Accelerator executable. It is evident from the figure that for small simulation stop times Normal mode results in quicker overall simulation times than either Accelerator mode or Rapid Accelerator mode.

The crossover point where Accelerator mode or Rapid Accelerator mode result in faster execution times depends on the complexity and content of your model. For instance, those models running in Accelerator mode containing large

numbers of blocks using interpreted code (see "Selecting Blocks for Accelerator Mode" on page 19-13) might not run much faster than they would in Normal mode unless the simulation stop time is very large. Similarly, models with a large Stateflow or Embedded MATLAB content might not show much speed improvement over Normal mode unless the simulation stop times are long.

For illustration purposes, the graphic represents a model with a large number of Stateflow or Embedded MATLAB blocks. The curve labeled "Normal" would have much smaller initial elapsed time than shown if the model did not contain these blocks.

Performance When the Targets Are Up to Date

As shown by the broken lines in the figure ("all targets up to date") the time for the Simulink® software to determine if the Accelerator target or the Rapid Accelerator executable are up to date is significantly less than the time required to generate code ("all targets out of date"). You can take advantage of this characteristic when you wish to test various design tradeoffs.

For instance, you can generate the Accelerator mode target code once and use it to simulate your model with a series of gain settings. This is an especially efficient way to use the Accelerator or Rapid Accelerator modes because this type of change does not result in the target code being regenerated. This means the target code is generated the first time the model runs, but on subsequent runs the Simulink code spends only the time necessary to verify that the target is up to date. This process is much faster than generating code, so subsequent runs can be significantly faster than the initial run.

Because checking the targets is quicker than code generation, the crossover point is smaller when the target is up to date than when code must be generated. This means subsequent runs of your model might simulate faster in Accelerator or Rapid Accelerator mode when compared to Normal mode, even for short stop times.

Measuring Performance

You can use the `tic`, `toc`, and `sim` commands to compare Accelerator mode or Rapid Accelerator mode execution times to Normal mode.

1 Use `load_system` to load your model into memory without opening a window.

2 From the **Simulation** menu, select **Normal**.

3 Use the `tic`, `toc`, and `sim` commands at the command line prompt to measure how long the model takes to simulate in Normal mode:

```
tic,[t,x,y]=sim('myModel',10000);toc
```

`tic` and `toc` work together to record and return the elapsed time and display a message such as the following:

```
Elapsed time is 17.789364 seconds.
```

4 Select either **Accelerator** or **Rapid Accelerator** from the **Simulation** menu, and build an executable for the model by clicking the **Start** button. The acceleration modes use this executable in subsequent simulations as long as the model remains structurally unchanged. "Code Regeneration in Accelerated Models" on page 19-7 discusses the things that cause your model to rebuild.

5 Rerun the compiled model at the command prompt:

```
tic,[t,x,y]=sim('myModel',10000);toc
```

6 The elapsed time displayed shows the run time for the accelerated model. For example:

```
Elapsed time is 12.419914 seconds.
```

The difference in elapsed times (5.369450 seconds in this example) shows the improvement obtained by accelerating your model.

Interacting with the Acceleration Modes Programmatically

Why Interact Programmatically?

You can build an accelerated model, select the simulation mode, and run the simulation from the command prompt or from M-files. With this flexibility, you can create Accelerator mode MEX-files in batch mode, allowing you to build the C code and executable before running the simulations. When you use the Accelerator mode interactively at a later time, it will not be necessary to generate or compile MEX-files at the start of the accelerated simulations.

Building Accelerator Mode MEX-files

With the `accelbuild` command, you can build the Accelerator mode MEX-file without actually simulating the model. For example, to build an Accelerator mode simulation of `myModel`:

```
accelbuild myModel
```

Controlling Simulation

You can control the simulation mode from the command line prompt by using the set_param command:

```
set_param('modelname','simulationmode','mode')
```

The simulation mode can be `normal`, `accelerator`, `rapid`, or `external`.

For example, to simulate your model with the Accelerator mode, you would use:

```
set_param('myModel','simulationmode','accelerator')
```

You can use gcs ("get current system") to set parameters for the currently active model (that is, the active model window) rather than *modelname* if you do not wish to explicitly specify the model name.

For example, to prepare to simulate the currently opened system with the rapid accelerator mode, you would use:

```
set_param(gcs,'simulationmode','rapid')
```

Simulating Your Model

Use set_param to configure the model parameters (such as acceleration mode and stop time), and the sim command to start the simulation:

```
sim('modelname')
```

You can substitute gcs for *modelname* if you do not want to explicitly specify the model name.

Unless target code has already been generated, the sim command first builds the executable and then runs the simulation. However, if the target code has already been generated and no significant changes have been made to the model (see "Code Regeneration in Accelerated Models" on page 19-7 for a description), the sim command executes the generated code without regenerating the code. This process lets you run your model after making simple changes without having to wait for the model to rebuild.

Simulation Example

The following sequence shows how to programmatically simulate myModel in Rapid Accelerator mode for 10,000 seconds.

First open myModel, and then type the following in the Command Window:

```
set_param('myModel','simulationmode','rapid');
set_param('myModel','stoptime','10000');
sim('myModel');
```

Use the `sim` command again to resimulate after making a change to your model. If the change is minor (adjusting the gain of a gain block, for instance), the simulation runs without regenerating code.

Customizing the Acceleration Build Process

You can programmatically control the Accelerator mode and Rapid Accelerator mode build process and the amount of information displayed during the build process. See "Customizing the Build Process" on page 19-19 for details on why doing so might be advantageous.

Controlling the Build Process

Use the `SimCompilerOptimization` parameter to control the acceleration build process. The permitted values are `on` or `off`. The default is `off`.

Enter the following at the command prompt to turn on compiler optimization:

```
set_param('myModel', 'SimCompilerOptimization', 'on')
```

Controlling Information During Code Generation

Use the `AccelVerboseBuild` parameter to display progress information during code generation. The permitted values are `on` or `off`. The default is `off`.

Enter the following at the command prompt to turn on verbose build:

```
set_param('myModel', 'AccelVerboseBuild', 'on')
```

Using the Accelerator Mode with the Simulink® Debugger

Advantages of Using Accelerator Mode with the Debugger

The Accelerator mode can shorten the length of your debugging sessions if you have large and complex models. For example, you can use the Accelerator mode to simulate a large model and quickly reach a distant break point.

For more information on the debugger, see Chapter 18, "Simulink® Debugger".

Note You cannot use the Rapid Accelerator mode with the debugger.

How to Run the Debugger

To run your model in the Accelerator mode with the debugger:

1 From the **Simulation** menu, select **Accelerator**.

2 At the command prompt, enter:

    sldebug *modelname*

3 At the debugger prompt, set a time break:

    tbreak 10000
    continue

4 Once you reach the breakpoint, use the debugger command emode (execution mode) to toggle between Accelerator and Normal mode.

When to Switch Back to Normal Mode

You must switch to Normal mode to step through the simulation by blocks, and when you want to use the following debug commands:

- `trace`

- `break`

- `zcbreak`

- `nanbreak`

Capturing Performance Data

In this section...
"What Is the Profiler?" on page 19-33
"How the Profiler Works" on page 19-33
"Enabling the Profiler" on page 19-35
"Simulation Profile" on page 19-36

What Is the Profiler?

The profiler captures data while your model runs and identifies the parts of your model requiring the most time to simulate. You use this information to decide where to focus your model optimization efforts.

Note You cannot use the Rapid Accelerator mode with the profiler.

Performance data showing the time spent executing each function in your model is placed in a report called the *simulation profile*.

How the Profiler Works

The following pseudocode summarizes the execution model on which the profiler is based.

```
Sim()
 ModelInitialize().
 ModelExecute()
  for t = tStart to tEnd
  Output()
  Update()
  Integrate()
   Compute states from derivs by repeatedly calling:
    MinorOutput()
    MinorDeriv()
   Locate any zero crossings by repeatedly calling:
    MinorOutput()
```

```
       MinorZeroCrossings()
    EndIntegrate
    Set time t = tNew.
  EndModelExecute
  ModelTerminate
EndSim
```

According to this conceptual model, your model is executed by invoking the following functions zero, one, or more times, depending on the function and the model.

Function	Purpose	Level
sim	Simulate the model. This top-level function invokes the other functions required to simulate the model. The time spent in this function is the total time required to simulate the model.	System
ModelInitialize	Set up the model for simulation.	System
ModelExecute	Execute the model by invoking the output, update, integrate, etc., functions for each block at each time step from the start to the end of simulation.	System
Output	Compute the outputs of a block at the current time step.	Block
Update	Update a block's state at the current time step.	Block
Integrate	Compute a block's continuous states by integrating the state derivatives at the current time step.	Block
MinorOutput	Compute a block's output at a minor time step.	Block

Function	Purpose	Level
MinorDeriv	Compute a block's state derivatives at a minor time step.	Block
MinorZeroCrossings	Compute a block's zero-crossing values at a minor time step.	Block
ModelTerminate	Free memory and perform any other end-of-simulation cleanup.	System
Nonvirtual Subsystem	Compute the output of a nonvirtual subsystem at the current time step by invoking the output, update, integrate, etc., functions for each block that it contains. The time spent in this function is the time required to execute the nonvirtual subsystem.	Block

The profiler measures the time required to execute each invocation of these functions and generates a report at the end of the model that describes how much time was spent in each function.

Enabling the Profiler

To profile a model, open the model and select **Profiler** from the **Tools** menu. Then start the simulation. When the simulation finishes, the Simulink® code generates and displays the simulation profile for the model in the Help browser.

Note Enabling the profiler on a parent model does not enable profiling for referenced models. Profiling must be enabled separately for each submodel. Profiling occurs only if the submodel executes in Normal mode. See Chapter 5, "Referencing a Model" for more information.

Simulation Profile

The Simulink software stores the simulation profile in the MATLAB® software working directory.

The report has two sections: a summary and a detailed report.

Summary Section

The summary file displays the following performance totals.

Item	Description
Total Recorded Time	Total time required to simulate the model
Number of Block Methods	Total number of invocations of block-level functions (e.g., `Output()`)
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. The link works only if you are viewing the profile in the Help browser.

Detailed Profile Section

This section contains detailed profiles for each function invoked to simulate the model. Each detailed profile contains all the information shown in the summary profile for the function. In addition, the detailed profile displays the function (parent function) that invoked the profiled function and the functions (child functions) invoked by the profiled function. Clicking the name of the parent or a child function takes you to the detailed profile for that function.

20

Customizing the Simulink® User Interface

Adding Items to Model Editor Menus

About Adding Items to the Model Editor Menus

You can add commands and submenus to the end of the Simulink® model editor menus. Adding an item to the end of a Model Editor menu entails performing the following tasks:

- For each item, create a function, called a *schema function*, that defines the item (see "Defining Menu Items" on page 20-4).

- Register the menu customizations with the Simulink customization manager at startup, e.g., in an sl_customization.m file on the MATLAB® path (see "Registering Menu Customizations" on page 20-9).

- Create callback functions that implement the commands triggered by the items that you add to the menus.

Note You can use the procedures described in the following sections to customize the Stateflow® Chart Editor's menu.

Code Example

The following sl_customization.m file adds four items to the editor's **Tools** menu.

```
function sl_customization(cm)

  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyMenuItems);
end

%% Define the custom menu function.
function schemaFcns = getMyMenuItems(callbackInfo)
  schemaFcns = {@getItem1,...
      @getItem2,...
      {@getItem3,3}... %% Pass 3 as user data to getItem3.
      @getItem4};
end

%% Define the schema function for first menu item.
function schema = getItem1(callbackInfo)
  schema = sl_action_schema;
  schema.label = 'Item One';
  schema.userdata = 'item one';
  schema.callback = @myCallback1;
end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata ' was called']);
end

function schema = getItem2(callbackInfo)
  % Make a submenu label 'Item Two' with
  % the menu item above three times.
  schema = sl_container_schema;
  schema.label = 'Item Two';
 schema.childrenFcns = {@getItem1, @getItem1, @getItem1};
end

function schema = getItem3(callbackInfo)
  % Create a menu item whose label is
  % 'Item Three: 3', with the 3 being passed
  % from getMyItems above.

  schema = sl_action_schema;
```

```
      schema.label = ['Item Three: ' num2str(callbackInfo.userdata)];
  end

  function myToggleCallback(callbackInfo)
      if strcmp(get_param(gcs, 'ScreenColor'), 'red') == 0
          set_param(gcs, 'ScreenColor', 'red');
      else
          set_param(gcs, 'ScreenColor', 'white');
      end
  end

  %% Define the schema function for a toggle menu item.
  function schema = getItem4(callbackInfo)
    schema = sl_toggle_schema;
    schema.label = 'Red Screen';
    if strcmp(get_param(gcs, 'ScreenColor'), 'red') == 1
      schema.checked = 'checked';
    else
      schema.checked = 'unchecked';
    end
    schema.callback = @myToggleCallback;
  end
```

Defining Menu Items

You define a menu item by creating a function that returns an object, called
a *schema* object, that specifies the information needed to create the menu
item. The menu item that you define may trigger a custom action or display a
custom submenu. See the following sections for more information.

- "Defining Menu Items That Trigger Custom Commands" on page 20-4
- "Defining Custom Submenus" on page 20-7

Defining Menu Items That Trigger Custom Commands

To define an item that triggers a custom command, your schema function
must accept a callback info object (see "Callback Info Object" on page 20-10)
and create and return an action schema object (see "Action Schema Object"
on page 20-5) that specifies the item's label and a function, called a *callback*,

to be invoked when the user selects the item. For example, the following schema function defines a menu item that displays a message when selected by the user.

```
function schema = getItem1(callbackInfo)

  %% Create an instance of an action schema.
  schema = sl_action_schema;

  %% Specify the menu item's label.
  schema.label = 'My Item 1';

  %% Specify the menu item's callback function.
  schema.callback = @myCallback1;

end

function myCallback1(callbackInfo)
  disp(['Callback for item ' callbackInfo.userdata
        ' was called']);
end
```

Action Schema Object. This object specifies information about menu items that trigger commands that you define, including the label that appears on the menu item and the function to be invoked when the user selects the menu item. Use the function sl_action_schema to create instances of this object in your schema functions. Its properties include

- tag

  Optional string that identifies this action, for example, so that it can be referenced by a filter function.

- label

  String specifying the label that appears on a menu item that triggers this action.

- state

  String that specifies the state of this action. Valid values are 'Enabled' (the default), 'Disabled', and 'Hidden'.

- statustip

  String specifying text to appear in the editor's status bar when the user selects the menu item that triggers this action.

- userdata

  Data that you specify. May be of any type.

- accelerator

  String specifying a keyboard shortcut that a user may use to trigger this action. The string must be of the form 'Ctrl+*K*', where *K* is the shortcut key. For example, 'Ctrl+T' specifies that the user may invoke this action by holding down the **Ctrl** key and pressing the **T** key.

- callback

  String specifying a MATLAB expression to be evaluated or a handle to a function to be invoked when a user selects the menu item that triggers this action. This function must accept one argument: a callback info object.

Toggle Schema Object. This object specifies information about a menu item that toggles some object on or off. Use the function sl_toggle_schema to create instances of this object in your schema functions. Its properties include

- tag

  Optional string that identifies this toggle action, for example, so that it can be referenced by a filter function.

- label

  String specifying the label that appears on a menu item that triggers this toggle action.

- checked

  This property must be set to one of the following values:

  'checked' Menu item displays a check mark.

  'uncheckedMenu item does not display a check mark.

- state

String that specifies the state of this toggle action. Valid values are `'Enabled'` (the default), `'Disabled'`, and `'Hidden'`.

- `statustip`

  String specifying text to appear in the editor's status bar when the user selects the menu item that triggers this toggle action.

- `userdata`

  Data that you specify. May be of any type.

- `accelerator`

  String specifying a keyboard shortcut that a user may use to trigger this action. The string must be of the form `'Ctrl+K'`, where *K* is the shortcut key. For example, `'Ctrl+T'` specifies that the user may invoke this action by holding down the **Ctrl** key and pressing the **T** key.

- `callback`

  String specifying a MATLAB expression to be evaluated or a handle to a function to be invoked when a user selects the menu item that triggers this action. This function must accept one argument: a callback info object.

Defining Custom Submenus

To define a submenu, create a schema function that accepts a callback info object and returns a container schema object (see "Container Schema Object" on page 20-7) that specifies the schemas that define the items on the submenu. For example, the following schema function defines a submenu that contains three instances of the menu item defined in the example in "Defining Menu Items That Trigger Custom Commands" on page 20-4.

```
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 20-4)

  - 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 20-10) passed to the schema function

  - `'separator'`, which causes a separator to appear between the item defined by the preceding entry in the cell array and the item defined in the following entry. This case is ignored for this entry, e.g., `'SEPARATOR'` and `'Separator'` are valid entries. A separator is also suppresses if it would appear at the beginning or end of the submenu and combines separators that would appear successively (e.g., as a result of an item being hidden) into a single separator.

  For example, the following cell array specifies two submenu entries:

  ```
  {@getItem1, 'separator', {@getItem2, 1}}
  ```

  In this example, a 1 is passed to getItem2 via a callback info object.

- generateFcn

  Pointer to a function that returns a cell array defining the contents of the submenu. The cell array must have the same format as that specified for the container schema objects `childrenFcns` property.

Note The `generateFcn` property takes precedence over the `childrenFcns` property. If you set both, the `childrenFcns` property is ignored and the cell array returned by the `generateFcn` is used to create the submenu.

Registering Menu Customizations

You must register custom items to be included on a Simulink menu with the customization manager. Use the `sl_customization.m` file for a Simulink installation (see "Registering Customizations" on page 20-21) 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 20-21). Each invocation should pass the tag of the menu (see "About Menu Tags" on page 20-10) to be customized and a custom menu function that specifies the items to be added to the menu (see "Creating the Custom Menu Function" on page 20-9) . For example, the following sl_customization function adds custom items to the Simulink Tools menu.

```
function sl_customization(cm)
  %% Register custom menu function.
  cm.addCustomMenuFcn('Simulink:ToolsMenu', @getMyItems);
```

Creating the Custom Menu Function

The custom menu function returns a list of schema functions that define custom items that you want to appear on the model editor menus (see "Defining Menu Items" on page 20-4).

Your custom menu function should accept a callback info object (see "Callback Info Object" on page 20-10) and return a cell array that lists the schema functions. Each element of the cell array can be either a handle to a schema function or a two-element cell array whose first element is a handle to a

schema function and whose second element is user-defined data to be passed to the schema function. For example, the following custom menu function returns a cell array that lists three schema functions.

```
function schemas = getMyItems(callbackInfo)
  schemas = {@getItem1, ...
             @getItem2, ...
               {@getItem3,3} }; % Pass 3 as userdata to getItem3.
end
```

Callback Info Object

Instances of these objects are passed to menu customization functions. Properties of these objects include

- uiObject

  Handle to the owner of the menu for which this is the callback. The owner can be the Simulink editor or the Stateflow editor.

- model

  Handle to the model being displayed in the editor window.

- userdata

  User data. The value of this field can be any type of data.

Debugging Custom Menu Callbacks

On systems using the Microsoft® Windows® operating system, selecting a custom menu item whose callback contains a breakpoint can cause the mouse to become unresponsive or the menu to remain open and on top of other windows. To fix these problems, use the M-file debugger's keyboard commands to continue execution of the callback.

About Menu Tags

A menu tag is a string that identifies a Simulink Model Editor or Stateflow Chart Editor menu bar or menu. You need to know a menu's tag to add

custom items to it (see "Registering Menu Customizations" on page 20-9). You can configure the editor to display all (see "Displaying Menu Tags" on page 20-11) but the following tags:

Tag	Usage
Simulink:MenuBar	Add menus to Model Editor's menu bar.
Simulink:ContextMenu	Add items to the end of Model Editor's context menu.
Simulink:PreContextMenu	Add items to the beginning of Model Editor's context menu.
Stateflow:MenuBar	Add menus to Chart Editor's menu bar.
Stateflow:ContextMenu	Add items to the end of Chart Editor's context menu.
Stateflow:PreContextMenu	Add items to the beginning of Chart Editor's context menu.

Displaying Menu Tags

You can configure the Simulink software (and the Stateflow product) to display the tag for a menu item next to the item's label, allowing you to determine at a glance the tag for a menu. To configure the editor to display menu tags, set the customization manager's showWidgetIdAsToolTip property to true, e.g., by entering the following commands at the command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip=true;
```

The tag of each menu item appears next to the item's label on the menu:
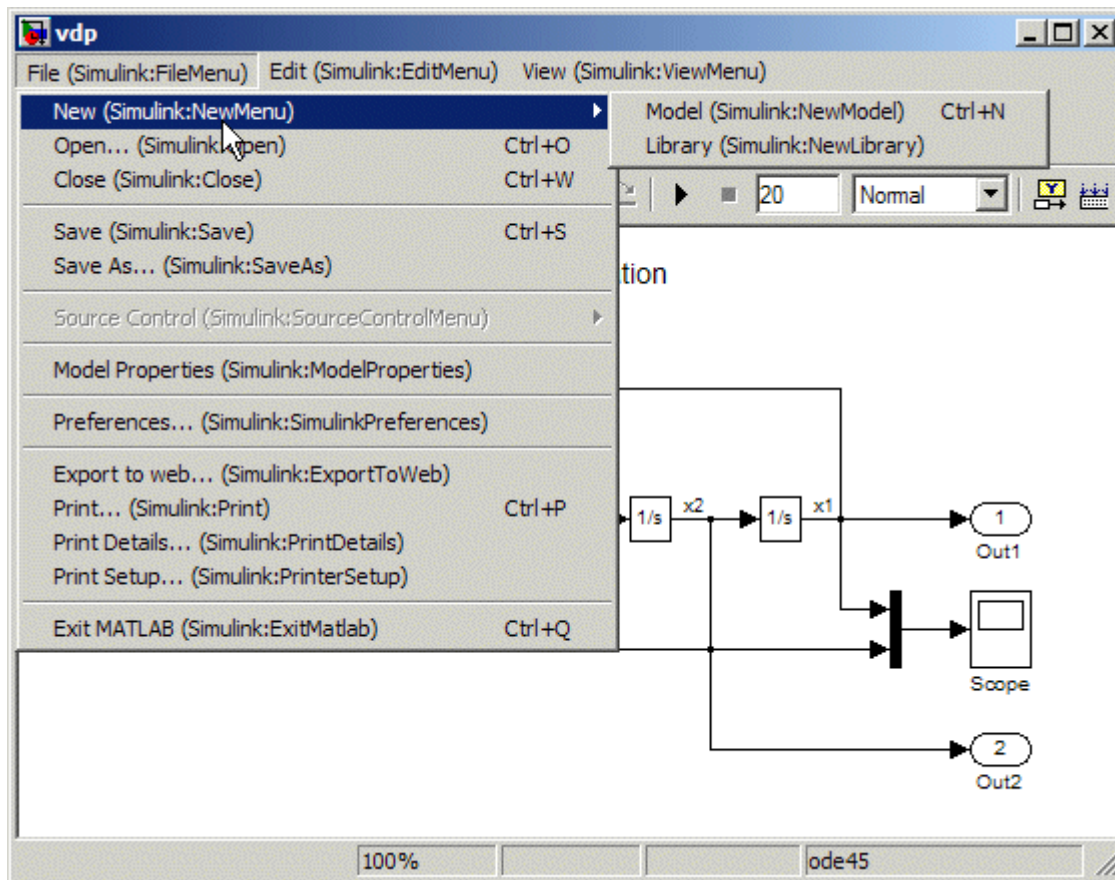
To turn off tag display, enter the following command at the command line:

```
cm.showWidgetIdAsToolTip=false;
```

Note Some menu items may not work while menu tag display is enabled. To ensure that all items work, turn off menu tag display before using the menus.

Disabling and Hiding Model Editor Menu Items

In this section...
"About Disabling and Hiding Model Editor Menu Items" on page 20-13
"Example: Disabling the New Model Command on the Simulink® Editor's File Menu" on page 20-13
"Creating a Filter Function" on page 20-13
"Registering a Filter Function" on page 20-14

About Disabling and Hiding Model Editor Menu Items

You can disable or hide items that appear on the Simulink® model editor menus. To disable or hide a menu item, you must:

- Create a filter function that disables or hides the menu item (see "Creating a Filter Function" on page 20-13).

- Register the filter function with the customization manager (see "Registering a Filter Function" on page 20-14).

For more information on Model Editor menu items, see:

Example: Disabling the New Model Command on the Simulink® Editor's File Menu

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end

function state = myFilter(callbackInfo)
  state = 'Disabled';
end
```

Creating a Filter Function

Your filter function must accept a callback info object and return a string that specifies the state that you want to assign to the menu item. Valid states are

- `'Hidden'`

- `'Disabled'`

- `'Enabled'`

Your filter function may have to compete with other filter functions and with Simulink itself to assign a state to an item. Who succeeds depends on the strength of the state that each assigns to the item. `'Hidden'` is the strongest state. If any filter function or Simulink assigns `'Hidden'` to the item, it is hidden. `'Enabled'` is the weakest state. For an item to be enabled, all filter functions and the Simulink or Stateflow® products must assign `'Enabled'`to the item. The `'Disabled'` state is of middling strength. It overrides `'Enabled'` but is itself overridden by `'Hidden'`. For example, if any filter function or Simulink or Stateflow assigns `'Disabled'` to a menu item and none assigns `'Hidden'` to the item, the item is disabled.

Note The Simulink software does not allow you to filter some menu items, for example, the **Exit MATLAB** item on the Simulink **File** menu. An error message is displayed if you attempt to filter a menu item that you are not allowed to filter.

Registering a Filter Function

Use the customization manager's addCustomFilterFcn method to register a filter function. The addCustomFilterFcn method takes two arguments: a tag that identifies the menu or menu item to be filtered (see "Displaying Menu Tags" on page 20-11) and a pointer to the filter function itself. For example, the following code registers a filter function for the **New Model** item on the Simulink **File** menu.

```
function sl_customization(cm)
  cm.addCustomFilterFcn('Simulink:NewModel',@myFilter);
end
```

Disabling and Hiding Dialog Box Controls

About Disabling and Hiding Controls

The Simulink® product includes a customization API that allows you to disable and hide controls (also referred to as *widgets*), such as text fields and buttons, on most of its dialog boxes. The customization API allows you to disable or hide controls on an entire class of dialog boxes, for example, parameter dialog boxes via a single method call.

Before attempting to customize a Simulink dialog box or class of dialog boxes, you must first ensure that the dialog box or class of dialog boxes is customizable. Any dialog box that appears in the dialog pane of Model Explorer is customizable. In addition, any dialog box that has dialog and widget IDs is customizable. To determine whether a standalone dialog box (i.e., one that does not appear in Model Explorer) is customizable, open the dialog box, enable dialog and widget ID display (see "Dialog Box and Widget IDs" on page 20-18), and position the mouse over a widget. If a widget ID appears, the dialog box is customizable.

Once you have determined that a dialog box or class of dialog boxes is customizable, you must write M-code to customize the dialog boxes. This entails writing callback functions that disable or hide controls for a specific dialog box or class of dialog boxes (see "Writing Control Customization Callback Functions" on page 20-17) and registering the callback functions via an object called the customization manager (see "Registering Control Customization Callback Functions" on page 20-19). Simulink then invokes

the callback functions to disable or hide the controls whenever a user opens the dialog boxes.

For more information on Dialog Box controls, see:

Example: Disabling a Button on a Simulink® Dialog Box

The following `sl_customization.m` file disables the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box for any model whose name contains "engine."

```
function sl_customization(cm)

% Disable for standalone Configuration Parameters dialog box.
cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)
% Disable for Configuration Parameters dialog box that appears in
% the Model Explorer.
cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end


function disableRTWBuildButton(dialogH)
  hSrc   = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName  = get_param(hModel, 'Name');

  if ~isempty(strfind(modelName, 'engine'))
    % Takes a cell array of widget Factory ID.
    dialogH.disableWidgets({'Simulink.RTWCC.Build'})
  end

end
```

To test this customization:

1 Put the preceding `sl_customization.m` file on the path.

2 Register the customization by entering `sl_refresh_customizations` at the command line or by restarting the MATLAB® software (see "Registering Customizations" on page 20-21).

3 Open the `sldemo_engine` demo model, for example, by entering the command `sldemo_engine` at the command line.

Writing Control Customization Callback Functions

A callback function for disabling or hiding controls on a dialog box should accept one argument: a handle to the dialog box object that contains the controls you want to disable or hide. The dialog box object provides methods that the callback function can use to disable or hide the controls that the dialog box contains.

The dialog box object also provides access to objects containing information about the current model. Your callback function can use these objects to determine whether to disable or hide controls. For example, the following callback function uses these objects to disable the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box displayed in Model Explorer for any model whose name contains "engine."

```
function disableRTWBuildButton(dialogH)

  hSrc  = dialogH.getSource;  % Simulink.RTWCC
  hModel = hSrc.getModel;
  modelName  = get_param(hModel, 'Name');

  if ~isempty(strfind(modelName, 'engine'))
     % Takes a cell array of widget Factory ID.
     dialogH.disableWidgets({'Simulink.RTWCC.Build'})
  end
```

Dialog Box Methods

Dialog box objects provide the following methods for enabling, disabling, and hiding controls:

- `disableWidgets(widgetIDs)`
- `hideWidgets(widgetIDs)`

where widgetIDs is a cell array of widget identifiers (see "Dialog Box and Widget IDs" on page 20-18) that specify the widgets to be disabled or hidden.

Dialog Box and Widget IDs

Dialog box and widget IDs are strings that identify a control on a Simulink dialog box. To determine the dialog box and widget ID for a particular control, execute the following code at the command line:

```
cm = sl_customization_manager;
cm.showWidgetIdAsToolTip = true
```

Then, open the dialog box that contains the control and move the mouse cursor over the control to display a tooltip listing the dialog box and the widget IDs for the control. For example, moving the cursor over the **Start time** field on the **Solver** pane of the Configuration Parameters dialog box reveals that the dialog box ID for the **Solver** pane is Simulink.SolverCC and the widget ID for the **Start time** field is Simulink.SolverCC.StartTime.
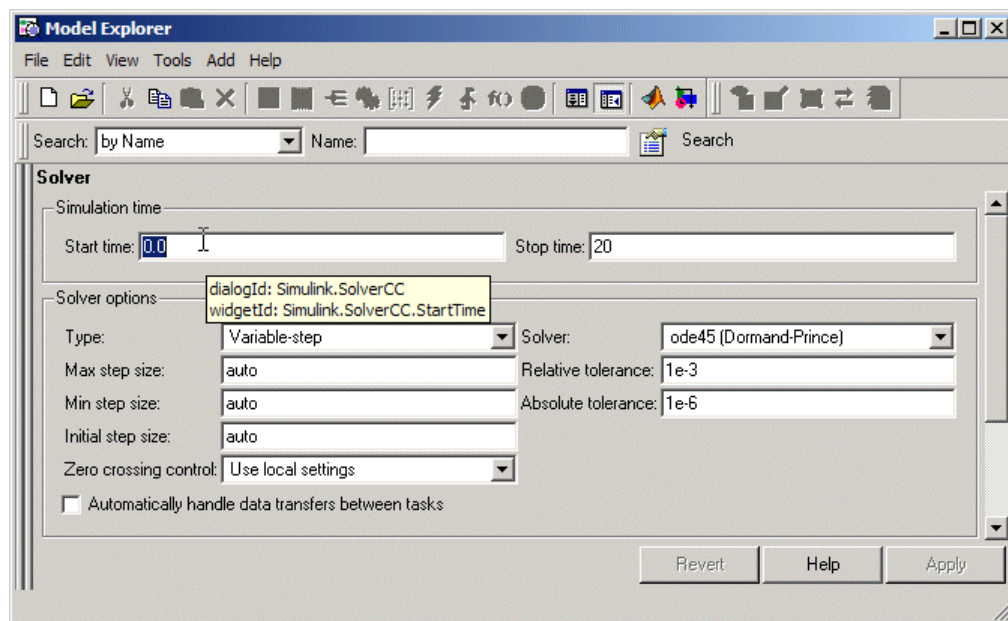
Registering Control Customization Callback Functions

To register control customization callback functions for a particular installation of the Simulink product, include code in the installation's `sl_customization.m` file (see "Registering Customizations" on page 20-21) 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 20-18) and the second is a pointer to the callback function to be registered. Invoking this method causes the registered function to be invoked for each dialog box of the type specified by the dialog box ID. The function is invoked before the dialog box is opened, allowing the function to perform the customizations before they become visible to the user.

The following example registers a callback that disables the **Build** button on the **Real-Time Workshop** pane of the Configuration Parameters dialog box (see "Writing Control Customization Callback Functions" on page 20-17).

```
function sl_customization(cm)

  % Disable for standalone Configuration Parameters dialog box.
  cm.addDlgPreOpenFcn('Simulink.ConfigSet',@disableRTWBuildButton)

  % Disable for Configuration Parameters dialog box that appears in
  % the Model Explorer
  cm.addDlgPreOpenFcn('Simulink.RTWCC',@disableRTWBuildButton)

end
```

Note Registering a customization callback causes the Simulink software to invoke the callback for every instance of the class of dialog boxes specified by the method's dialog box ID argument. This allows you to use a single callback to disable or hide a control for an entire class of dialog boxes. In particular, you can use a single callback to disable or hide the control for a parameter that is common to most built-in blocks. This is because most built-in block dialog boxes are instances of the same dialog box super class.

Registering Customizations

In this section...
"About Registering User Interface Customizations" on page 20-21
"Customization Manager" on page 20-21

About Registering User Interface Customizations

You must register your user-interface customizations using an M-file function called sl_customization.m. This is located on the MATLAB® path of the Simulink® installation that you want to customize. The sl_customization function should accept one argument: a handle to 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 20-21). 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 20-2
- "Disabling and Hiding Model Editor Menu Items" on page 20-13
- "Disabling and Hiding Dialog Box Controls" on page 20-15

The sl_customization.m file is read when the Simulink software starts. If you subsequently change the sl_customization.m file, you must restart the Simulink software or enter the following command at the command line to effect the changes:

```
sl_refresh_customizations
```

Customization Manager

The customization manager includes the following methods:

- addCustomMenuFcn(stdMenuTag, menuSpecsFcn)

  Adds the menus specified by menuSpecsFcn to the end of the standard Simulink menu specified by stdMenuTag. The stdMenuTag argument is a string that specifies the menu to be customized. For example, the stdMenuTag for the Simulink editor's **Tools** menu is 'Simulink:ToolsMenu' (see "Displaying Menu Tags" on page 20-11 for more information). The menuSpecsFcn argument is a handle to a function that returns a list of functions that specify the items to be added to the specified menu. See "Adding Items to Model Editor Menus" on page 20-2 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 20-11 for more information). The filterFcn argument is a pointer to a function that hides or disables the specified menu item. See "Disabling and Hiding Model Editor Menu Items" on page 20-13 for more information.

21

Creating Custom Blocks

When to Create Custom Blocks

Custom blocks allow you to expand the modeling functionality provided by the Simulink® product. By creating a custom block, you can:

- Model behaviors for which the Simulink product does not provide a built-in solution.

- Build more advanced systems.

- Encapsulate systems into a library block that can be copied into multiple models.

- Provide custom graphical user interfaces or analysis routines.

Types of Custom Blocks

MATLAB® Function Blocks

MATLAB® function blocks allow you to use functions to define custom functionality. These blocks serve as a good starting point for creating a custom block if:

- You have an existing MATLAB function that models the custom functionality.
- You find it easier to model custom functionality via a MATLAB function than via a block diagram.
- The custom functionality does not include continuous or discrete dynamic states.

You can create a custom block from an M-function using one of the following types of MATLAB function blocks.

- The Fcn block allows you to use a MATLAB expression to define a single-input, single-output (SISO) block.
- The MATLAB Fcn block allows you to use a MATLAB function to define a SISO block.
- The Embedded MATLAB Function block allows you to define a custom block with multiple inputs and outputs that can be deployed to embedded processors.

Each of these blocks has advantages in particular modeling applications. For example, you can generate code from models containing Embedded MATLAB™ Function blocks while you cannot generate code for models containing a Fcn block.

Subsystem Blocks

Subsystem blocks allow you to build a Simulink® diagram to define custom functionality. These blocks serve as a good starting point for creating a custom block if:

- You have an existing Simulink diagram that models custom functionality.

- You find it easier to model custom functionality via a graphical representation than via hand-written code.

- The custom functionality is a function of continuous or discrete system states.

- The custom functionality can be modeled using existing Simulink blocks.

Once you have a Simulink subsystem that models the desired behavior, you can convert it into a custom block by:

1 Masking the block to hide the block's contents and provide a custom block dialog.

2 Placing the block in a library to prohibit modifications and allow for easily updating copies of the block.

See Chapter 7, "Working with Block Libraries" and Chapter 17, "Creating Block Masks" in Using Simulink for more information.

S-Function Blocks

S-function blocks allow you to write M, C, or C++ code to define custom functionality. These blocks serve as a good starting point for creating a custom block if:

- You have existing M, C, or C++ code that models custom functionality.

- You need to model continuous or discrete dynamic states or other system behaviors that require access to the S-function API.

- The custom functionality cannot be modeled using existing Simulink blocks.

You can create a custom block from an S-function using one of the following types of S-function blocks.

- The Level-2 M-File S-Function block allows you to write your S-function in M. (See "Writing S-Functions in M".) An M-file S-function can be debugged during simulation using the MATLAB debugger.

- The S-Function block allows you to write your S-function in C or C++, or to incorporate existing code into your model via a C MEX wrapper. (See "Writing S-Functions in C".)

- The S-Function Builder block assists you in creating a new C MEX S-function or a wrapper function to incorporate legacy C or C++ code. (See "Building S-Functions Automatically".)

- The Legacy Code Tool transforms existing C or C++ functions into C MEX S-functions. (See "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool".)

The S-function target in the Real-Time Workshop® product automatically generates a C MEX S-function from a graphical subsystem. If you want to build your custom block in a Simulink subsystem, but implement the final version of the block in an S-function, you can use the S-function target to convert the subsystem to an S-function. See "S-Function Target" in the Real-Time Workshop User's Guide for details and limitations on using the S-function target.

Comparing M-File S-Functions to Embedded MATLAB™ Functions

M-file S-functions and Embedded MATLAB functions have some fundamental differences.

- The Real-Time Workshop product can generate code for both M-file S-functions and Embedded MATLAB functions. However, M-file S-functions require a Target Language Compiler (TLC) file for code generation. Embedded MATLAB functions do not require a TLC-file.

- M-file S-functions can use any MATLAB function. Embedded MATLAB functions support only a subset of the MATLAB functions. See "Embedded MATLAB Function Library Reference" in the Embedded MATLAB documentation for a list of supported functions.

- M-file S-functions can model discrete and continuous state dynamics. Embedded MATLAB functions cannot model state dynamics.

Using S-Function Blocks to Incorporate Legacy Code

Each S-function block allows you to incorporate legacy code into your model, as follows.

- An M-file S-function accesses legacy code through its TLC-file. Therefore, the legacy code is available only in the generated code, not during simulation.

- A C MEX S-functions directly calls legacy C or C++ code.

- The S-Function Builder generates a wrapper function that calls the legacy C or C++ code.

- The Legacy Code Tool generates a C MEX S-function to call the legacy C or C++ code, which is optimized for embedded systems. See "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" for more information.

See "Integrating Legacy and Custom Code" in the Real-Time Workshop User's Guide for more information.

See "Example Using S-Functions to Incorporate Legacy C Code" in "Writing S-Functions" for an example.

Comparison of Custom Block Functionality

In this section...
"Custom Block Considerations" on page 21-7
"Modeling Requirements" on page 21-10
"Speed and Code Generation Requirements" on page 21-13

Custom Block Considerations

When creating a custom block, you may want to consider the following.

- Does the custom block need multiple input and output ports?
- Does the block need to model continuous or discrete state behavior?
- Will the block's inputs and outputs have various data attributes, such as data types or complexity?
- How important is the affect of the custom block on the speed of updating the Simulink® diagram or simulating the Simulink model?
- Do you need to generate code for a model containing the custom block?

The following two tables provide an overview of how each custom block type addresses the previous questions. More detailed information for each consideration follows these two tables.

Modeling Requirements

Custom Block Type	Supports Multiple Inputs and Outputs	Models State Dynamics	Supports Various Data Attributes
Subsystem	Yes, including bus signals.	Yes.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.

Modeling Requirements (Continued)

Custom Block Type	Supports Multiple Inputs and Outputs	Models State Dynamics	Supports Various Data Attributes
Fcn	No. Must have a single vector input and scalar output.	No.	Supports only real scalar signals with a data type of double or single.
MATLAB® Fcn	No. Must have a single vector input and output.	No.	Supports only n-D, real, or complex signals with a data type of double.
Embedded MATLAB™ Function	Yes, including bus signals.	No.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.
Level-2 M-file S-function	Yes.	Yes, including limited access to other S-function APIs.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.
C MEX S-function	Yes, including bus signals if using the Legacy Code Tool to generate the S-function.	Yes, including full access to all S-function APIs.	Yes, including all data types, numeric types, and dimensions supported by the Simulink software. Also supports frame-based signals.

Speed and Code Generation Requirements

Custom Block Type	Speed of Updating the Diagram	Simulation Overhead	Code Generation Support
Subsystem	Proportional to the complexity of the subsystem. For library blocks, can be slower the first time the library is loaded.	Proportional to the complexity of the subsystem. Library blocks introduce no additional overhead.	Natively supported.
Fcn	Very fast.	Minimal, but these blocks also provide limited functionality.	Natively supported.
MATLAB Fcn	Fast.	High and incurred when calling out to the MATLAB interpreter. These calls add overhead that should be avoided if simulation speed is a concern.	Not supported.
Embedded MATLAB Function	Can be slower if code must be generated to update the diagram.	Minimal if the MATLAB interpreter is not called. Simulation speed is equivalent to C MEX S-functions when the MATLAB interpreter is not called.	Natively supported, with exceptions. See "Code Generation" on page 21-16 for more information.

Speed and Code Generation Requirements (Continued)

Custom Block Type	Speed of Updating the Diagram	Simulation Overhead	Code Generation Support
Level-2 M-file S-function	Can be slower if the S-function overrides methods executed when updating the diagram.	Higher than for MATLAB Fcn blocks because the MATLAB interpreter is called for every S-function method used. Very flexible, but very costly.	M-file S-functions initialized as a `SimViewingDevice` do not generate code. Otherwise, M-file S-functions require a TLC-file for code generation.
C MEX S-function	Can be slower if the S-function overrides methods executed when updating the diagram.	Minimal, but proportional to the complexity of the algorithm and the efficiency of the code.	Might require a TLC-file.

Modeling Requirements

Multiple Input and Output Ports

The following types of custom blocks support multiple input and output ports.

Custom Block Type	Multiple Input and Output Port Support
Subsystem	Supports multiple input and output ports, including bus signals. In addition, you can modify the number of input and output ports based on user-defined parameters. See "Self-Modifying Linked Subsystems" on page 7-5 for more information.
Fcn, MATLAB Fcn	Supports only a single input and a single output port. You must use a Mux block to combine the inputs and a Demux block to separate the outputs if you need to pass multiple signals into or out of these blocks.

Custom Block Type	Multiple Input and Output Port Support
Embedded MATLAB Function	Supports multiple input and output ports, including bus signals. See "Working with Structures and Bus Signals" on page 22-77 for more information.
S-function (M-file or C MEX)	Supports multiple input and output ports. In addition, you can modify the number of input and output ports based on user-defined parameters. S-functions generated using the Legacy Code Tool also accept Simulink bus signals. See "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" for more information.

State Behavior and the S-Function API

Simulink blocks communicate with the Simulink engine through the S-function API, a set of methods that fully specifies the behavior of blocks. Each custom block type accesses a different sets of the S-function APIs, as follows.

Custom Block Type	S-Function API Support
Subsystem	Communicates directly with the engine. You can model state behaviors using appropriate blocks from the Continuous and Discrete Simulink block libraries.
Fcn, MATLAB Fcn, Embedded MATLAB Function	All create an `mdlOutput` method to calculate the value of the outputs given the value of the inputs. You cannot access any other S-function API methods using one of these blocks and, therefore, cannot model state behavior.
M-file S-function	Accesses a larger subset of the S-function APIs, including the methods needed to model continuous and discrete states. For a list of supported methods, see "Level-2 M-File S-Function Callback Methods" in "Writing S-Functions".
C MEX S-function	Accesses the complete set of S-function APIs.

Data Attribute Support

All custom block types support real scalar inputs and outputs with a data type of double.

Custom Block Type	Data Attribute Support
Subsystem	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.
Fcn	Supports only double or single data types. In addition, the input and output cannot be complex and the output must be a scalar signal. Does not support frame-based signals.
MATLAB Fcn	Supports 2-D, n-D, and complex signals, but the signal must have a data type of double. Does not support frame-based signals.
Embedded MATLAB Function	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.
S-function (M-file or C MEX)	Supports any data type supported by the Simulink software, including fixed-point types. Also supports complex, 2-D, n-D, and frame-based signals.

Speed and Code Generation Requirements

Updating the Simulink® Diagram

The Simulink software updates the diagram before every simulation and whenever requested by the user. Every block introduces some overhead into the "update diagram" process.

Custom Block Type	Speed of Updating the Diagram
Subsystem	The speed is proportional to the complexity of the algorithm implemented in the subsystem. If the subsystem is contained in a library, some cost is incurred when the Simulink software loads any unloaded libraries the first time the diagram is updated or readied for simulation. If all referenced library blocks remain unchanged, the Simulink software does not subsequently reload the library and compiling the model becomes faster than if the model did not use libraries.
Fcn, MATLAB Fcn	Does not incur greater update cost than other Simulink blocks.
Embedded MATLAB Function	Performs simulation through code generation, so these blocks might take a significant amount of time when first updated. However, because code generation is incremental, if the block and the signals connected to it have not changed, the Simulink software does not repeatedly update the block.
S-function (M-file or C MEX)	Incurs greater costs than other Simulink blocks only if it overrides methods executed when updating the diagram. If these methods become complex, they can contribute significantly to the time it takes to update the diagram. For a list of methods executed when updating the diagram, see the process view in "How the Simulink Engine Interacts with C S-Functions". When updating the diagram, the Simulink software invokes all relevant methods in the model initialization phase up to, but not including, `mdlStart`.

Simulation Overhead

For most applications, any of the custom block types provide acceptable simulation performance. Use the Simulink profiler to obtain an indication of the actual performance. See "Capturing Performance Data" on page 19-33 for more information.

You can break simulation performance into two categories. The interface cost is the time it takes to move data from the Simulink engine into the block. The algorithm cost is the time needed to perform the algorithm that the block implements.

Custom Block Type	Simulation Overhead
Subsystem	If included in a library, introduces no interface or algorithm costs beyond what would normally be incurred if the block existed as a regular subsystem in the model.
Fcn	Has the least simulation overhead. The block is tightly integrated with the Simulink engine and implements a rudimentary expression language that is efficiently interpreted.
MATLAB Fcn	Has a higher interface cost than most blocks and the same algorithm cost as a MATLAB function.
	When block data (such as inputs and outputs) is accessed or returned from a MATLAB Fcn block, the Simulink engine packages this data into MATLAB arrays. This packaging takes additional time and causes a temporary increase in memory during communication. If you pass large amounts of data across this interface, such as, frames or arrays, this overhead can be substantial.
	Once the data has been converted, the MATLAB interpreter executes the algorithm. As a result, the algorithm cost is the same as for MATLAB function. Efficient code can be competitive with C code if the MATLAB software is able to optimize it, or if the code uses the highly optimized MATLAB library functions.

Custom Block Type	Simulation Overhead
Embedded MATLAB Function	Performs simulation through code generation and so incurs the same interface cost as standard blocks. The algorithm cost of this block is harder to analyze because of the block's implementation. On average, an Embedded MATLAB function and a MATLAB function run at about the same speed. To further reduce the algorithm cost, you can disable debugging for all the Embedded MATLAB Function blocks in your model. If the Embedded MATLAB function uses simulation-only capabilities to call out to the MATLAB interpreter, it incurs all the costs that an M-file S-function or MATLAB Fcn block incur. Calling out to the MATLAB interpreter from an Embedded MATLAB function produces a warning to prevent you from doing so unintentionally.
M-file S-function	Incurs the same algorithm costs as the MATLAB Fcn block, but with a slightly higher interface cost. Because M-file S-functions can handle multiple inputs and outputs, the packaging is more complicated than for the MATLAB Fcn block. In addition, the Simulink engine calls the MATLAB interpreter for each block method you implement whereas for the MATLAB Fcn block, it calls the MATLAB interpreter only for the `mdlOutput` method.
C MEX S-function	Simulates via the compiled code and so incurs the same interface cost as standard blocks. The algorithm cost depends on the complexity of the S-function.

Code Generation

Not all custom block types support code generation.

Custom Block Type	Code Generation Support
Subsystem	Supports code generation.
Fcn	Supports code generation.
MATLAB Fcn	Does not support code generation.
Embedded MATLAB Function	Supports code generation. However, if your Embedded MATLAB Function block calls out to the MATLAB interpreter, it will build with the Real-Time Workshop® product only if the calls to the MATLAB interpreter do not affect the block's outputs. Under this condition, the Real-Time Workshop product omits these calls from the generated C code. This feature allows you to leave visualization code in place, even when generating embedded code.
M-file S-function	Generates code only if you implement the algorithm using a Target Language Compiler (TLC) function. In accelerated and external mode simulations, you can choose to execute the S-function in interpretive mode by calling back to the MATLAB interpreter without implementing the algorithm in TLC. If the M-file S-function is a `SimViewingDevice`, the Real-Time Workshop product automatically omits the block during code generation.
C MEX S-function	Supports code generation. For noninlined S-functions, the Real-Time Workshop product uses the C MEX function during code generation. However, you must write a TLC-file for the S-function if you need to either inline the S-function or create a wrapper for hand-written code. See "Writing S-Functions for Real-Time Workshop Code Generation" in the Real-Time Workshop User's Guide for more information.

Expanding Custom Block Functionality

You can expand the functionality of any custom block using callbacks and Handle Graphics.

Block callbacks perform user-defined actions at specific points in the simulation. For example, the callback can load data into the MATLAB® workspace before the simulation or generate a graph of simulation data at the end of the simulation. You can assign block callbacks to any of the custom block types. For a list of available callbacks and more information on how to use them, see "Creating Block Callback Functions" on page 3-55.

GUIDE, the MATLAB graphical user interface development environment, provides tools for easily creating custom user interfaces. See *MATLAB Creating Graphical User Interfaces* for more information on using GUIDE.

Tutorial: Creating a Custom Block

How to Design a Custom Block

In general, you use the following process to design a custom block:

1 Define the behavior required by the custom block.

2 Decide which custom block type to use.

3 Determine if the block should reside in a library.

4 Add a graphical user interface to the block.

Suppose you want to create a customized saturation block that limits the upper and lower bounds of a signal based on either a block parameter or the value of an input signal. In a second version of the block, you want the option to plot the saturation limits after the simulation is finished. The following tutorial steps you through designing these blocks. The library customsat_lib.mdl contains the two versions of the customized saturation block.
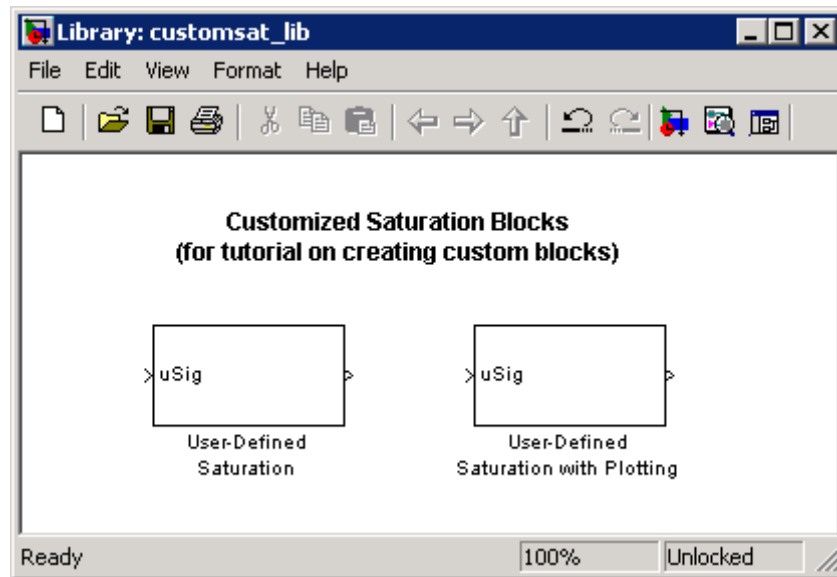
The example model `sldemo_customsat.mdl` uses the basic version of the block.
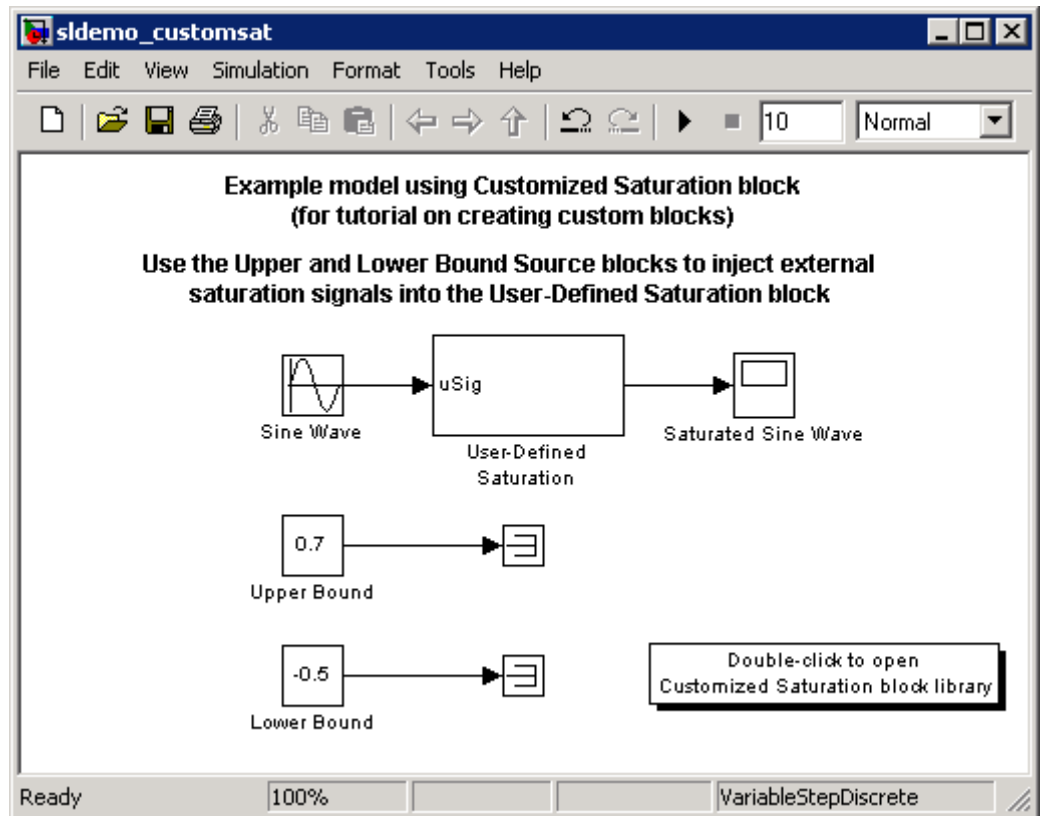
Defining Custom Block Behavior

Begin by defining the features and limitations of your custom block. In this example, the block supports the following features:

- Turning on and off the upper or lower saturation limit.

- Setting the upper and/or lower limits via a block parameters.

- Setting the upper and/or lower limits using an input signal.

It also has the following restrictions:

- The input signal under saturation must be a scalar.

- The input signal and saturation limits must all have a data type of double.

- Code generation is not required.

Deciding on a Custom Block Type

Based on the custom block's features, the implementation needs to support the following:

- Multiple input ports

- A relatively simple algorithm

- No continuous or discrete system states

Therefore, this tutorial implements the custom block using a Level-2 M-file S-function. M-file S-functions support multiple inputs and, because the algorithm is simple, do not have significant overhead when updating the diagram or simulating the model. See "Comparison of Custom Block Functionality" on page 21-7 for a description of the different functionality provided by M-file S-functions as compared to other types of custom blocks.

Parameterizing the M-File S-Function

Begin by defining the S-function parameters. This example requires four parameters:

- The first parameter indicates how the upper saturation limit is set. The limit can be off, set via a block parameter, or set via an input signal.

- The second parameter is the value of the upper saturation limit. This value is used only if the upper saturation limit is set via a block parameter. In the event this parameter is used, you should be able to change the parameter's value during the simulation, i.e., the parameter is tunable.

- The third parameter indicates how the lower saturation limit is set. The limit can be off, set via a block parameter, or set via an input signal.

- The fourth parameter is the value of the lower saturation limit. This value is used only if the lower saturation limit is set via a block parameter. As with the upper saturation limit, this parameter is tunable when in use.

The first and third S-function parameters represent modes that must be translated into values the S-function can recognize. Therefore, define the following values for the upper and lower saturation limit modes:

- 1 indicates that the saturation limit is off.

- 2 indicates that the saturation limit is set via a block parameter.

- 3 indicates that the saturation limit is set via an input signal.

Writing the M-File S-Function

Once the S-function parameters and functionality are defined, write the S-function. The template msfuntmpl.m provides a starting point for writing a Level-2 M-file S-function. You can find a completed version of the custom saturation block in the file custom_sat.m. Save this file to your working directory before continuing with this tutorial.

This S-function modifies the S-function template as follows:

- The setup function initializes the number of input ports based on the values entered for the upper and lower saturation limit modes. If the limits are set via input signals, the method adds input ports to the block. The setup method then indicates there are four S-function parameters and sets the parameter tunability. Finally, the method registers the S-function methods used during simulation.

```
function setup(block)

% The Simulink engine passes an instance of the Simulink.MSFcnRunTimeBlock
% class to the setup method in the input argument "block". This is known as
% the S-function block's run-time object.

% Register original number of input ports based on the S-function
% parameter values

try % Wrap in a try/catch, in case no S-function parameters are entered
    lowMode   = block.DialogPrm(1).Data;
    upMode    = block.DialogPrm(3).Data;
    numInPorts = 1 + isequal(lowMode,3) + isequal(upMode,3);
catch
```

```
    numInPorts=1;
end % try/catch
block.NumInputPorts = numInPorts;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';

% Override output port properties
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';

% Register parameters. In order:
% -- If the upper bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The upper limit value. Should be empty if the upper limit is off or
%    set via an input signal
% -- If the lower bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The lower limit value. Should be empty if the lower limit is off or
%    set via an input signal
block.NumDialogPrms     = 4;
block.DialogPrmsTunable = {'Nontunable','Tunable','Nontunable', ...
    'Tunable'};

% Register continuous sample times [0 offset]
block.SampleTimes = [0 0];

%% ------------------------------------------------------------------
%% Options
%% ------------------------------------------------------------------
% Specify if Accelerator should use TLC or call back into
% M-file
block.SetAccelRunOnTLC(false);
```

```
%% -----------------------------------------------------------------
%% Register methods called during update diagram/compilation
%% -----------------------------------------------------------------

block.RegBlockMethod('CheckParameters',     @CheckPrms);
block.RegBlockMethod('ProcessParameters',   @ProcessPrms);
block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Outputs',             @Outputs);
block.RegBlockMethod('Terminate',           @Terminate);
%end setup function
```

- The CheckParameters method verifies the values entered into the Level-2 M-file S-function block.

```
function CheckPrms(block)

lowMode = block.DialogPrm(1).Data;
lowVal  = block.DialogPrm(2).Data;
upMode  = block.DialogPrm(3).Data;
upVal   = block.DialogPrm(4).Data;

% The first and third dialog parameters must have values of 1-3
if ~any(upMode == [1 2 3]);
    error('The first dialog parameter must be a value of 1, 2, or 3');
end

if ~any(lowMode == [1 2 3]);
    error('The first dialog parameter must be a value of 1, 2, or 3');
end

% If the upper or lower bound is specified via a dialog, make sure there
% is a specified bound. Also, check that the value is of type double
if isequal(upMode,2),
    if isempty(upVal),
        error('Enter a value for the upper saturation limit.');
    end
    if ~strcmp(class(upVal), 'double')
        error('The upper saturation limit must be of type double.');
    end
end
```

```
        if isequal(lowMode,2),
            if isempty(lowVal),
                error('Enter a value for the lower saturation limit.');
            end
            if ~strcmp(class(lowVal), 'double')
                error('The lower saturation limit must be of type double.');
            end
        end

        % If a lower and upper limit are specified, make sure the specified
        % limits are compatible.
        if isequal(upMode,2) && isequal(lowMode,2),
            if lowVal >= upVal,
                error('The lower bound must be less than the upper bound.');
            end
        end

        %end CheckPrms function
```

- The `ProcessParameters` and `PostPropagationSetup` methods handle the S-function parameter tuning.

```
    function ProcessPrms(block)

    %% Update run time parameters
    block.AutoUpdateRuntimePrms;

    %end ProcessPrms function


    function DoPostPropSetup(block)

    %% Register all tunable parameters as runtime parameters.
    block.AutoRegRuntimePrms;

    %end DoPostPropSetup function
```

- The `Outputs` method calculates the block's output based on the S-function parameter settings and any input signals.

```
function Outputs(block)

lowMode   = block.DialogPrm(1).Data;
upMode    = block.DialogPrm(3).Data;
sigVal    = block.InputPort(1).Data;
lowPortNum = 2; % Initialize potential input number for lower saturation limit

% Check upper saturation limit
if isequal(upMode,2), % Set via a block parameter
    upVal = block.RuntimePrm(2).Data;
elseif isequal(upMode,3), % Set via an input port
    upVal = block.InputPort(2).Data;
    lowPortNum = 3; % Move lower boundary down one port number
else
    upVal = inf;
end

% Check lower saturation limit
if isequal(lowMode,2), % Set via a block parameter
    lowVal = block.RuntimePrm(1).Data;
elseif isequal(lowMode,3), % Set via an input port
    lowVal = block.InputPort(lowPortNum).Data;
else
    lowVal = -inf;
end

% Assign new value to signal
if sigVal > upVal,
    sigVal = upVal;
elseif sigVal < lowVal,
    sigVal=lowVal;
end

block.OutputPort(1).Data = sigVal;

%end Outputs function
```

Placing Custom Blocks in a Library

Libraries allow you to share your custom blocks with other users, easily
update the functionality of copies of the custom block, and collect blocks for

a particular project into a single location. This example places the custom saturation block into a library as follows:

1 Open a new Simulink® library (see).

2 Add a new Level-2 M-file S-Function block from the Simulink User-Defined Functions library into your new library.

3 Double-click the block to open its Block Parameters dialog box. Enter the name of the S-function `custom_sat` into the **M-file name** field.

4 Enter the following default values into the **Parameters** field.

```
2,-1,2,1
```

5 Click **OK** on the Block Parameters dialog box.

6 Save the library to your working directory as `saturation_lib.mdl`. The following figure shows the resulting custom saturation block library.
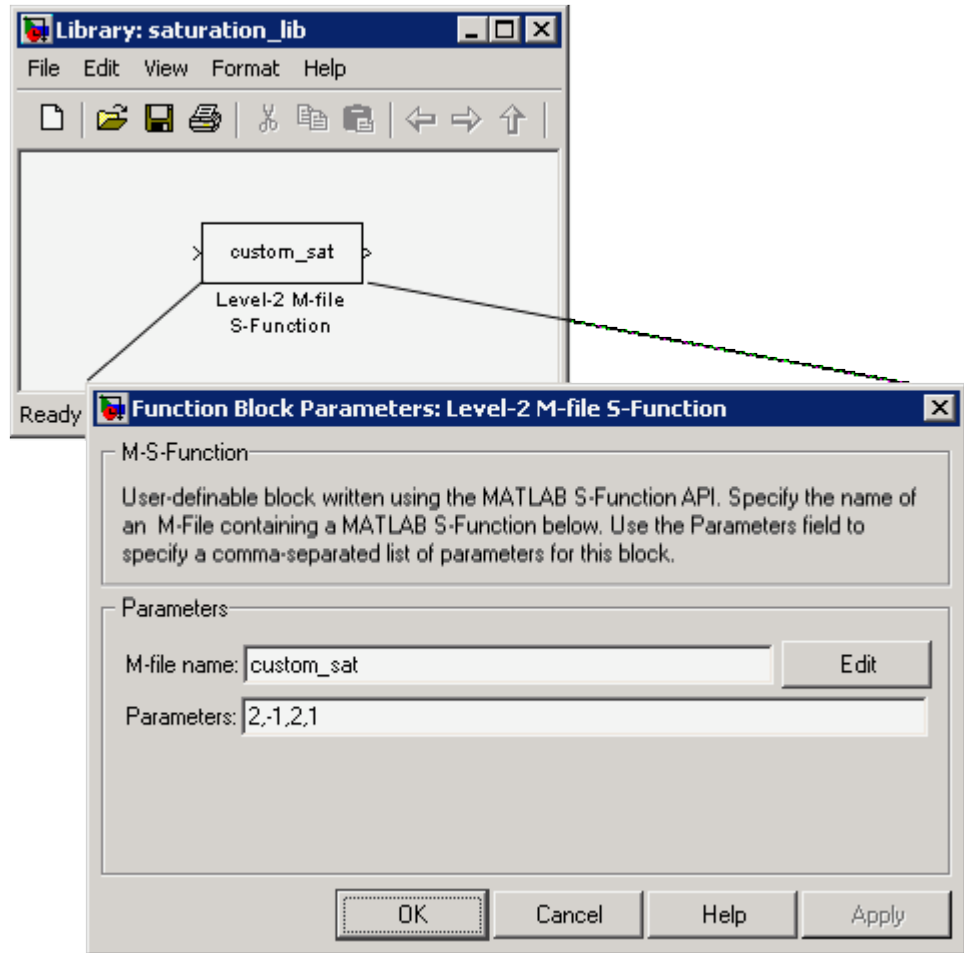
At this point, you have created a custom saturation block that can be shared with other users. You can make the block easier to use by adding a customized graphical user interface.

Adding a Graphical User Interface to a Custom Block

You can create a simple block dialog for the custom saturation block using the provided masking capabilities. Masking the block also allows you to add port labels to indicate which port corresponds to the input signal and the saturation limits.

To mask the block:

1 Right-click the custom saturation block in `saturation_lib.mdl` and select **Mask M-file S-Function** from the context menu. The Mask Editor opens.

2 On the **Icon** pane, enter the following into **Drawing commands**.

```
port_label('input',1,'uSig')
```

This command labels the default port as the input signal under saturation.

3 On the **Parameters** pane, add four parameters corresponding to the four S-function parameters. From top to bottom, set up each parameter's properties as follows.

Prompt	Variable	Type	Tunable	Popups	Action for Dialog Callback
Upper boundary:	upMode	popup	No	No limit | Enter limit as parameter | Limit using input signal	`'upperbound_callback'`
Upper limit:	upVal	edit	Yes	N/A	`'upperparam_callback'`
Lower boundary:	lowMode	popup	No	No limit | Enter limit as parameter | Limit using input signal	`'lowerbound_callback'`
Lower limit:	lowVal	edit	Yes	N/A	`'lowerparam_callback'`

The dialog callback is invoked using the `action` string in the following command:

```
customsat_callback(action,gcb)
```

The M-file `customsat_callback.m` contains the mask parameter callbacks. If you are stepping through this tutorial, open this file and save it to your working directory. This M-file, described in detail later, has two input arguments. The first input argument is a string indicating which mask parameter invoked the callback. The second input argument is the handle to the associated Level-2 M-file S-Function block.

The following figure shows the final **Parameters** pane in the Mask Editor.
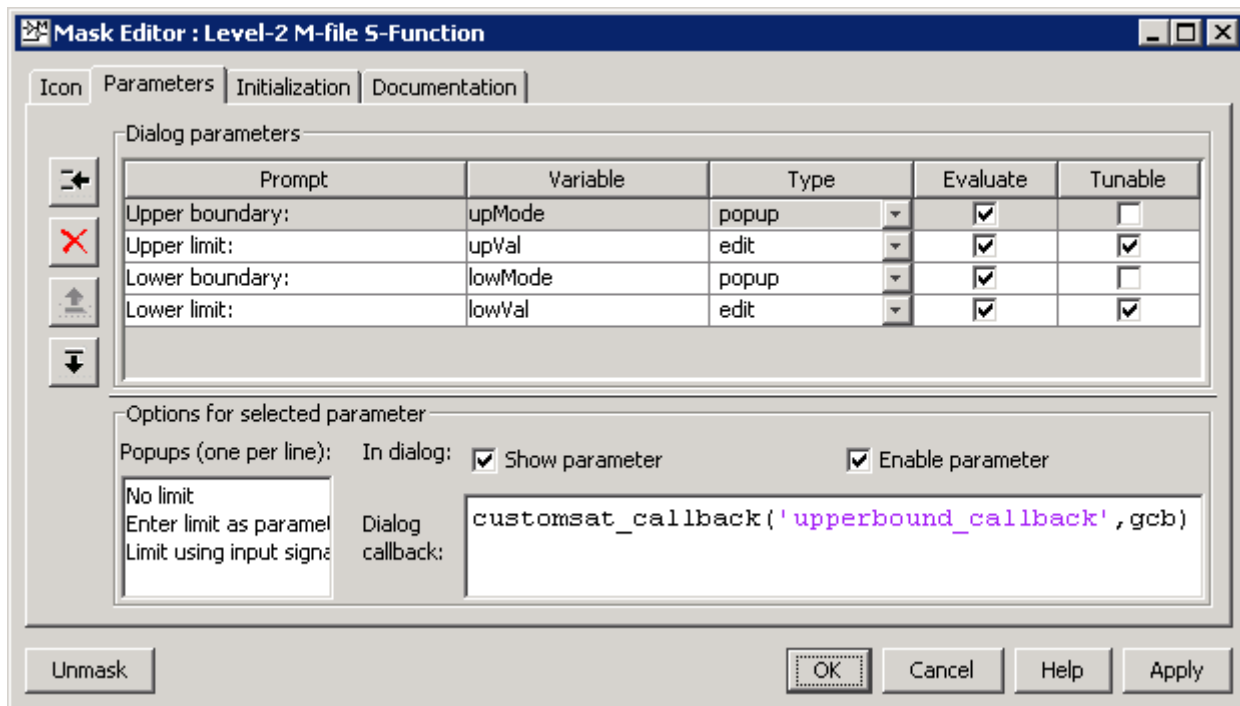
4 On the Mask Editor's **Initialization** pane, select **Allow library block to modify its contents**. This setting allows the S-function to change the number of ports on the block.

5 On the **Documentation** pane:

- Enter `Customized Saturation` into the **Mask type** field.

- Enter the following into the **Mask description** field.

  `Limit the input signal to an upper and lower saturation value`
  `set either through a block parameter or input signal.`

6 Click **OK** on the Mask Editor to complete the mask parameters dialog.

7 To map the S-function parameters to the mask parameters, right-click the Level-2 M-file S-Function block and select **Look Under Mask**. The Level-2 M-file S-Function Block Parameters dialog box opens.

8 Change the entry in the **Parameters** field as follows.

  `lowMode,lowVal,upMode,upVal`

The following figure shows the new Block Parameters dialog.
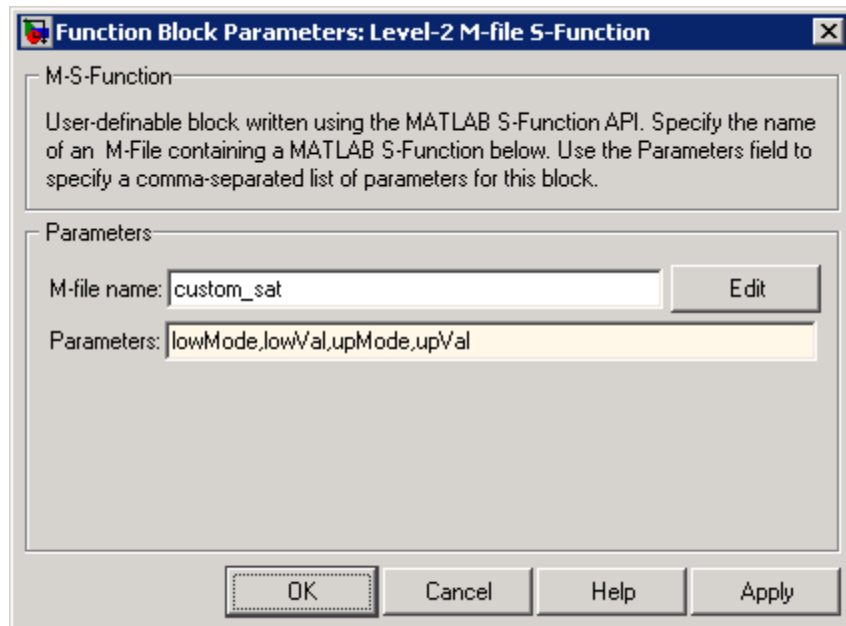
9 Click **OK** on the Level-2 M-file S-Function Block Parameters dialog box. Double-clicking the new version of the customized saturation block opens the mask parameter dialog box shown in the following figure.
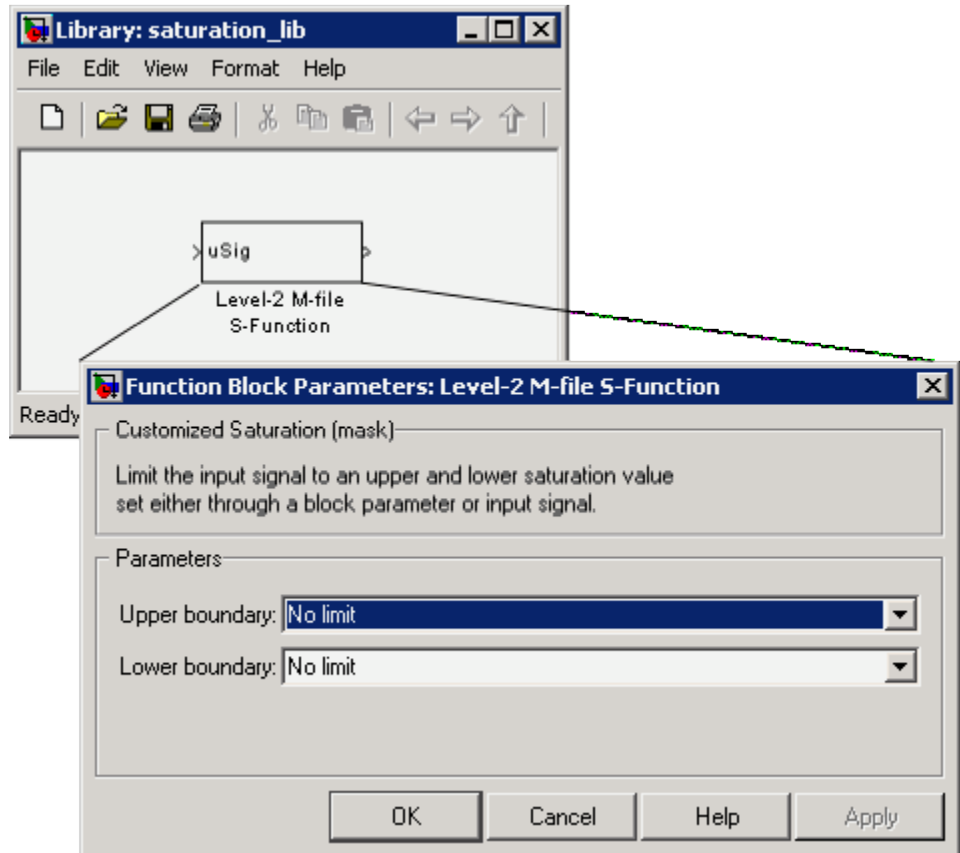
To create a more complicated graphical user interface, place a Handle Graphics user interface on top of the masked block. The block's OpenFcn would invoke the Handle Graphics user interface, which uses calls to set_param to modify the S-function block's parameters based on settings in the user interface.

Writing the Mask Callback

The function customsat_callback.m contains the mask callback code for the custom saturation block's mask parameter dialog box. This function invokes subfunctions corresponding to each mask parameter through a call to feval.

The following subfunction controls the visibility of the upper saturation limit's field based on the selection for the upper saturation limit's mode. The callback begins by obtaining values for all mask parameters using a call to get_param with the property name MaskValues. If the callback needed the value of only one mask parameter, it could call get_param with the specific mask parameter name, for example, get_param(block,'upMode'). Because this example needs two of the mask parameter values, it uses the MaskValues property to reduce the calls to get_param.

The callback then obtains the visibilities of the mask parameters using a call to get_param with the property name MaskVisbilities. This call returns a cell array of strings indicating the visibility of each mask parameter. The callback alters the values for the mask visibilities based on the selection for the upper saturation limit's mode and then updates the port label string.

The callback finally uses the set_param command to update the block's MaskDisplay property to label the block's input ports.

```
function customsat_callback(action,block)
% CUSTOMSAT_CALLBACK contains callbacks for custom saturation block

%   Copyright 2003-2007 The MathWorks, Inc.

%% Use function handle to call appropriate callback
feval(action,block)

%% Upper bound callback
function upperbound_callback(block)

vals = get_param(block,'MaskValues');
vis = get_param(block,'MaskVisibilities');
portStr = {'port_label(''input'',1,''uSig'')'};
switch vals{1}
    case 'No limit'
        set_param(block,'MaskVisibilities',[vis(1);{'off'};vis(3:4)]);
```

```
      case 'Enter limit as parameter'
          set_param(block,'MaskVisibilities',[vis(1);{'on'};vis(3:4)]);
      case 'Limit using input signal'
          set_param(block,'MaskVisibilities',[vis(1);{'off'};vis(3:4)]);
          portStr = [portStr;{'port_label(''input'',2,''up'')'}];
  end
  if strcmp(vals{3},'Limit using input signal'),
      portStr = [portStr;{['port_label(''input'',',num2str(length(portStr)+1), ...
          ',''low'')']}];
  end
  set_param(block,'MaskDisplay',char(portStr));
```

The final call to set_param invokes the setup function in the M-file S-function custom_sat.m. Therefore, the setup function can be modified to set the number of input ports based on the mask parameter values instead of on the S-function parameter values. This change to the setup function keeps the number of ports on the Level-2 M-File S-Function block consistent with the values shown in the mask parameter dialog box.

The modified M-file S-function custom_sat_final.m contains the following new setup function. If you are stepping through this tutorial, open the file and save it to your working directory.

```
%% Function: setup =====================================================
function setup(block)

% Register original number of ports based on settings in Mask Dialog
ud = getPortVisibility(block);
numInPorts = 1 + isequal(ud(1),3) + isequal(ud(2),3);

block.NumInputPorts = numInPorts;
block.NumOutputPorts = 1;

% Setup port properties to be inherited or dynamic
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;

% Override input port properties
block.InputPort(1).DatatypeID  = 0;  % double
block.InputPort(1).Complexity  = 'Real';
```

```
% Override output port properties
block.OutputPort(1).DatatypeID  = 0; % double
block.OutputPort(1).Complexity  = 'Real';

% Register parameters. In order:
% -- If the upper bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The upper limit value. Should be empty if the upper limit is off or
%    set via an input signal
% -- If the lower bound is off (1) or on and set via a block parameter (2)
%    or input signal (3)
% -- The lower limit value. Should be empty if the lower limit is off or
%    set via an input signal
block.NumDialogPrms     = 4;
block.DialogPrmsTunable = {'Nontunable','Tunable','Nontunable','Tunable'};

% Register continuous sample times [0 offset]
block.SampleTimes = [0 0];

%% ------------------------------------------------------------------
%% Options
%% ------------------------------------------------------------------
% Specify if Accelerator should use TLC or call back into
% M-file
block.SetAccelRunOnTLC(false);

%% ------------------------------------------------------------------
%% Register methods called during update diagram/compilation
%% ------------------------------------------------------------------

block.RegBlockMethod('CheckParameters',      @CheckPrms);
block.RegBlockMethod('ProcessParameters',    @ProcessPrms);
block.RegBlockMethod('PostPropagationSetup', @DoPostPropSetup);
block.RegBlockMethod('Outputs',              @Outputs);
block.RegBlockMethod('Terminate',            @Terminate);
%endfunction
```

The getPortVisibility subfunction in custom_sat_final.m uses the saturation limit modes to construct a flag that is passed back to the setup

function. The setup function uses this flag to determine the necessary number of input ports.

```
%% Function: Get Port Visibilities ========================================
function ud = getPortVisibility(block)

ud = [0 0];

vals = get_param(block.BlockHandle,'MaskValues');
switch vals{1}
    case 'No limit'
        ud(2) = 1;
    case 'Enter limit as parameter'
        ud(2) = 2;
    case 'Limit using input signal'
        ud(2) = 3;
end

switch vals{3}
    case 'No limit'
        ud(1) = 1;
    case 'Enter limit as parameter'
        ud(1) = 2;
    case 'Limit using input signal'
        ud(1) = 3;
end
```

Updating the Library

Update the library saturation_lib.mdl so that it calls custom_sat_final.m.

1 Right-click the Level-2 M-file S-Function block in saturation_lib.mdl and select **Look Under Mask**. The Level-2 M-file S-Function Block Parameters dialog box opens.

2 Enter custom_sat_final in the **M-file name** field, as shown in the following figure.
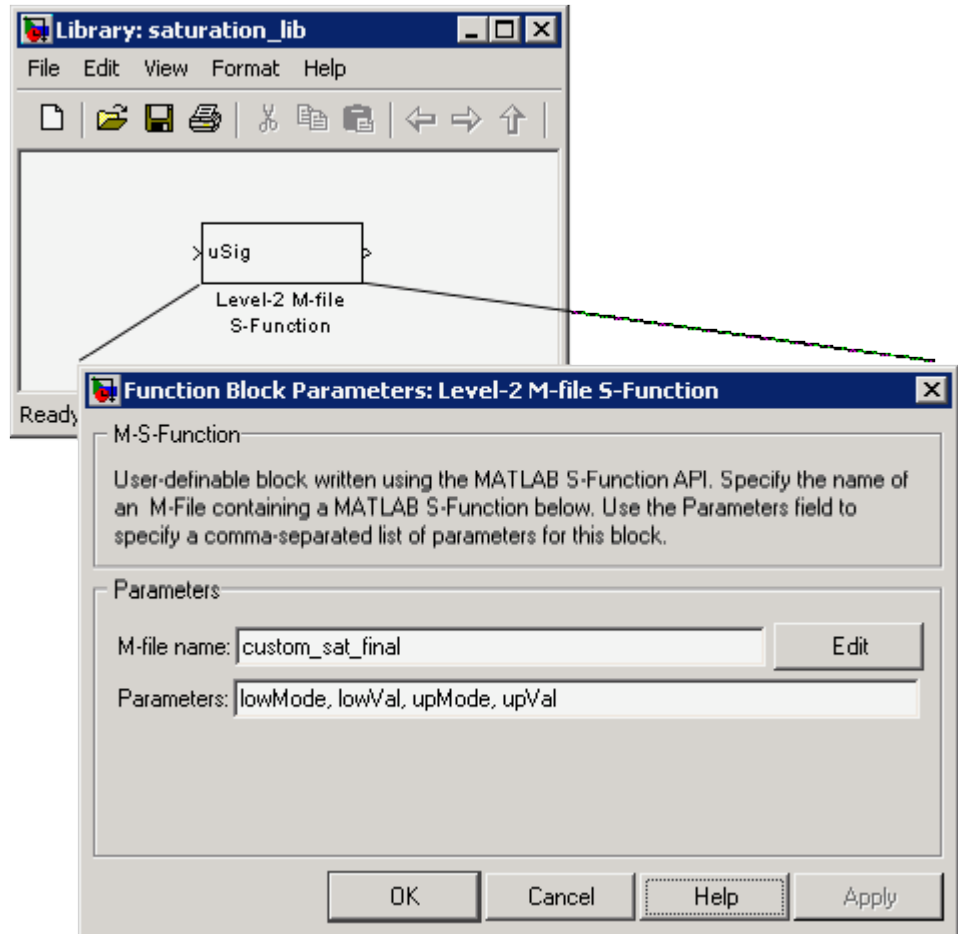
3 Click **OK** on the Block Parameters dialog box.

Adding Block Functionality Using Block Callbacks

The User-Defined Saturation with Plotting block in `customsat_lib.mdl`
uses block callbacks to add functionality to the original custom saturation
block. This block provides an option to plot the saturation limits when the
simulation ends. The following steps show how to modify the original custom
saturation block to create this new block.

1 Add a check box to the mask parameter dialog box to toggle the plotting option on and off, as shown in the following figure.
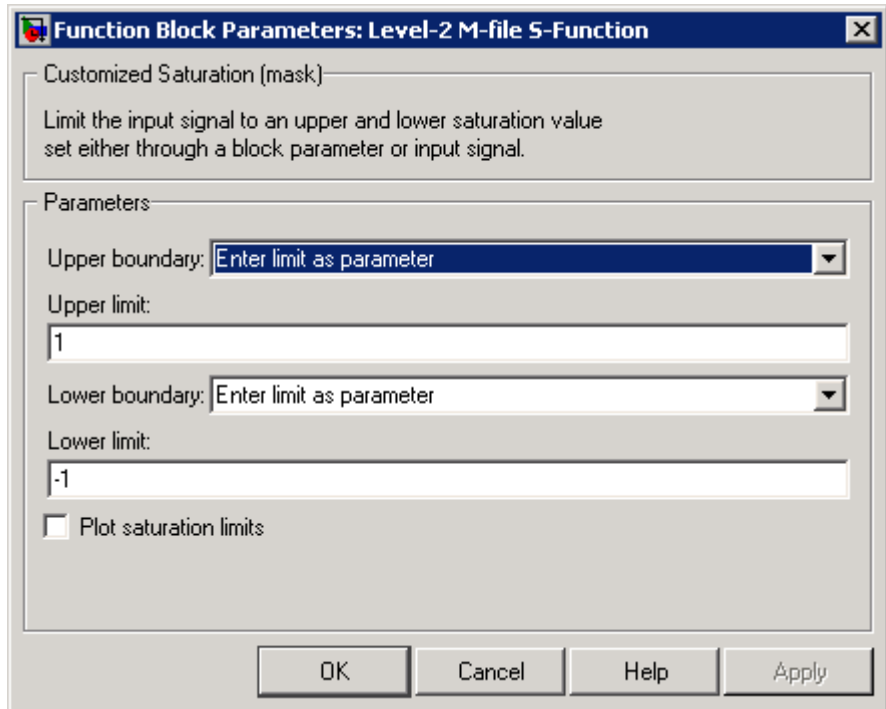
To add this check box:

a Right-click the Level-2 M-file S-Function block in saturation_lib.mdl and select **Edit Mask**.

b On the Mask Editor's **Parameters** pane, add a fifth mask parameter with the following properties.

Property	Value
Prompt	Plot saturation limits
Variable	plotcheck
Type	checkbox

Property	Value
Tunable	No
Dialog callback	customsat_callback('plotsaturation',gcb);

c Click **OK** on the Mask Editor.

2 Write a callback for the new check box. The callback initializes a structure to store the saturation limit values during simulation in the Level-2 M-File S-Function block's UserData. The M-file customsat_plotcallback.m contains this new callback, as well as modified versions of the previous callbacks to handle the new mask parameter. If you are following through this example, open customsat_plotcallback.m and copy its subfunctions over the previous subfunctions in customsat_callback.m.

```
%% Plotting checkbox callback
function plotsaturation(block)

% Reinitialize the block's userdata
vals = get_param(block,'MaskValues');
ud = struct('time',[],'upBound',[],'upVal',[],'lowBound',[],'lowVal',[]);

if strcmp(vals{1},'No limit'),
    ud.upBound = 'off';
else
    ud.upBound = 'on';
end

if strcmp(vals{3},'No limit'),
    ud.lowBound = 'off';
else
    ud.lowBound = 'on';
end

set_param(gcb,'UserData',ud);
```

3 Update the M-file S-function's Outputs method to store the saturation limits, if applicable, as done in the new M-file S-function custom_sat_plot.m. If you are following through this example, copy the

Outputs method in `custom_sat_plot.m` over the original `Outputs` method in `custom_sat_final.m`

```
%% Function: Outputs =====================================================
function Outputs(block)

lowMode    = block.DialogPrm(1).Data;
upMode     = block.DialogPrm(3).Data;
sigVal     = block.InputPort(1).Data;
vals = get_param(block.BlockHandle,'MaskValues');
plotFlag = vals{5};
lowPortNum = 2;

% Check upper saturation limit
if isequal(upMode,2)
    upVal = block.RuntimePrm(2).Data;
elseif isequal(upMode,3)
    upVal = block.InputPort(2).Data;
    lowPortNum = 3; % Move lower boundary down one port number
else
    upVal = inf;
end

% Check lower saturation limit
if isequal(lowMode,2),
    lowVal = block.RuntimePrm(1).Data;
elseif isequal(lowMode,3)
    lowVal = block.InputPort(lowPortNum).Data;
else
    lowVal = -inf;
end

% Use userdata to store limits, if plotFlag is on
if strcmp(plotFlag,'on');
    ud = get_param(block.BlockHandle,'UserData');
    ud.lowVal = [ud.lowVal;lowVal];
    ud.upVal = [ud.upVal;upVal];
    ud.time = [ud.time;block.CurrentTime];
    set_param(block.BlockHandle,'UserData',ud)
end
```

```
% Assign new value to signal
if sigVal > upVal,
    sigVal = upVal;
elseif sigVal < lowVal,
    sigVal=lowVal;
end

block.OutputPort(1).Data = sigVal;

%endfunction
```

4 Write the function `plotsat.m` to plot the saturation limits. This function takes the handle to the Level-2 M-File S-Function block and uses this handle to retrieve the block's `UserData`. If you are following through this tutorial, save `plotsat.m` to your working directory.

```
function plotSat(block)

% PLOTSAT contains the plotting routine for custom_sat_plot
%   This routine is called by the S-function block's StopFcn.

ud = get_param(block,'UserData');
fig=[];
if ~isempty(ud.time)
    if strcmp(ud.upBound,'on')
        fig = figure;
        plot(ud.time,ud.upVal,'r');
        hold on
    end
    if strcmp(ud.lowBound,'on')
        if isempty(fig),
            fig = figure;
        end
        plot(ud.time,ud.lowVal,'b');
    end
    if ~isempty(fig)
        title('Upper bound in red. Lower bound in blue.')
    end
```

```
            % Reinitialize userdata
            ud.upVal=[];
            ud.lowVal=[];
            ud.time = [];
            set_param(block,'UserData',ud);
        end
```

5 Right-click the Level-2 M-file S-Function block and select **Block Properties**. The Block Properties dialog box opens. On the **Callbacks** pane, modify the StopFcn to call the plotting callback as shown in the following figure, then click **OK**.
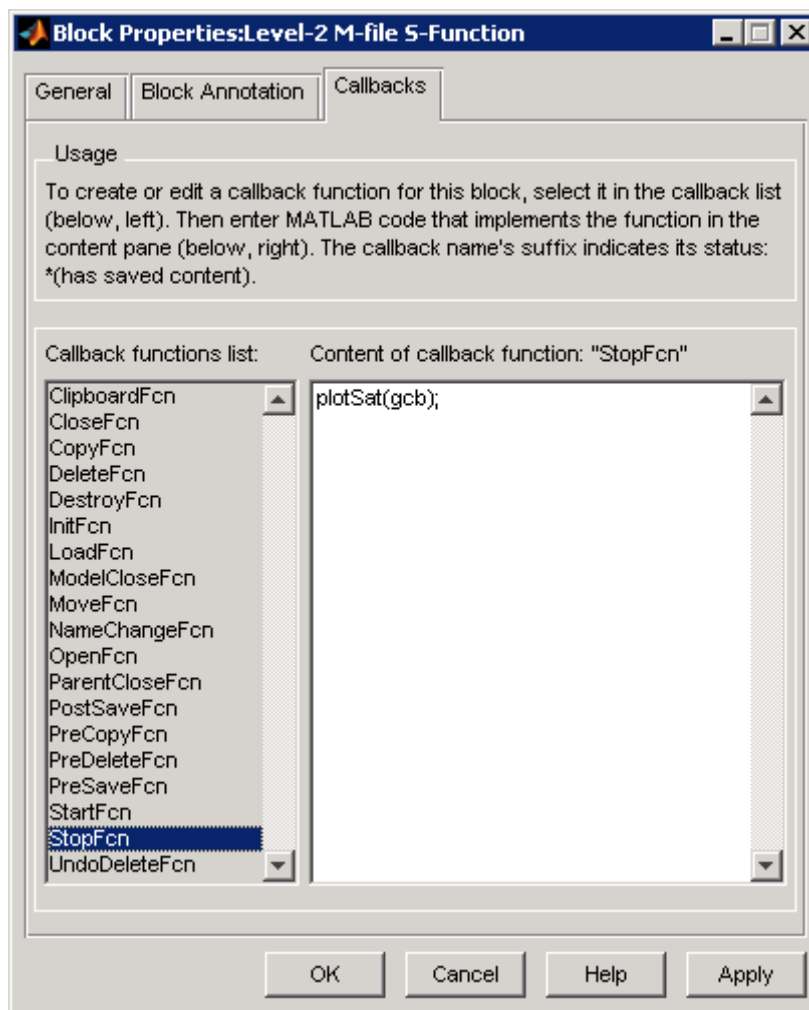
Custom Block Examples

In this section...
"Creating Custom Blocks from Masked Library Blocks" on page 21-44
"Creating Custom Blocks from MATLAB® Functions" on page 21-44
"Creating Custom Blocks from S-Functions" on page 21-45

Creating Custom Blocks from Masked Library Blocks

The Additional Math and Discrete Simulink® library is a group of custom blocks created by extending the functionality of built-in Simulink blocks. The Additional Discrete library contains a number of masked blocks that extend the functionality of the standard Unit Delay block. See Chapter 7, "Working with Block Libraries" for more general information on Simulink libraries.

Creating Custom Blocks from MATLAB® Functions

The Simulink product provides a number of demonstrations that show how to incorporate MATLAB® functions into a custom block.

- The Single Hydraulic Cylinder Simulation, `sldemo_hydcyl.mdl`, uses a Fcn block to model the control valve flow. In addition, the Control Valve Flow block is a library link to one of a number of custom blocks in the library `hydlib.mdl`.

- The Radar Tracking Model, `sldemo_radar.mdl`, uses a MATLAB Fcn block to model an extended Kalman filter. The M-file `aero_extkalman.m` implements the Kalman filter found inside the Radar Kalman Filter subsystem. In this example, the M-file requires three inputs, which are bundled together using a Mux block in the Simulink model.

- The Spiral Galaxy Formation demonstration, `sldemo_eml_galaxy.mdl`, uses several Embedded MATLAB™ Function blocks to construct two galaxy and calculate the effects of gravity as these two galaxies nearly collide. The demo also uses Embedded MATLAB Function blocks to plot the simulation results using a subset of MATLAB functions not supported for code generation. However, because these Embedded MATLAB Function blocks have no outputs, the Real-Time Workshop® product optimizes them away during code generation.

Creating Custom Blocks from S-Functions

The Simulink model sfundemos.mdl contains various examples of M-file and C MEX-file S-functions. For more information on writing M-file S-functions, see "Writing S-Functions in M". For more information on writing C MEX S-functions, see "Writing S-Functions in C". For a list of available S-function demos, see "S-Function Examples" in Writing S-Functions.

Using the Embedded MATLAB™ Function Block

Introduction to Embedded MATLAB™ Function Blocks

In this section...
"What Is an Embedded MATLAB™ Function Block?" on page 22-3
"Why Use Embedded MATLAB™ Function Blocks?" on page 22-5

What Is an Embedded MATLAB™ Function Block?

The Embedded MATLAB™ Function block allows you to add MATLAB® functions to Simulink® models for deployment to embedded processors. This capability is useful for coding algorithms that are better stated in the textual language of the MATLAB software than in the graphical language of the Simulink product. This block works with a subset of the MATLAB language called the Embedded MATLAB subset, which provides optimizations for generating efficient, production-quality C code for embedded applications. For more information, see "Working with the Embedded MATLAB Subset" in the MATLAB documentation. For more information on fixed-point support in MATLAB, refer to"Fixed-Point Embedded MATLAB Subset Features" in the Fixed-Point Toolbox™ documentation.

Here is an example of a Simulink model that contains an Embedded MATLAB Function block:
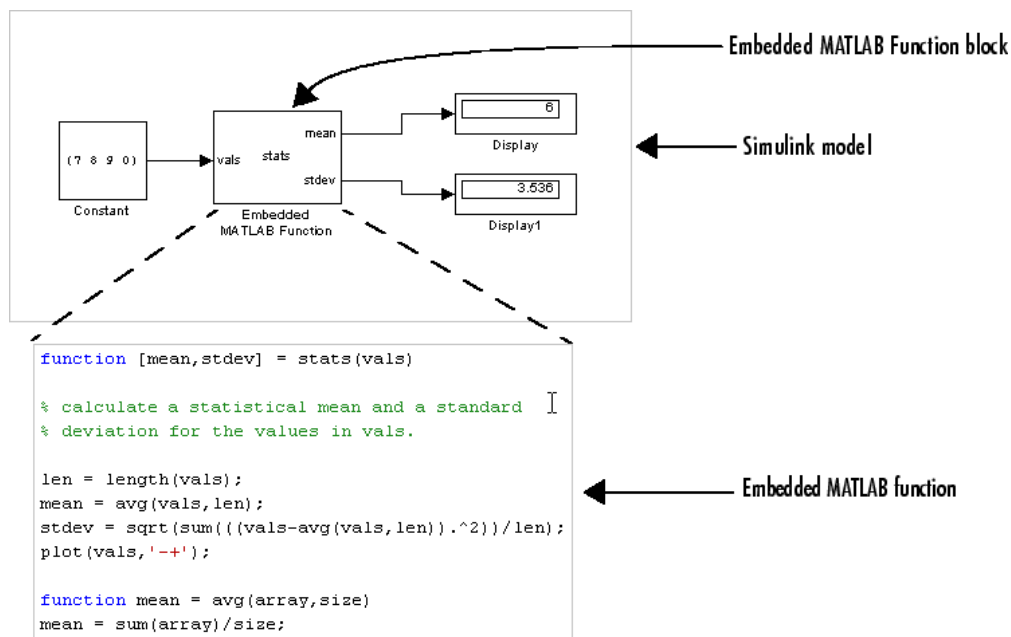
You will build this model in "Creating an Example Embedded MATLAB™ Function" on page 22-7 .

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:

```
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 By Assignment" 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 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 Function Library Reference" in the Embedded MATLAB documentation). With this support, you can use Real-Time Workshop to generate embeddable C code from Embedded MATLAB Function blocks that implements a variety of sophisticated mathematical applications. In this way, you can build executables that harness MATLAB functionality, but run outside the MATLAB environment.

- **Inherit properties from Simulink input and output signals** — By default, both the size and type of input and output signals to an Embedded MATLAB Function block are inherited from Simulink signals. You can also choose to specify the size and type of inputs and outputs explicitly in the Model Explorer or Ports and Data Manager (see "Ports and Data Manager" on page 22-37).

Creating an Example Embedded MATLAB™ Function

In this section...

Adding an Embedded MATLAB™ Function Block to a Model

Start by creating an empty model and filling it with an Embedded MATLAB Function block, and other blocks necessary to complete the model.

1 Create a new model with the Simulink® product and add an Embedded MATLAB Function block to it from the User-Defined Function library.
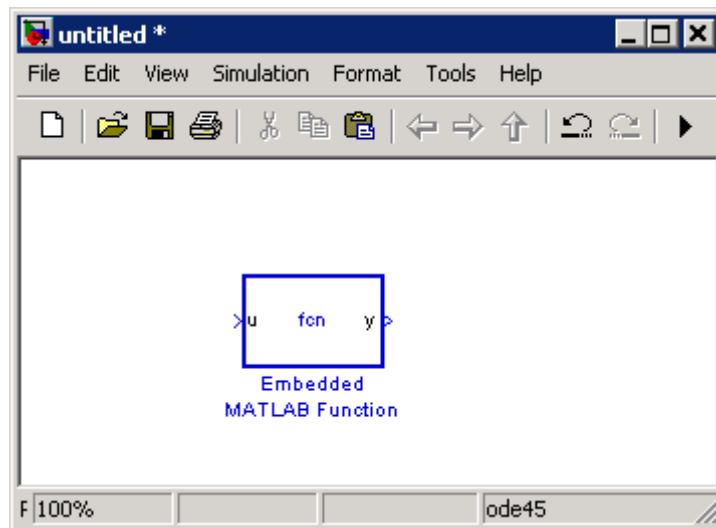
An Embedded MATLAB Function block has two names. The name in the middle of the block is the name of the function you build for the Embedded MATLAB Function block. Its name defaults to fcn. The name at the

bottom of the block is the name of the block itself. Its name defaults to `Embedded MATLAB Function`.

The default Embedded MATLAB Function block has an input port and an output port. The input port is associated with the input argument `u`, and the output port is associated with the output argument `y`.

2 Add the following Source and Sink blocks to the model:

- From the Sources library, add a Constant block to the left of the Embedded MATLAB Function block and set its value to the vector `[2 3 4 5]`.

- From the Sinks library, add two Display blocks to the right of the Embedded MATLAB Function block.

The model should now have the following appearance:
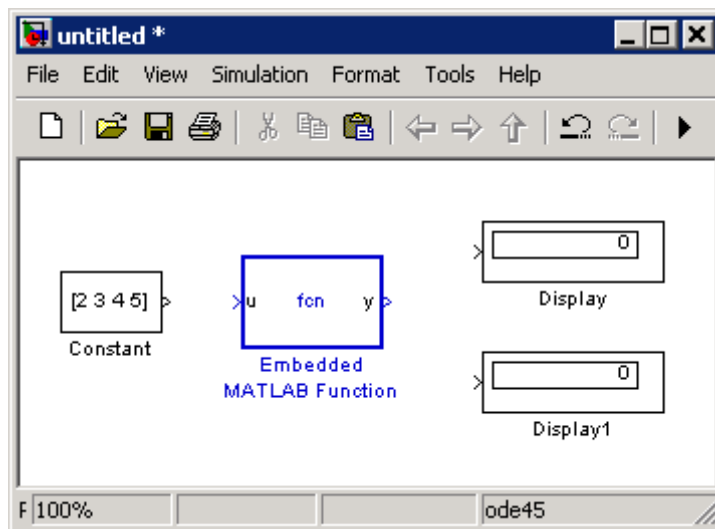
3 In the window displayed by the Simulink product, select **File > Save As** and save the model as `call_stats_block1`.

Programming the Embedded MATLAB™ Function

You create a model with an Embedded MATLAB Function block in "Creating an Example Embedded MATLAB™ Function" on page 22-7. 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 Open the `call_stats_block1` model that you save at the end of "Adding an Embedded MATLAB™ Function Block to a Model" on page 22-7. Double-click the Embedded MATLAB Function block fcn to open it for editing.

The Embedded MATLAB™ Editor appears.
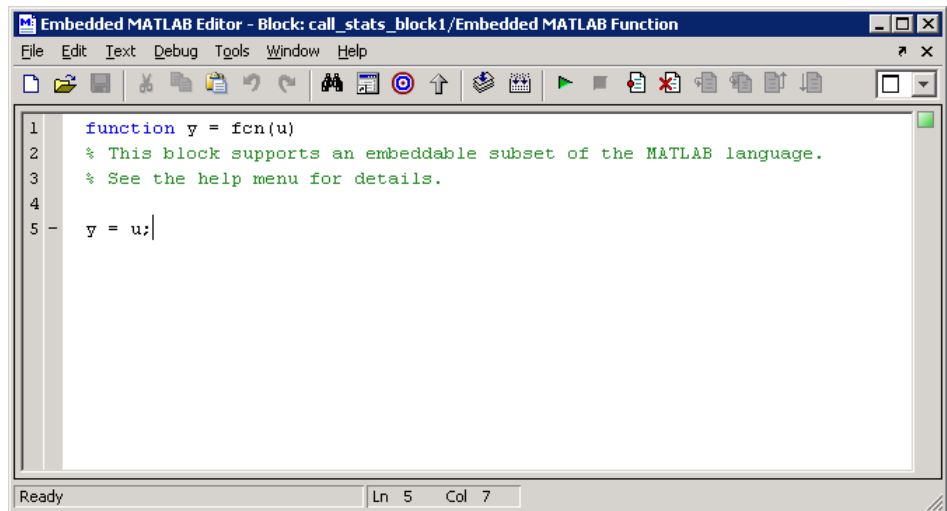
The Embedded MATLAB Editor window is titled with the syntax *<model name>/<Embedded MATLAB Function block name>* in its header. In this example, the model name is `call_stats_block1`, and the block name is `Embedded MATLAB Function`, the name that appears at the bottom of the Embedded MATLAB Function block.

Inside the Embedded MATLAB Editor is an edit window for editing the function that specifies the Embedded MATLAB Function block. A function

header with the function name `fcn` is at the top of the edit window. The header specifies an argument to the function, `u`, and a return value, `y`.

2 Edit the function header line with the return values, function name, and argument:

```
function [mean,stdev] = stats(vals)
```

The Embedded MATLAB function `stats` calculates a statistical mean and standard deviation for the values in the vector `vals`. The function header declares `vals` to be an argument to the `stats` function and `mean` and `stdev` to be return values from the function.

3 In the Embedded MATLAB Editor, select **File > Save Model As** and save the model as `call_stats_block2`.

Saving the model updates the model window, which now looks like this:
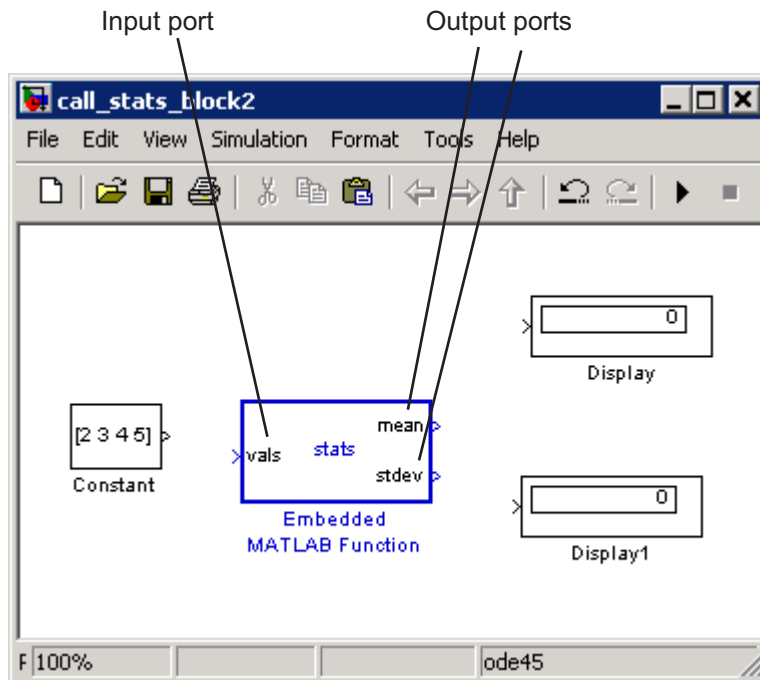
Changing the function header of the Embedded MATLAB Function block makes the following changes to the Embedded MATLAB Function block in the Simulink model:

- The function name in the middle of the block changes to stats.

- The argument vals appears as an input port to the block.

- The return values mean and stdev appear as output ports to the block.

4 Complete the connections to the Embedded MATLAB Function block as shown.
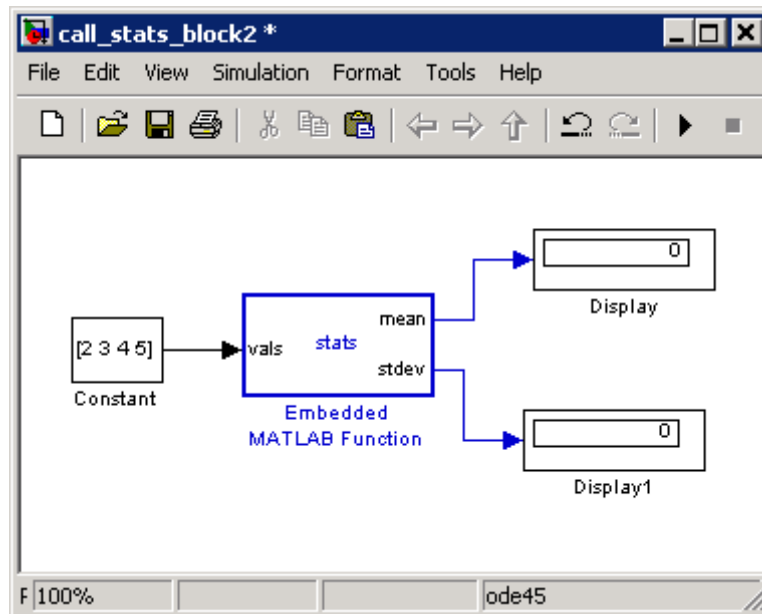
5 In the Embedded MATLAB Editor, enter a line space after the function header and replace the default comment line with the following comment lines:

```
% 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®.

22-11

6 Enter a line space after the comments and replace the default function line `y = u;` with the following:

```
len = length(vals);
```

The function `length` is an example of a built-in function supported by the runtime function library for Embedded MATLAB Function blocks. This `length` works just like the MATLAB function `length`. It returns the vector length of its argument `vals`. However, when you simulate this model, C code is generated for this function in the simulation application. Callable functions supported for Embedded MATLAB Function blocks are listed in the topic "Embedded MATLAB Function Library Reference" in the Embedded MATLAB documentation.

The variable `len` is a local variable that is automatically typed as a scalar `double` because the Embedded MATLAB runtime library function, `length`, returns a scalar of type `double`. If you want, you can declare `len` to have a different type and size by changing the way you declare it in the function. See "Creating Local Variables By Assignment" 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 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:

10 Save the model 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 Open the call_stats_block2 model that you save at the end of "Programming the Embedded MATLAB™ Function" on page 22-9. Double-click its Embedded MATLAB Function block stats to open it for editing.

2 In the Embedded MATLAB Editor, click the **Build** icon ![icon] to compile and build the example model.

If errors are found, the **Diagnostics Manager** window lists them. If no errors are found, the **Diagnostics Manager** window displays log messages that depends on the contents of your Embedded MATLAB Function block. The final log message is

```
Model Compilation for model_name successful.
```

The following is an example of what happens when an error is found.

1 Change the subfunction avg to a fictitious subfunction aug and then compile to see the following messages in the **Diagnostics Manager** window:
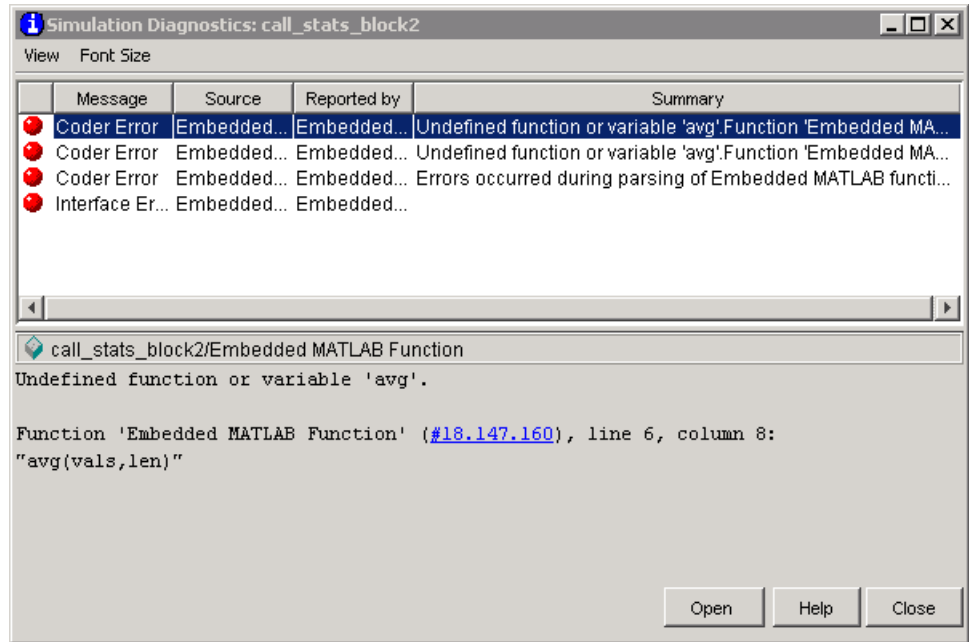
Each detected error appears with a red button.

2 Click the first error line to display its diagnostic message in the bottom half of the error window.

3 In the diagnostic message for the selected error, click the blue link to find the offending code:
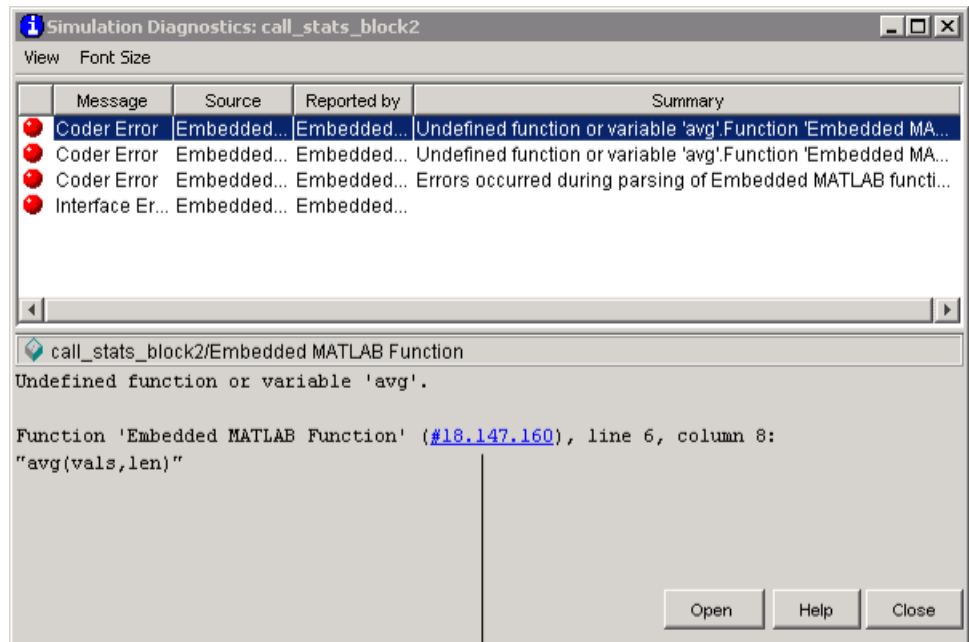
Click to display the offending code
in the Embedded MATLAB Editor

The offending line appears highlighted in the Embedded MATLAB Editor:

4 Correct the error by changing `aug` back to `avg` and recompile. No errors are found and the compile completes successfully.

Defining Inputs and Outputs

In the `stats` function header for the Embedded MATLAB Function block you define in "Programming the Embedded MATLAB™ Function" on page 22-9, 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 signals attached to their ports. In this topic, you examine input and output data for the Embedded MATLAB Function block to verify that it inherits the correct type and size.

1 Open the `call_stats_block2` model that you save at the end of "Programming the Embedded MATLAB™ Function" on page 22-9. Double-click the Embedded MATLAB Function block `stats` to open it for editing.

2 In the Embedded MATLAB Editor, select **Tools > Model Explorer**.

The Model Explorer window opens:
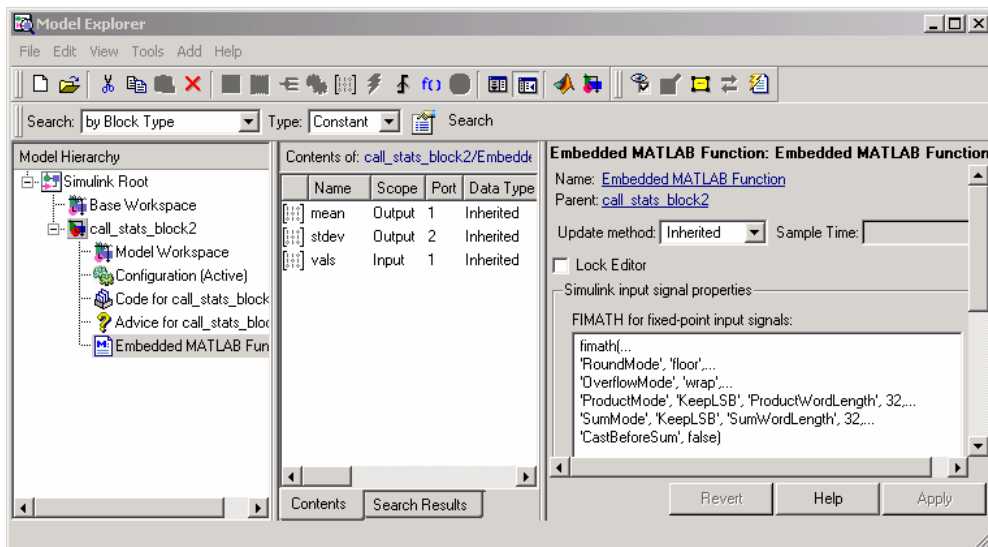
You can use the Model Explorer to define arguments for Embedded MATLAB Function blocks. Notice that the Embedded MATLAB Function block Embedded MATLAB is highlighted in the left **Model Hierarchy** pane.

The **Contents** pane displays the argument vals and the return values mean and stdev that you have already created for the Embedded MATLAB Function block. Notice that vals is assigned a **Scope** of Input, which is short for **Input from Simulink**. mean and stdev are assigned the **Scope** of Output, which is short for **Output to Simulink**.

You can also use the Ports and Data Manager to define arguments for Embedded MATLAB Function blocks (see "Ports and Data Manager" on page 22-37).

3 In the **Contents** pane of the Model Explorer window, click anywhere in the row for vals to highlight it:

Inherited size | Inherited type

Inherited complexity
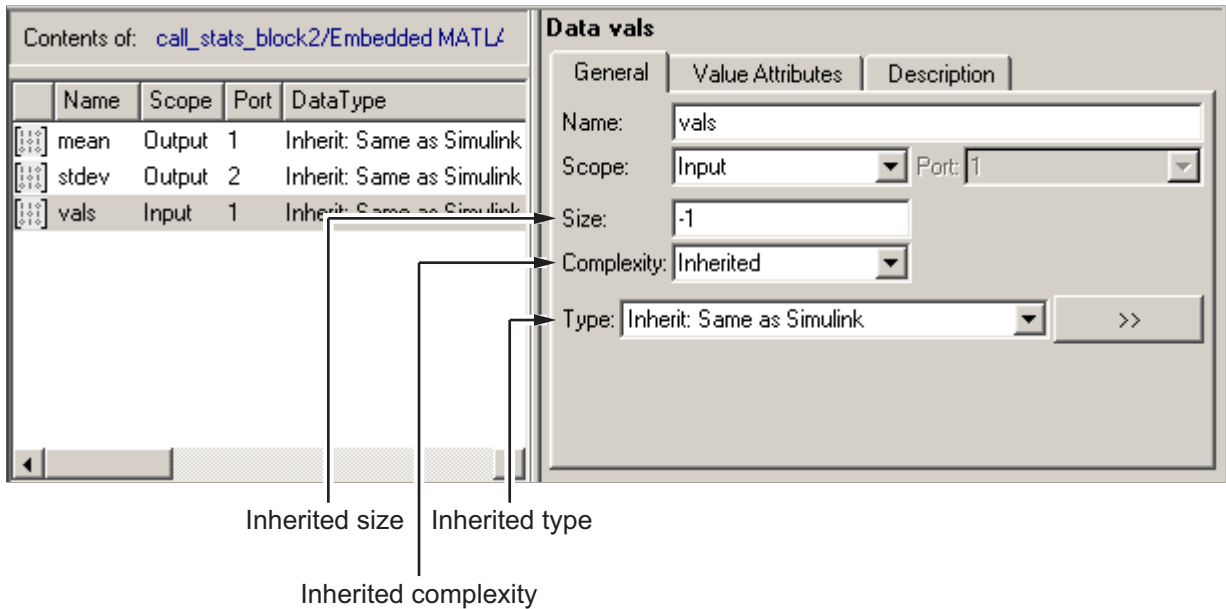
The right pane displays the **Data** properties dialog box for vals. By default, the type, size, and complexity of input and output arguments are inherited from the signals attached to each input or output port. Inheritance is specified by setting **Size** to -1**, Complexity** to Inherited, and **Type** to Inherit: Same as Simulink.

The actual inherited values for size and type are set during compilation of the model, and are reported in the **Compiled Type** and **Compiled Size** columns of the **Contents** pane.

You can specify the type of an input or output argument by selecting a type in the **Type** field of the **Data** properties dialog box, for example, double. You can also specify the size of an input or output argument by entering an expression in the **Size** field of the **Data** properties dialog box for the argument. For example, you can enter [2 3] in the **Size** field to size vals as a 2-by-3 matrix. See "Typing Function Arguments" on page 22-61 and

"Sizing Function Arguments" on page 22-71 for more information on the expressions that you can enter for type and size.

Note The default first index for any arrays that you add to an Embedded MATLAB Function block function is 1, just as it would be in MATLAB.

Debugging an Embedded MATLAB™ Function Block

In this section...

How Debugging Affects Simulation Speed

Debugging an Embedded MATLAB™ function slows simulation speed. If your model has many Embedded MATLAB Function blocks and debugging is enabled, the simulation speed is much slower than when debugging is disabled. For maximum simulation speed, disable debugging as described in "Enabling and Disabling Debugging" on page 22-21.

Enabling and Disabling Debugging

There are two levels of debugging available when using Embedded MATLAB Function blocks, model level debugging and block level debugging.

Debugging is enabled for all Embedded MATLAB functions by default, except for Embedded MATLAB functions in Stateflow®. Debugging for Embedded MATLAB functions in Stateflow is controlled in the **Simulation Target** dialog.

Disable debugging for an entire model by clearing the **Enable debugging/animation** check box in the **Simulation Target** dialog. Disable debugging for an individual Embedded MATLAB Function block by clicking **Enable Debugging** in the Embedded MATLAB Editor Debug menu. If **Enable Debugging** is unavailable, then the **Simulation Target** dialog is controlling debugging.

Debugging the Function in Simulation

In "Creating an Example Embedded MATLAB™ Function" on page 22-7, you create and specify an example model with an Embedded MATLAB Function block. You use this block to specify an Embedded MATLAB function stats that calculates the mean and standard deviation for a set of input values. In this section, you debug stats in the example model.

You can debug your Embedded MATLAB Function block just like you can debug a function in MATLAB®. In simulation, you test your Embedded MATLAB functions for runtime errors with tools similar to the MATLAB debugging tools. See "Watching with the Command Line Debugger" on page 22-31 and "Debugging Tools" on page 22-34 for more information.

When you start simulation of your model, the Simulink® software checks to see if the Embedded MATLAB Function block has been built since creation, or since a change has been made to the block. If not, it performs the build described in "Checking the Function for Errors" on page 22-14. If no diagnostic errors are found, the simulation of your model begins.

Use the following procedure to debug the stats Embedded MATLAB function during simulation of the model:

1 Open the call_stats_block2 model that you save at the end of "Programming the Embedded MATLAB™ Function" on page 22-9. 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.

Breakpoint indicator
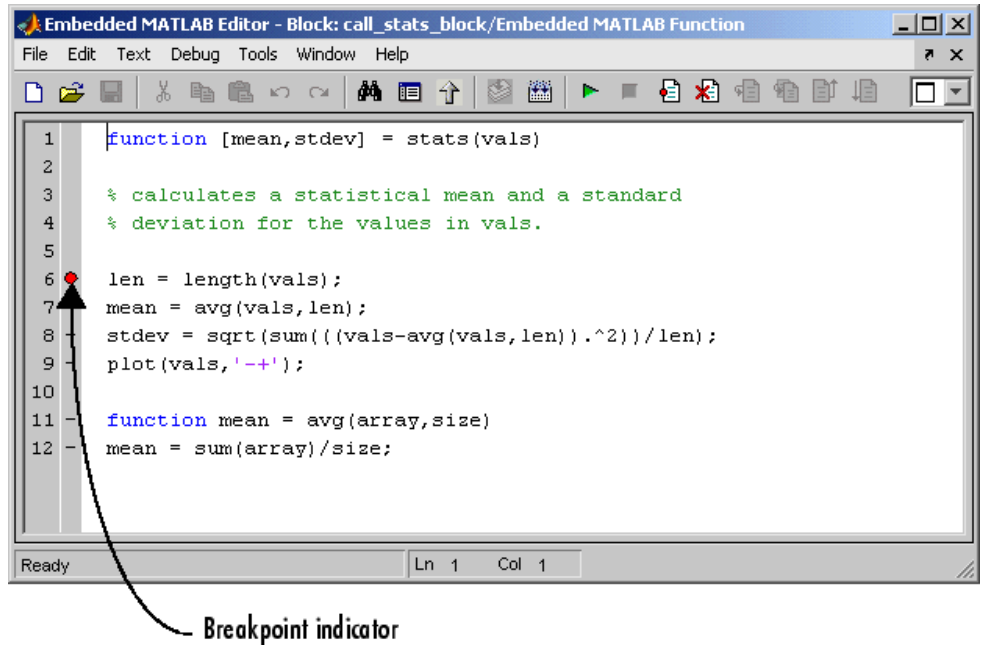
A small red ball appears in the margin of line 6, indicating that you have set a breakpoint.

3 Begin simulating the model:

If you get any errors or warnings, make corrections before you try to simulate again. Otherwise, simulation pauses when execution reaches the breakpoint you set. This is indicated by a small green arrow in the left margin, as shown.
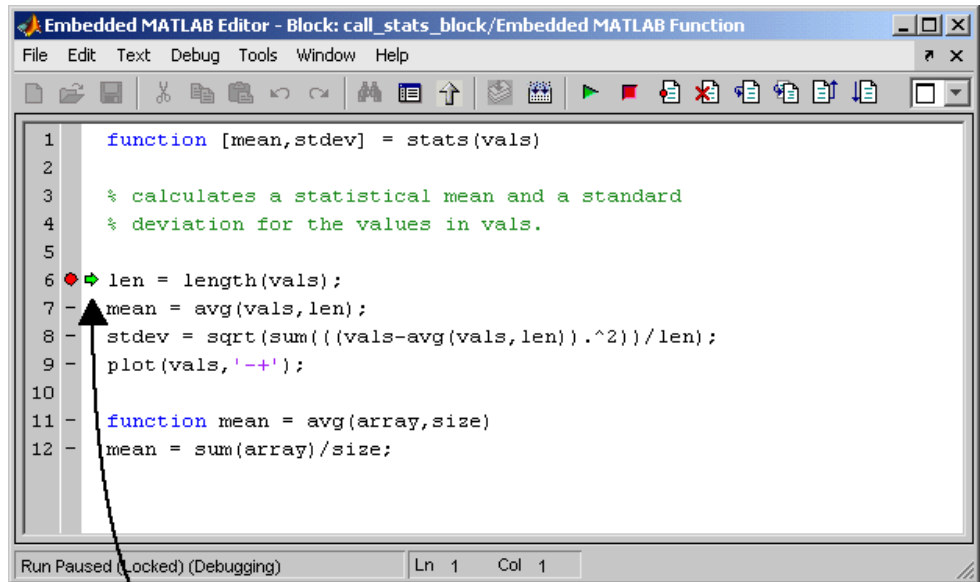
Execution pauses prior to next step
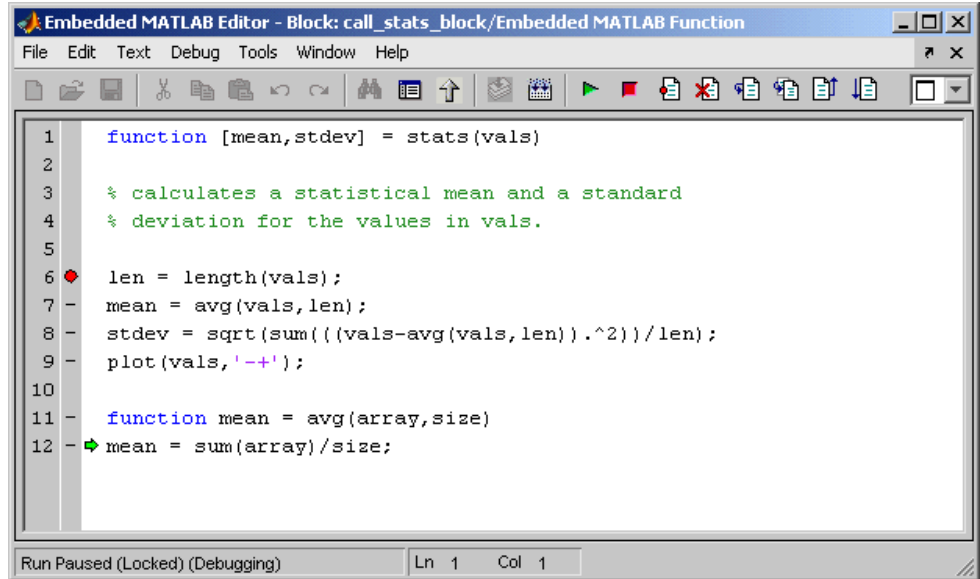
4 Click the **Step** icon to advance execution:

The execution arrow advances to line 7 of stats. 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 [icon].

Execution advances to enter the subfunction avg:

Once you are in a subfunction, you can use the **Step** or **Step In** icons to advance execution. If the subfunction calls another subfunction, use the **Step In** icon to enter it. If you want to execute the remaining lines of the subfunction, click the **Step Out** icon [icon].

6 Click the **Step** icon to execute the only line in the subfunction `avg`.
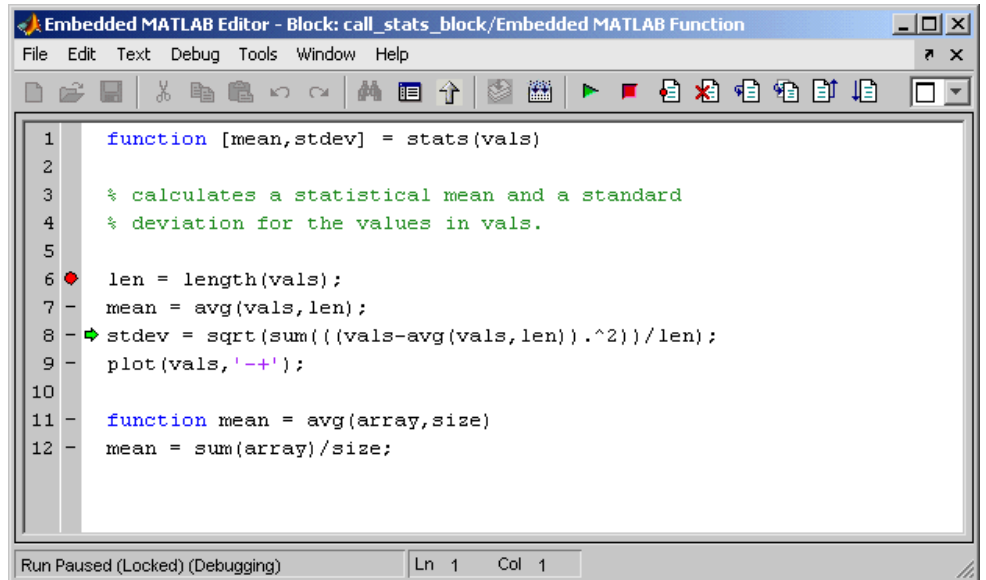
The subfunction `avg` finishes its execution, and you see a green arrow pointing down under its last line as shown.

Subfunction completed

7 Click the **Step** icon to return to the function `stats`.

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.
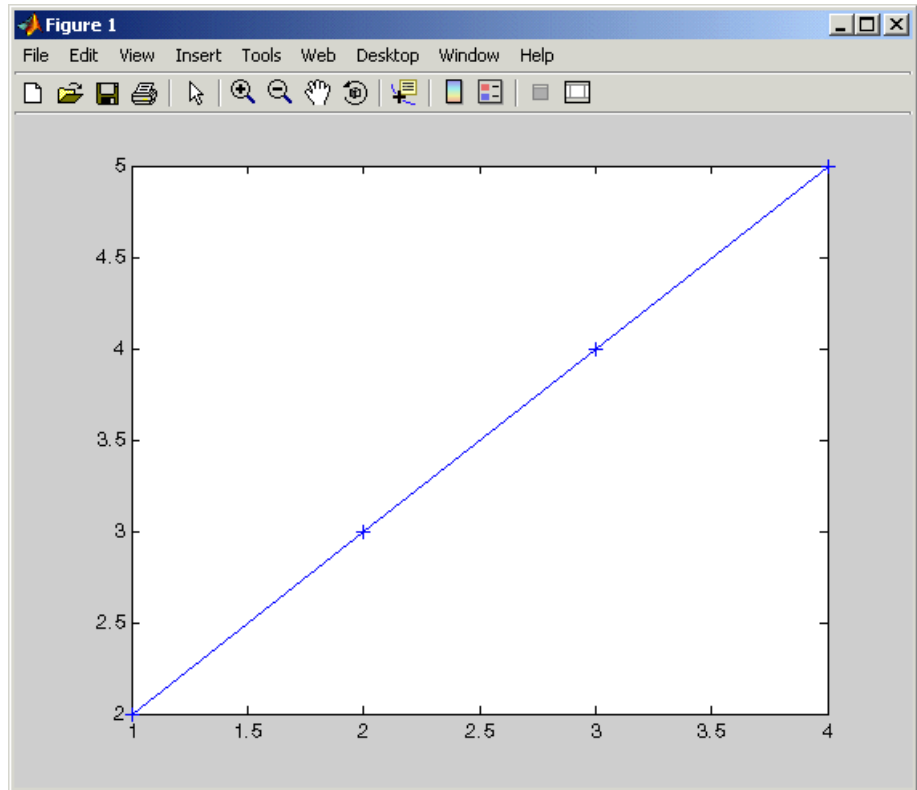
In the Embedded MATLAB Editor, a green arrow points down under line 9, indicating the completion of the function stats.
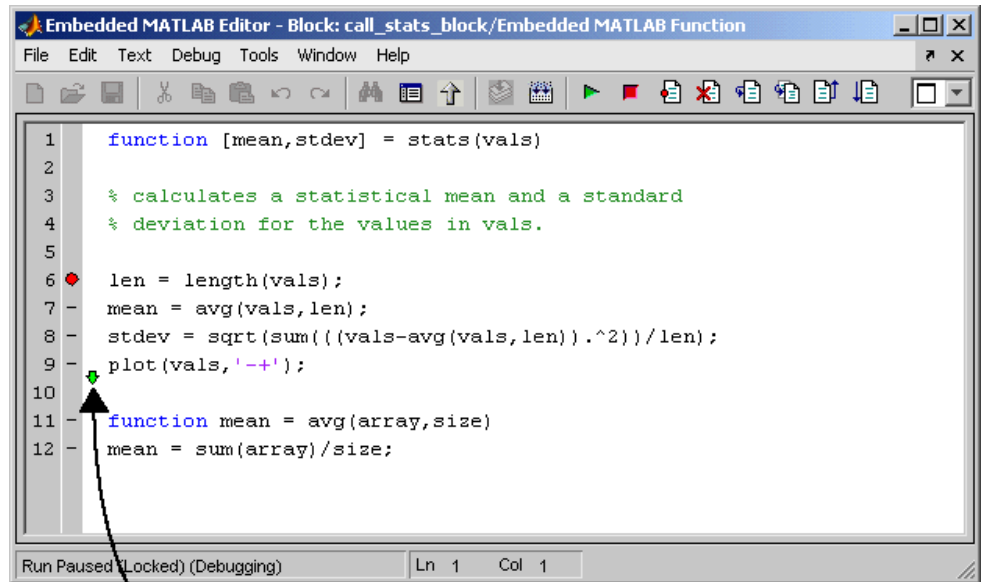
Function completed arrow

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.

The computed values of mean and stdev now appear in the Display blocks.
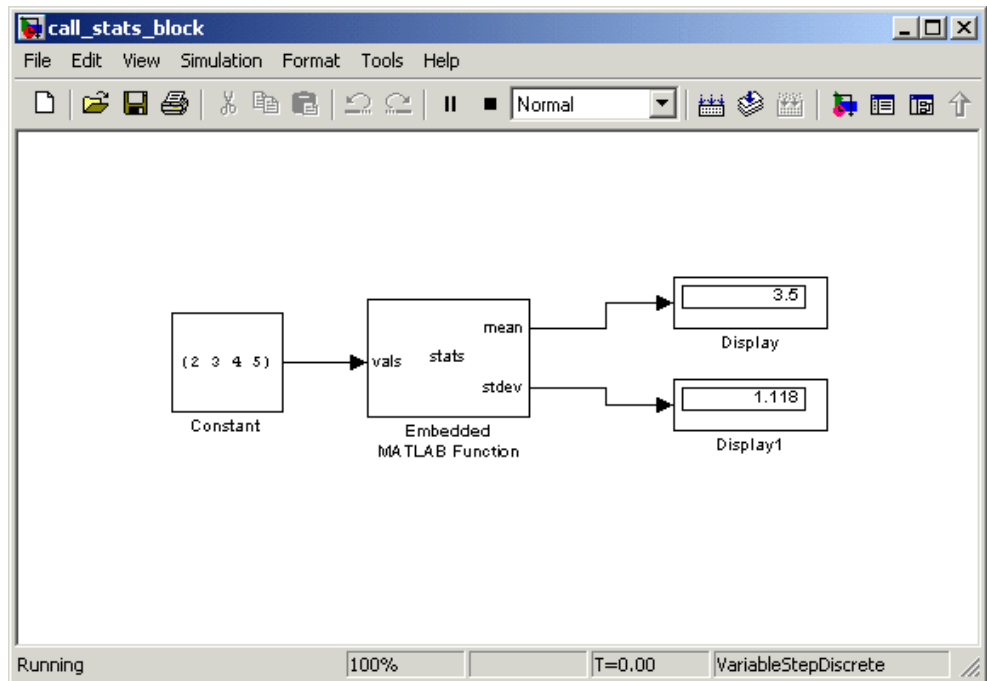
10 In the Embedded MATLAB Editor, click the **Exit Debug Mode** icon ■ to stop simulation.

For more information, see "How Exiting Debug Mode Affects Simulation" on page 22-33.

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. The value of `len` appears adjacent to the cursor, as shown:

Display of value for variable `len`
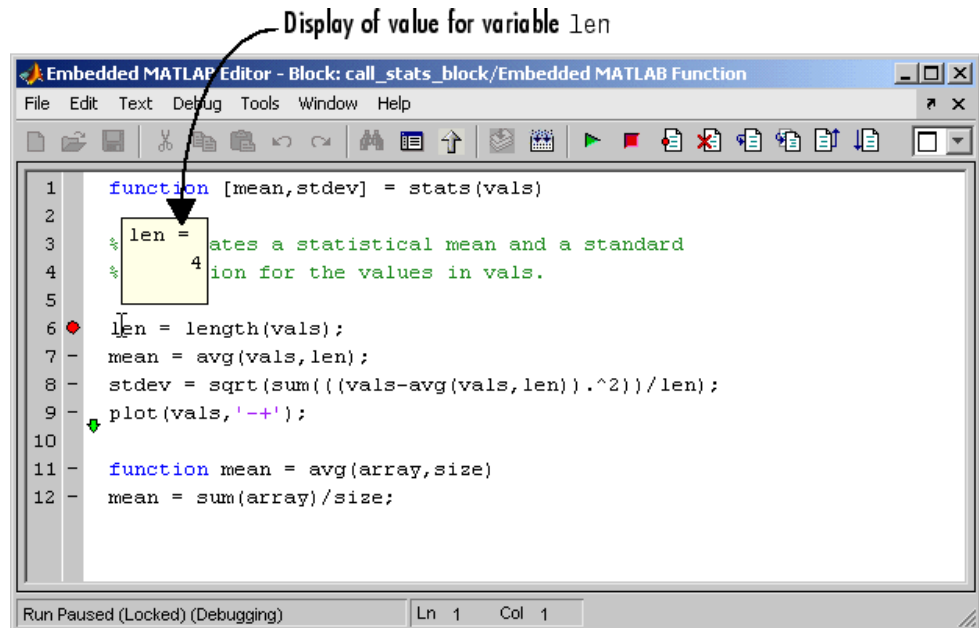
You can display the value for any variable in the Embedded MATLAB function in this way, no matter where it appears in the function.

Watching with the Command Line Debugger

You can report the values for an Embedded MATLAB function variable with the Command Line Debugger utility in the MATLAB window during simulation. When you reach a breakpoint, the Command Line Debugger prompt, `debug>>`, appears. At this prompt, you can see the value of a variable defined for the Embedded MATLAB Function block by entering its name:

```
debug>> stdev

 1.1180

debug>>
```

The Command Line Debugger also provides the following commands during simulation:

Command	Description
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.

Checking for Data Range Violations

When you enable debugging, Embedded MATLAB Function blocks automatically check input and output data for data range violations when the values enter or leave the blocks.

Specifying a Range

To specify a range for input and output data, follow these steps:

1 In the Model Explorer, select the input or output of interest in the Embedded MATLAB Function block.

The data properties dialog box opens in the Dialog pane of the Model Explorer.

2 In the data properties dialog box, select the Value Attributes tab and enter a limit range, as described in "Setting Value Attributes Properties" on page 22-51 .

How Exiting Debug Mode Affects Simulation

The behavior of the **Exit Debug Mode** option depends on the run-time context:

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

Debugging Tools

Use the following tools during an Embedded MATLAB function debugging session:

Tool Button	Description
● **Breakpoint Indicator**	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 existing 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 **F10**, 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 the first line of the subfunction. You can use this tool only after execution has stopped at a breakpoint. Alternatively, press **Shift+F11**.
Step Out	Step out of line-by-line execution of the current function or subfunction. If in a subfunction, the debugger continues to the line following the call to this subfunction. You can use this tool only after execution has stopped at a breakpoint. Alternatively, press **F10**.
Continue Debugging	Continue debugging after a pause, such as stopping at a breakpoint. You can use this tool only after execution has stopped at a breakpoint. Alternatively, press **F10**.

The Embedded MATLAB™ Function Editor

Customizing the Embedded MATLAB™ Editor

Use the toolbar icons to customize the appearance of the Embedded MATLAB™ Editor in the same manner as the MATLAB® editor. See "Arranging the Desktop" in the MATLAB documentation.

Embedded MATLAB™ Editor Tools

The following tools are specific to Embedded MATLAB:

Tool Button	Description
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 22-37). 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.
Simulation Target	Opens the Simulation Target dialog to enable debugging or include custom code. See "Enabling and Disabling Debugging" on page 22-21 for more information on debugging, and "Including C Functions in Simulation Targets" in the Embedded MATLAB documentation for more information on including custom code.

Tool Button	Description
⬆ **Go To Diagram**	Displays the Embedded MATLAB function in its native diagram without closing the Embedded MATLABB 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 22-17 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.

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 model hierarchy, use the Model Explorer, as described in "The Model Explorer" on page 13-2.

The Ports and Data Manager Dialog

The Ports and Data Manager dialog allows you to add and define data arguments, input triggers, and function call outputs for Embedded MATLAB Function blocks. Using this dialog, you can also modify properties for the Embedded MATLAB Function block and the objects it contains.

The dialog consists of two panes:

- 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.

Opening the Ports and Data Manager

To open the Ports and Data Manger 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 6-30).

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 model. These implicit events are the sample times (discrete or continuous) of the signals that provide inputs to the chart. If you define data inputs, the Embedded MATLAB Function block samples at the rate of the fastest data input. If you do not define data inputs, the Embedded MATLAB Function block samples as defined by its parent subsystem's execution behavior.
Discrete	The Embedded MATLAB Function block is sampled at the rate you specify as the block's **Sample Time** property. An implicit event is generated at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the model can have different sample times.
Continuous	The Simulink software wakes up (samples) the Embedded MATLAB Function block at each step in the simulation, as well as at intermediate time points that can be requested by the solver. This method is consistent with the continuous method.

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:

Setting	Action When Overflow Occurs
Enabled (default)	Saturates an integer by setting it to the maximum positive or negative value allowed by the word size. Matches MATLAB behavior.
Disabled	In simulation mode, generates a runtime error. For Real-Time Workshop® code generation, the behavior depends on your C language compiler.

Note The **Saturate on Integer Overflow** option is relevant only for integer arithmetic. It has no effect on fixed point or double-precision arithmetic.

When you enable **Saturate on Integer Overflow**, the Embedded MATLAB Function block adds additional checks in the generated code to detect integer overflow or underflow. Therefore, it is more efficient to disable this option if you are sure that integer overflow and underflow will not occur in your Embedded MATLAB function code.

Even when you disable this option, the code for a simulation target checks for integer overflow and underflow. If either condition occurs, simulation stops and an error is generated. If you enabled debugging for the Embedded MATLAB Function block, the debugger displays the error and lets you examine the data.

If you have not enabled debugging for the Embedded MATLAB Function block, the block generates a runtime error, as in this example:
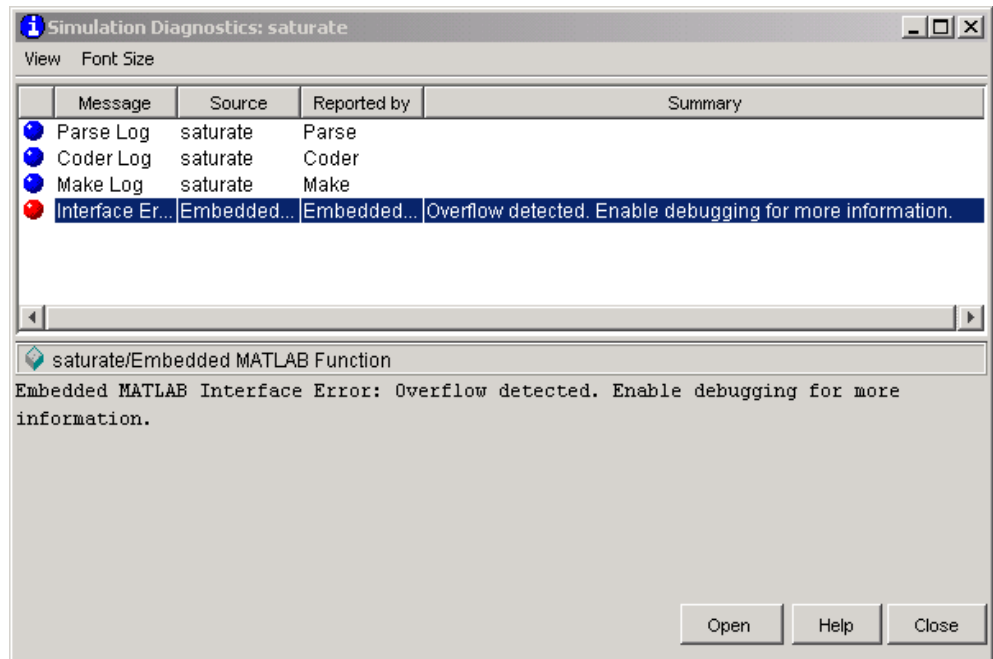
It is important to note that the code for an Real-Time Workshop target does *not* check for integer overflow or underflow and, therefore, may produce unpredictable results when **Saturate on Integer Overflow** is disabled. In this situation, it is recommended that you simulate first to test for overflow and underflow before generating the Real-Time Workshop target.

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 input signals:

Property	Description

FIMATH for fixed-point input signals	Defines the `fimath` object to be associated with 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 MATLABworkspace.
	
	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)
	``
	
	Note 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:
	
	``
	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 these inherited signal types as fi objects	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 documentation for the Embedded MATLAB Function block. To document an Embedded MATLAB Function block, set the **Document Link** property to a Web URL address or MATLAB expression that displays documentation in a suitable format (for example, an HTML file or text in the MATLAB Command Window). The Embedded MATLAB Function block evaluates the expression when you click the blue **Document Link** text.

Adding Data to an Embedded MATLAB™ Function Block

You can define input and output data arguments for an Embedded MATLAB Function block directly in the script, or by using the Ports and Data Manager or Model Explorer. You can use the Ports and Data Manager to add data arguments to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of data arguments in the block.

You can define data arguments for Embedded MATLAB Function blocks in the following methods:

Method	For Defining	Reference
Define data directly in the Embedded MATLAB function script	Input and output data	See "Defining Inputs and Outputs" on page 22-17.
Use the Ports and Data Manager	Input, output, and parameter data in the Embedded MATLAB Function block that is open and has focus	See "Defining Data in the Ports and Data Manager" on page 22-45.
Use the Model Explorer	Input, output, and parameter data in Embedded MATLAB Function blocks at all levels of the model hierarchy	See "Defining Data in the Model Explorer" on page 22-46

Defining Data in the Ports and Data Manager. To add a data argument and modify its properties, follow these steps:

1 In the Ports and Data Manager, click the **Add Data** icon [⊞].

The Ports and Data Manager adds a default definition of the data to the Embedded MATLAB Function block and displays the new data argument.
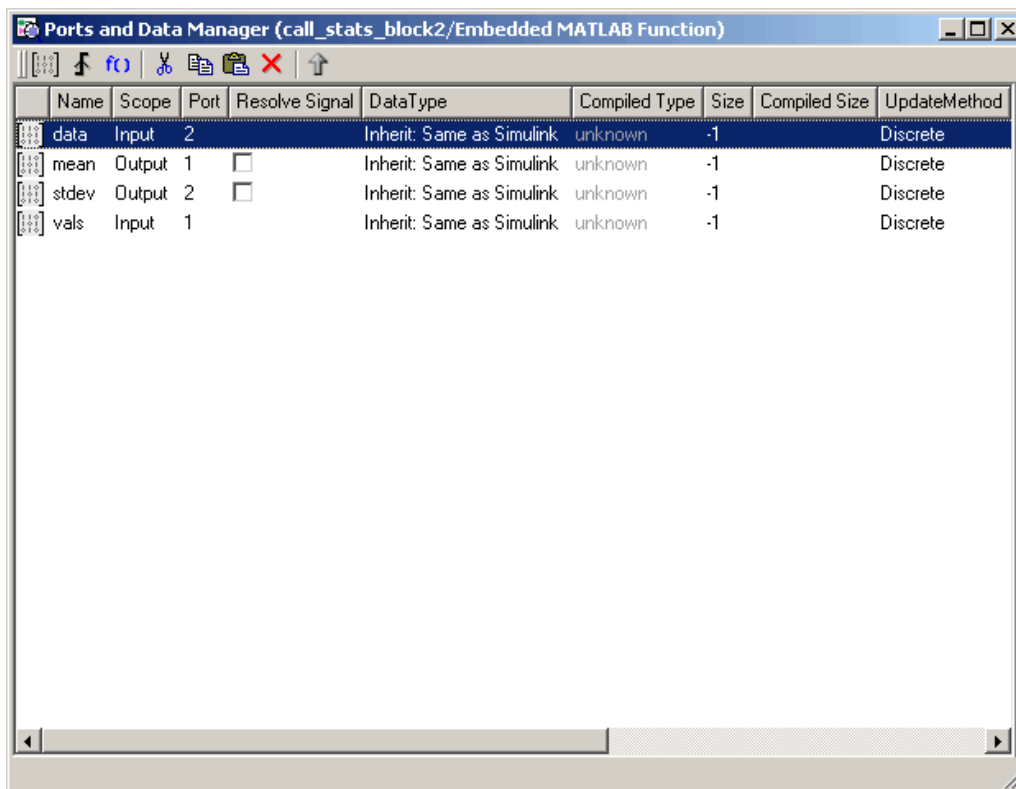
2 Select the row containing the new data argument.

3 Select the data property you want to modify, and specify a new value, as in this example:

4 Repeat step 3 to specify values for other data properties.

Defining Data in the Model Explorer. The Data properties dialog in the Model Explorer allows you to set and modify the properties of data arguments in Embedded MATLAB Function blocks. Properties vary according to the scope and type of the data object. Therefore, the Data properties dialog is dynamic, displaying only the property fields that are relevant for the data argument you are defining.

Open the Data properties dialog by selecting a data argument in the Contents pane.

The Data properties dialog provides a set of tabbed panes, as in this example:
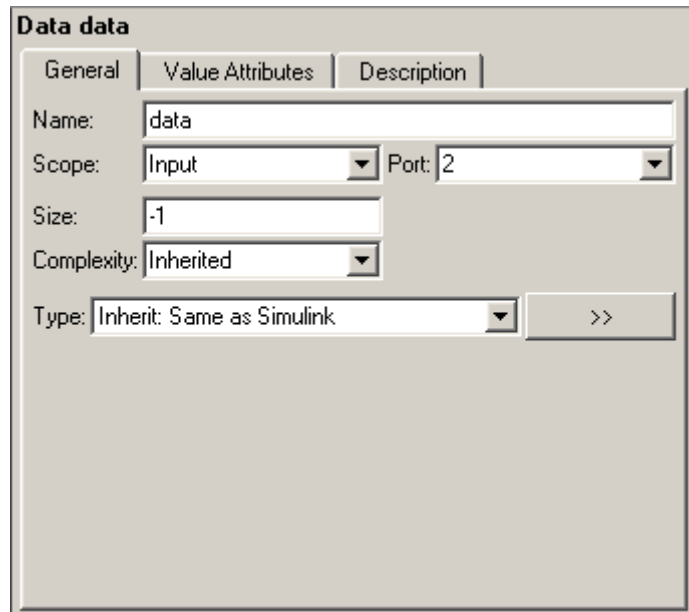
Each tab lets you define different features of your data argument:

- The **General** tab lets you define the scope, size, complexity, and type of the data argument. See "Setting General Properties" on page 22-47.

- The **Value Attributes** tab lets you set a limit range and save data argument values. See "Setting Value Attributes Properties" on page 22-51.

- The **Description** tab lets you enter a description and link to documentation about the data argument. See "Setting Description Properties" on page 22-53.

Setting General Properties. The General tab of the Data properties dialog looks like this:
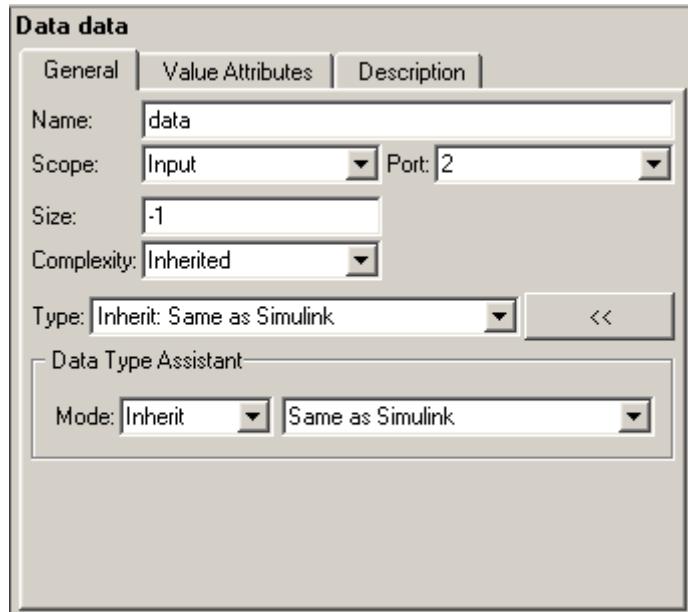
You can set the following properties in the General tab:

Property	Description
Name	Name of the data argument, following the same naming conventions used in MATLAB (see "Naming Variables" in the MATLAB documentation.

Property	Description
Scope	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, the variable closest to the block in the in the workspace hierarchy is used (see "Using Model Workspaces" on page 3-61). • Input— Data provided by the model via an input port to the Embedded MATLAB Function block. • Output— Data provided by the Embedded MATLAB Function block via an output port to the model. For more information, see "Defining Inputs and Outputs" on page 22-17 and "Parameter Arguments in Embedded MATLAB™ Functions" on page 22-74.
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 2-9). This property applies only to parameter data. You must clear this option if you want to use the parameter where Embedded MATLAB requires a constant expression, such as zeros (see entry for zeros in "Embedded MATLAB Function Library — Alphabetical List" in the Embedded MATLAB documentation).
Data must resolve to Simulink signal object	Specifies that the data argument must resolve to a Simulink signal object. This property applies only to output data. See "Resolving Symbols" on page 3-69 for more information.
Size	Size of the data argument. Size can be a scalar value or a MATLAB vector of values. Size defaults to –1, which means that it is inherited, as described in "Inheriting Argument Sizes from Simulink®" on page 22-71.For more details, see "Sizing Function Arguments" on page 22-71.

Property	Description
Complexity	Indicates real or complex data arguments. 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.
Sampling mode	Specifies how an output signal propagates through a model. This property applies only to data with scope equal to Output. You can set sampling mode to one of the following values: • Sample based: Propagate the signal sample by sample (default) • Frame based: Propagate the signal in batches of samples
Type	Type of data object. You can specify the data type by: • Selecting a built-in type from the **Type** drop down list. • Entering an expression in the **Type** field that evaluates to a data type (see "Working with Data Types" on page 10-2 in Using Simulink® on page 1). • Using the Data Type Assistant to specify a data **Mode**, then specifying the data type based on that mode. **Note** Click the Show data type assistant button `>>` to display the Data Type Assistant. For more information, see "Specifying Argument Types" on page 22-61.

Setting Value Attributes Properties. The **Value Attributes** tab of the **Data** properties dialog looks like this:
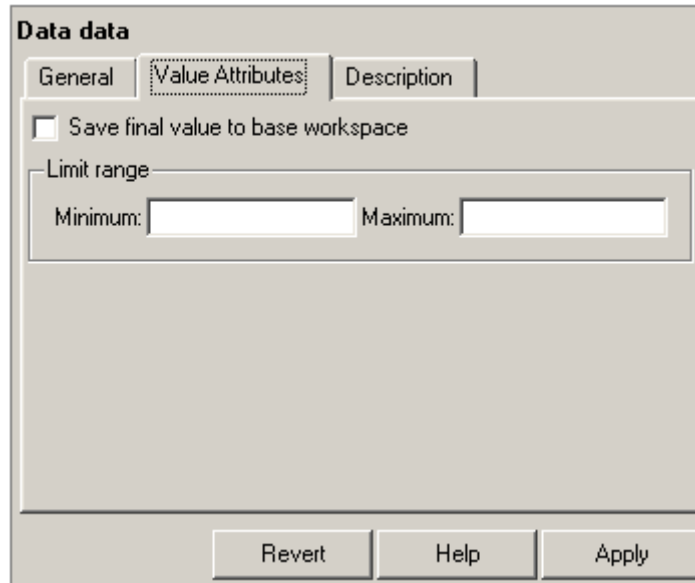
You can set the following properties on the **Value Attributes** tab:

Property	Description
Save final value to base workspace	The Embedded MATLAB Function block assigns the value of the data argument to a variable of the same name in the MATLAB base workspace at the end of simulation.
Limit range properties	Specify the range of acceptable values for input or output data. The Embedded MATLAB Function block uses this range to validate the input or output as it enters or leaves the block. You can enter an expression or parameter that evaluates to a numeric scalar value. • **Minimum** — The smallest value allowed for the data item during simulation. The default value is `-inf`. • **Maximum** — The largest value allowed for the data item during simulation. The default value is `inf`.

Setting Description Properties. The **Description** tab of the Data properties dialog looks like this:
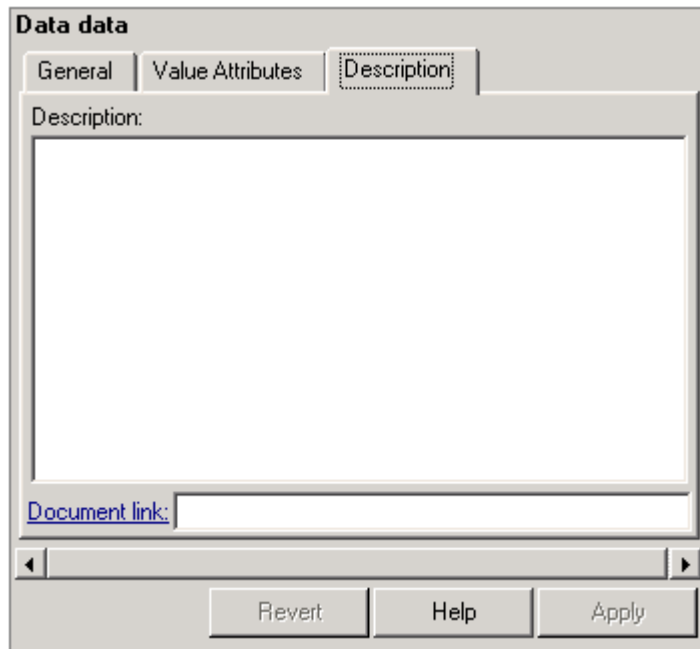
You can set the following properties on the Description tab:

Property	Description
Description	Description of the data argument.
Document link	Link to documentation for the data argument. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click the blue text, **Document link**, displayed at the bottom of the **Data** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

Adding Input Triggers to an Embedded MATLAB™ Function Block

You can use the Ports and Data Manager to add input triggers to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of input triggers in the block.

To add an input trigger and modify its properties, follow these steps:

1 In the Ports and Data Manager, click the **Add Input Trigger** icon ⬆.

The Ports and Data Manager adds a default definition of the new input trigger to the Embedded MATLAB Function block and displays the Trigger properties dialog.
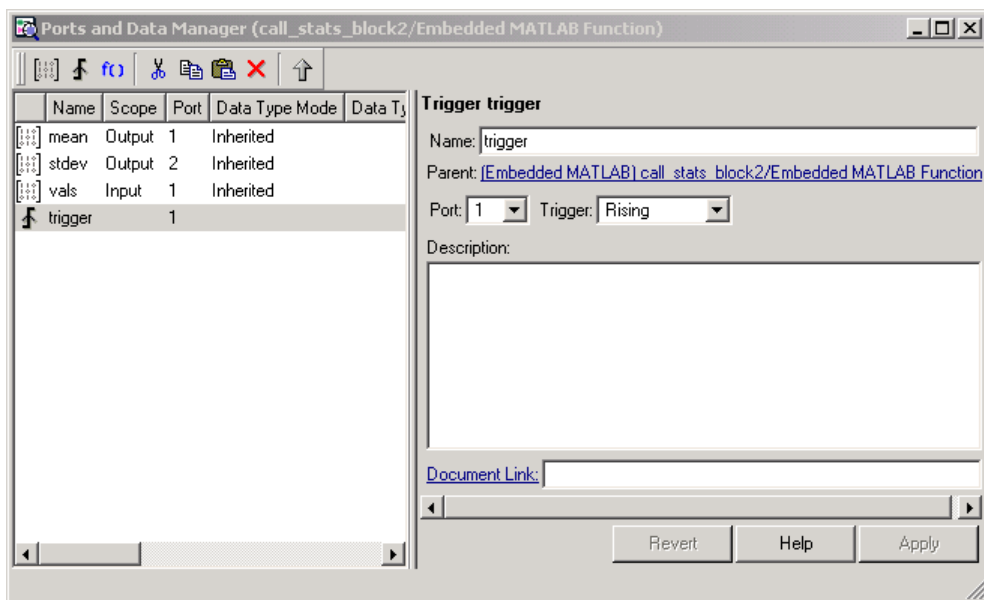
2 Modify properties for the new input trigger, using one of the following methods:

- In the Contents pane, select the row that contains the input trigger you want to modify and then edit the property of interest, as in this example:
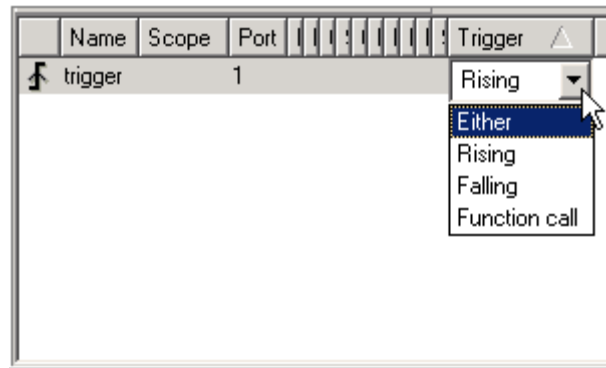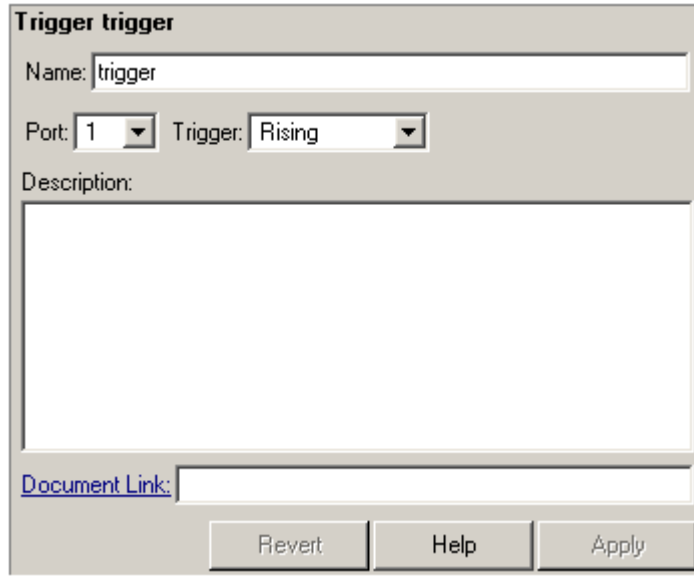
- Modify fields in the Trigger properties dialog, as described in "The Trigger Properties Dialog" on page 22-55.

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:

Setting Input Trigger Properties. You can set the following properties in the Trigger properties dialog:

Property	Description
Name	Name of the input trigger, following the same naming conventions used in MATLAB (see "Naming Variables" in the MATLAB documentation.
Port	Index of the port associated with the input trigger. The default value is 1.

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` (default) — Triggers execution of the Embedded MATLAB Function block when the control signal rises from a negative or zero value to a positive value (or zero if the initial value is negative). • `Falling`— Triggers execution of the Embedded MATLAB Function block when the control signal falls from a positive or zero value to a negative value (or zero if the initial value is positive). • `Either`— Triggers execution of the Embedded MATLAB Function block when the control signal is either rising or falling. • `Function call`— Triggers execution of the Embedded MATLAB Function block from a block that outputs function-call events, or from an S-function
Description	Description of the input trigger.
Document link	Link to documentation for the input trigger. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click the blue text that reads **Document link** displayed at the bottom of the **Trigger** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

Adding Function Call Outputs to an Embedded MATLAB™ Function Block

You can use the Ports and Data Manager to add and modify function call outputs to an Embedded MATLAB Function block that is open and has focus. You can also modify the properties of function call outputs in the block.

To add a function call output and modify its properties, follow these steps:

1 In the Ports and Data Manager, click the **Add Function Call Output** icon [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.
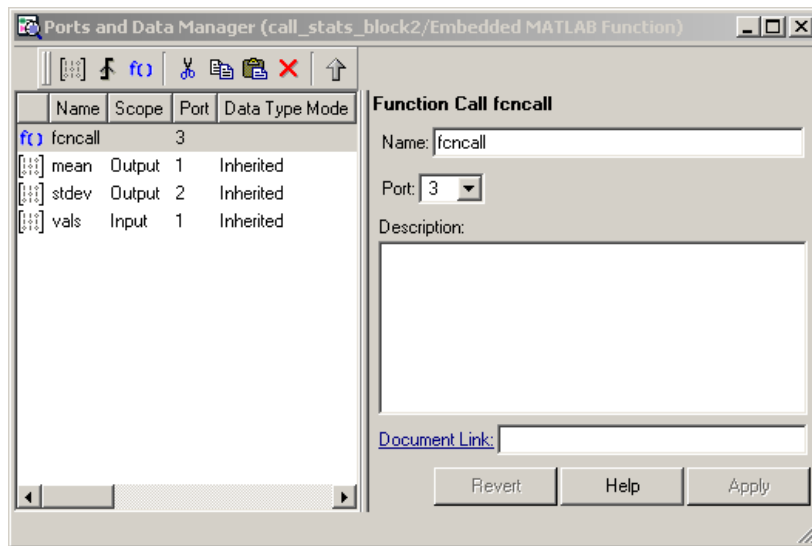
2 Modify properties for the new function call output, using one of the following methods:

- In the Contents pane, select the row that contains the function call output you want to modify and then edit the property of interest, as in this example:

- Modify fields in the Function Call properties dialog, as described in "The Function Call Properties Dialog" on page 22-59.

The Function Call Properties Dialog. The Function Call properties dialog in the Ports and Data Manager allows you to edit the properties of function call outputs in Embedded MATLAB Function blocks.

You can open the Function Call properties dialog 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:
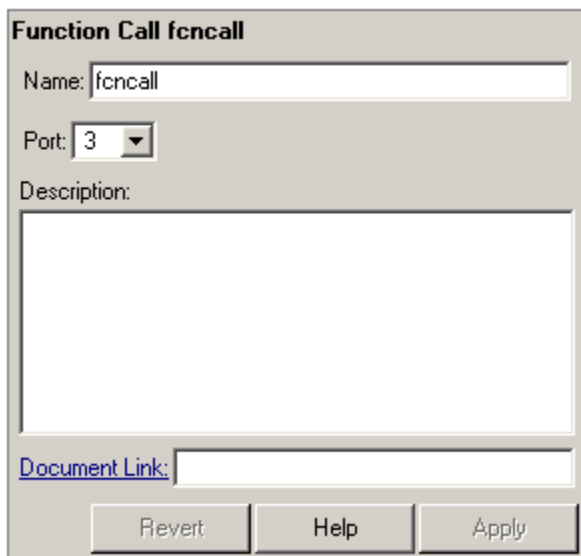
Setting Function Call Output Properties. You can set the following properties in the Function Call properties dialog:

Property	Description
Name	Name of the function call output, following the same naming conventions used in MATLAB (see "Naming Variables" in the MATLAB documentation.
Port	Index of the port associated with the function call output. The default value is 3.
Description	Description of the function call output.
Document link	Link to documentation for the function call output. You can enter a Web URL address or a MATLAB command that displays documentation in a suitable format, such as an HTML file or text in the MATLAB Command Window. When you click **Document link** displayed at the bottom of the **Function Call** properties dialog, the Embedded MATLAB Function block evaluates the link and displays the documentation.

Typing Function Arguments

About Function Arguments

You create function arguments for an Embedded MATLAB Function block by entering them in its function header in the Embedded MATLAB™ Editor. When you define arguments, the Simulink® software creates corresponding ports on the Embedded MATLAB Function block that you can attach to signals. You can select a *data type mode* for each argument that you define for an Embedded MATLAB Function block. Each data type mode presents its own set of options for selecting a *data type*.

By default, the data type mode for Embedded MATLAB function arguments is **Inherited**. This means that the function argument inherits its data type from the incoming or outgoing signal. To override the default type, you first choose a data type mode and then select a data type based on the mode. The following procedure describes how to use the Model Explorer to set data types for function arguments. You can also use the Ports and Data Manager tool (see "Ports and Data Manager" on page 22-37).

Specifying Argument Types

To specify the type of an Embedded MATLAB function argument:

1 From the Embedded MATLAB Editor, select **Tools > Model Explorer**.

Model Explorer appears with the Embedded MATLAB Function block highlighted in the **Model Hierarchy** pane.
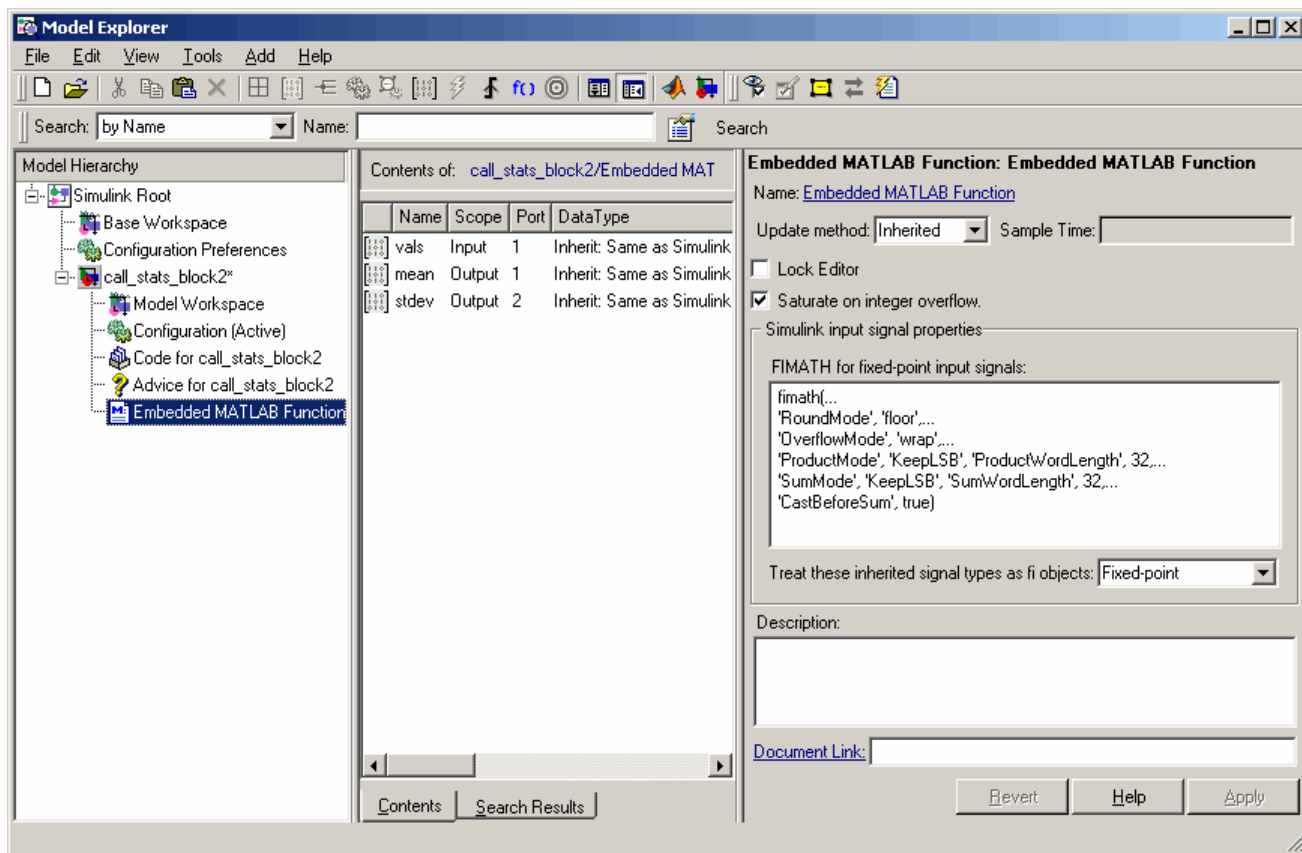
2 In the **Contents** pane (in the middle), click the row containing the argument of interest.

3 In the **Data** properties dialog (on the right), click the Show data type assistant button [>>] to display the Data Type Assistant. Then, choose an option from the **Mode** drop-down menu, as shown:
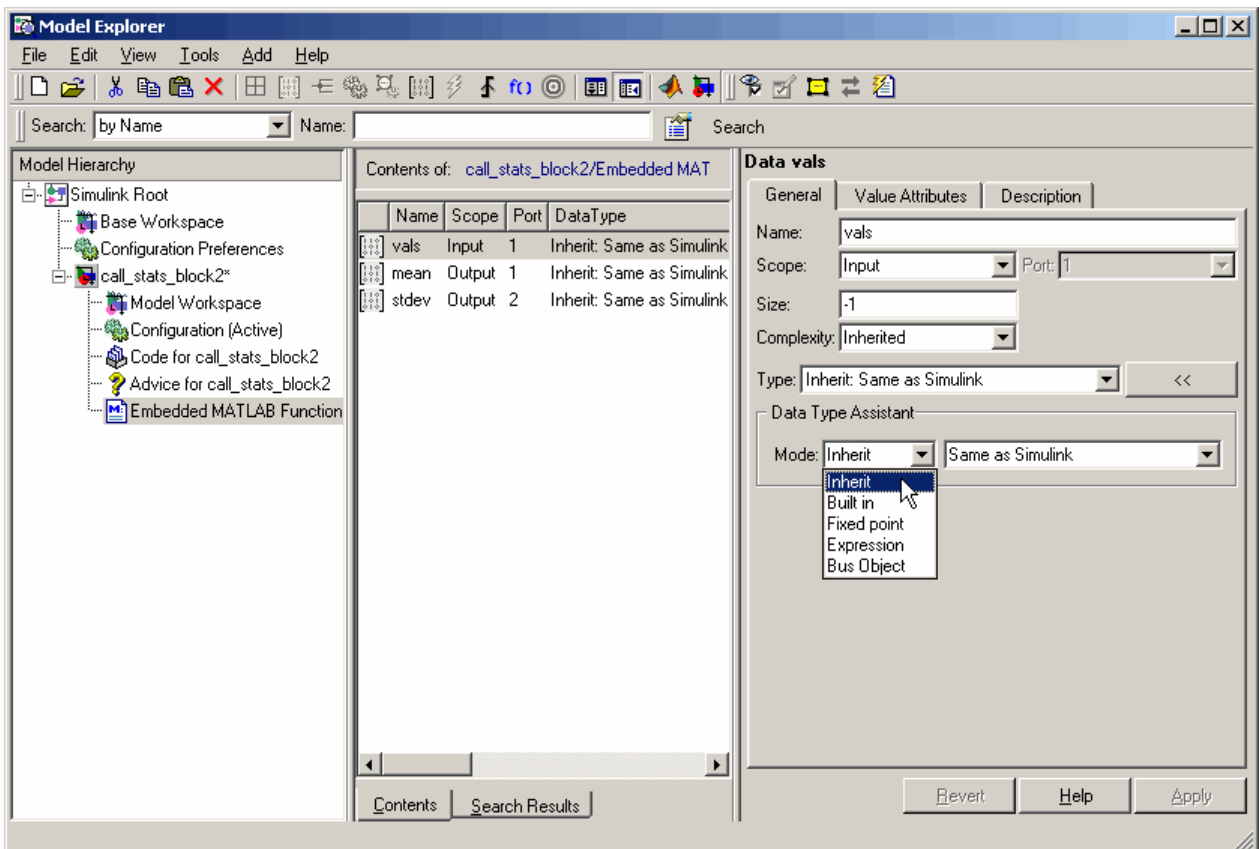
The **Data** properties dialog changes dynamically to display additional fields for specifying the data type associated with the mode.

4 Based on the mode you select, specify a desired data type:

Mode	What to Specify
Inherit (default)	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: • If scope is **Input**, data type is inherited from the input signal on the designated port. • If scope is **Output**, data type is inherited from the output signal on the designated port. • If scope is **Parameter**, data type is inherited from the associated parameter, which can be defined in the Simulink masked subsystem or the MATLAB® workspace. See "Inheriting Argument Data Types" on page 22-64.
Built in	In the **Data type** field, select from the drop-down list of supported data types, as described in "Built-In Data Types for Arguments" on page 22-66.
Fixed point	Specify the fixed-point data properties as described in "Specifying Simulink® Fixed Point™ Data Properties" on page 22-67.
Expression	Enter an expression that evaluates to a data type, as described in "Specifying Argument Types with Expressions" on page 22-66.
Bus Object	In the **Bus object** field, enter the name of a Simulink.Bus object to define the properties of an Embedded MATLAB structure. You must define the bus object in the base workspace. See "Working with Structures and Bus Signals" on page 22-77. **Note** You can click the **Edit** button to create or modify Simulink.Bus objects using the Simulink Bus Types Editor (see in the Simulink User's Guide).

Inheriting Argument Data Types

Embedded MATLAB function arguments can inherit their data types, including fixed point types, from the signals to which they are connected. Select the argument of interest in the Contents pane of the Model Explorer or Ports and Data Manager, and set data type mode using one of these methods:

- In the **Data** properties dialog, select `Inherit:  Same as Simulink` from the **Type** drop-down menu.

- In the Contents pane, set the **Data Type** column to **Inherit: Same as Simulink**.

See "Built-In Data Types for Arguments" on page 22-66 for a list of supported data types.

Note An argument can also inherit its complexity (whether its value is a real or complex number) from the signal that is connected to it. To inherit complexity, set the **Complexity** field on the **Data** properties dialog to **Inherited**.

Once you build the model, the **Compiled Type** column of the Model Explorer or Ports and Data Manager gives the actual type used in the compiled simulation application.

In the following figure, an Embedded MATLAB Function block argument inherits its data type from an input signal of type `double`:

Actual compiled types

The inherited type of output data is inferred from diagram actions that store values in the specified output. In the preceding example, the variables `mean` and `stdev` are computed from operations with double operands, which yield results of type `double`. If the expected type matches the inferred type, inheritance is successful. In all other cases, a mismatch occurs during build time.

Note Library Embedded MATLAB Function blocks can have inherited data types, sizes, and complexities like ordinary Embedded MATLAB Function blocks. However, all instances of the library block in a given model must have inputs with the same properties.

Built-In Data Types for Arguments

When you select **Built-in** for **Data type mode**, the **Data** properties dialog displays a **Data type** field that provides a drop-down list of supported data types. You can also choose a data type from the **Data Type** column in the **Contents** pane of the Model Explorer or Ports and Data Manager. The supported data types are:

Data Type	Description
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 <data type expression> from the **Type** drop-down menu of the Data properties dialog.

2 In the **Type** field, replace "*<data type expression>*" with an expression that evaluates to a data type. The following expressions are allowed:

- Alias type from the MATLAB workspace, as described in "Creating a Data Type Alias" in the Simulink reference documentation.

- fixdt function to create a Simulink.NumericType object describing a fixed-point or floating-point data type

- type operator, to base the type on previously defined data

  In the following figure, the data type of input argument data1 is **int32**. The data type of input argument data2 is based on data1 using the expression type(data1).

  When the model is compiled, the actual type of data2 appears in the Compiled Type column in the **Contents** pane:

Compiled data types

Specifying Simulink® Fixed Point™ Data Properties

Embedded MATLAB Function blocks can represent signals and parameter values as fixed-point numbers. To simulate models that use fixed-point data in Embedded MATLAB Function blocks, you must install the Simulink® Fixed Point™ product on your system (see "Product Overview" in the Simulink Fixed Point documentation).

When you select the `Fixed point` data type Mode, the Data Type Assistant displays fields for additional information about your fixed-point data, as in this example:
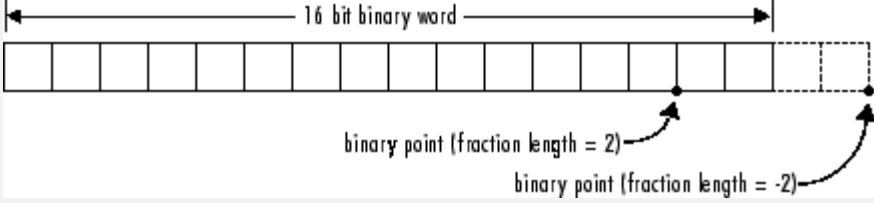
You can set the following fixed-point properties:

Sign. Select whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative quantities. Unsigned data represents positive values only. The default is `Signed`.

Word length. Specify the size (in bits) of the word that will hold the quantized integer. Large word sizes represent large quantities with greater precision than small word sizes. Word length can be any integer between 0 and 32. The default is 16.

Scaling. Specify the method for scaling your fixed point data to avoid overflow conditions and minimize quantization errors. You can select the following scaling modes:

Scaling Mode	Description
Binary point (default)	If you select this mode, the Data Type Assistant displays the **Fraction Length** field, specifying the binary point location. Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example: The default is 0.
Slope and bias	If you select this mode, the Data Type Assistant displays fields for entering the **Slope** and **Bias**. • Slope can be any *positive* real number. The default is 1.0. • Bias can be any real number. The default value is 0.0. You can enter slope and bias as expressions that contain parameters defined in the MATLAB workspace.

Note You should use binary-point scaling whenever possible to simplify the implementation of fixed-point data in generated code. Operations with fixed-point data using binary-point scaling are performed with simple bit shifts and eliminate the expensive code implementations required for separate slope and bias values.

Calculate Best-Precision Scaling. The Simulink software can automatically calculate "best-precision" values for both Binary point and Slope and bias scaling, based on the Limit range properties you specify on the **Value Attributes** tab.

To automatically calculate best precision scaling values:

1 Select the **Value Attributes** tab.

2 Specify **Minimum**, **Maximum**, or both Limit range properties.

3 Select the **General** tab.

4 Click **Calculate Best-Precision Scaling**.

The Simulink software calculates the scaling values, then displays them in either the **Fraction Length**, or **Slope** and **Bias** fields.

Note The Limit range properties do not apply to Constant or Parameter scopes. Therefore, the Simulink software cannot calculate best-precision scaling for these scopes.

Lock output scaling against changes by the autoscaling tool. Specify whether you want to prevent the Simulink software from replacing the current fixed-point type with a type chosen by the autoscaling tool. The default allows replacement. See "Automatic Scaling" in the Simulink Fixed Point documentation for instructions on autoscaling fixed-point data.

Sizing Function Arguments

In this section...
"Specifying Argument Size" on page 22-71
"Inheriting Argument Sizes from Simulink®" on page 22-71
"Specifying Argument Sizes with Expressions" on page 22-73

Specifying Argument Size

To examine or specify the size of an argument, follow these steps:

1 From the Embedded MATLAB™ Editor, select **Model Explorer** or **Tools > Edit Data/Ports**.

2 In the **Contents** pane, click the row that contains the data argument.

3 Enter the size of the argument in one of two places:

- Size field of the Data properties dialog, located in the **Dialog** pane

- Size column in the row that contains the data argument, located in the **Contents** pane

Note The default value is -1, indicating that size is inherited, as described in "Inheriting Argument Sizes from Simulink®" on page 22-71.

Inheriting Argument Sizes from Simulink®

Size defaults to -1, which means that the data argument inherits its size from Simulink® based on its scope:

For Scope	**Inherits Size**
Input	From the Simulink input signal connected to the argument.

For Scope	Inherits Size
Output	From the Simulink output signal connected to the argument.
Parameter	From the Simulink or MATLAB® parameter to which it is bound. See "Parameter Arguments in Embedded MATLAB™ Functions" on page 22-74.

After you compile the model, the **Compiled Size** column in the **Contents** pane displays the actual size used in the compiled simulation application:

Actual compiled sizes
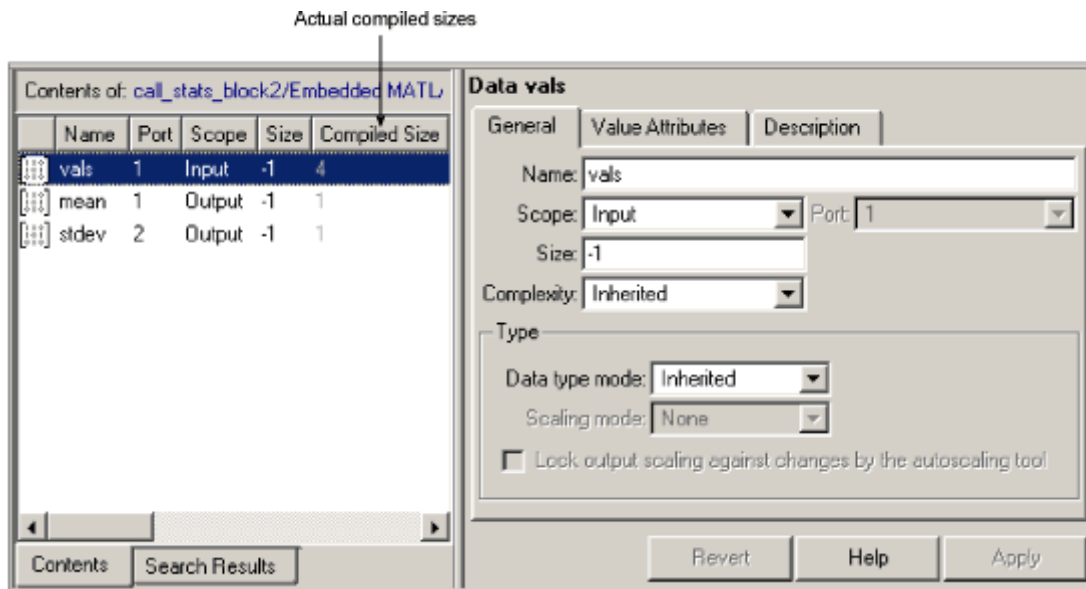
The size of an output argument is the size of the value that is assigned to it. If the expected size in the Simulink model does not match, a mismatch error occurs during compilation of the model.

Note No arguments with inherited sizes are allowed for Embedded MATLAB Function blocks in a library.

Specifying Argument Sizes with Expressions

The size of a data argument can be a scalar value or a MATLAB vector of values.

To specify size as a scalar, set the **Size** field to 1 or leave it blank. To specify **Size** as a vector, enter an array of up to two dimensions in `[row column]` format where

- The number of dimensions equals the length of the vector.
- The size of each dimension corresponds to the value of each element of the vector.

For example, a value of `[2 4]` defines a 2-by-4 matrix. To define a row vector of size 5, set the **Size** field to `[1 5]`. To define a column vector of size 6, set the **Size** field to `[6 1]` or just 6. You can enter a MATLAB expression for each `[row column]` element in the **Size** field. Each expression can use one or more of the following elements:

- Numeric constants
- Arithmetic operators, restricted to +, -, *, and /
- Parameters defined in the MATLAB Workspace or the parent Simulink masked subsystem
- Calls to the MATLAB functions `min`, `max`, and `size`

The following examples are valid expressions for **Size**:

```
k+1
size(x)
min(size(y),k)
```

In these examples, k, x, and y are variables of scope `Parameter`.

Once you build the model, the **Compiled Size** column in the **Contents** pane displays the actual size used in the compiled simulation application.

Parameter Arguments in Embedded MATLAB™ Functions

Parameter arguments for Embedded MATLAB Function blocks do not take their values from signals in the Simulink® model. Instead, their values come from parameters defined in a parent Simulink masked subsystem or variables defined in the MATLAB® base workspace. Using parameters allows you to pass read-only constants in the Simulink model to the Embedded MATLAB Function block.

Use the following procedure to add a parameter argument to a function for an Embedded MATLAB Function block.

1 In the Embedded MATLAB™ Editor, add an argument to the function header of the Embedded MATLAB Function block.

The name of the argument must be identical to the name of the masked subsystem parameter or MATLAB variable that you want to pass to the Embedded MATLAB Function block. For information on declaring parameters for masked subsystems, see "Mask Editor" on page 17-17.

2 Bring focus to the Embedded MATLAB Function block.

The new argument appears as an input port in the Simulink diagram.

3 In the Embedded MATLAB Editor, select **Model Explorer** or **Tools > Edit Data/Ports**.

4 In the **Contents** pane, click the row that contains the new argument.

5 Set **Scope** to Parameter.

6 Examine the Embedded MATLAB Function block.

The input port no longer appears for the parameter argument.

Note Parameter arguments appear as arguments in the function header of the Embedded MATLAB Function block to maintain MATLAB consistency. This lets you test functions in an Embedded MATLAB Function block by copying and pasting them to MATLAB.

Resolving Signal Objects for Output Data

In this section...
"Implicit Signal Resolution" on page 22-75
"Eliminating Warnings for Implicit Signal Resolution in the Model" on page 22-75
"Disabling Implicit Signal Resolution for an Embedded MATLAB™ Function Block" on page 22-76
"Forcing Explicit Signal Resolution for an Output Data Signal" on page 22-76

Implicit Signal Resolution

Embedded MATLAB™ Function blocks participate in signal resolution with Simulink® signal objects. By default, output data from Embedded MATLAB Function blocks become associated with Simulink signal objects of the same name during a process called *implicit signal resolution*, as described in Simulink.Signal in the Reference documentation.

By default, implicit signal resolution generates a warning when you update the chart in the Simulink model. The following sections show you how to manage implicit signal resolution at various levels of the model hierarchy. See "Resolving Symbols" on page 3-69 and "Explicit and Implicit Resolution" on page 3-72 for more information.

Eliminating Warnings for Implicit Signal Resolution in the Model

To enable implicit signal resolution for all signals in a model, but eliminate the attendant warnings, follow these steps:

1 In the Simulink Model Editor, select **Simulation > Configuration Parameters**.

The Configuration Parameters dialog appears.

2 In the left pane of the Configuration Parameters dialog, under Diagnostics, select **Data Validity**.

Data Validity configuration parameters appear in the right pane.

3 In the Signal resolution field, select **Explicit and implicit**.

Disabling Implicit Signal Resolution for an Embedded MATLAB™ Function Block

To disable implicit signal resolution for an Embedded MATLAB Function block in your model, follow these steps:

1 Right-click the Embedded MATLAB Function block and select **Subsystem Parameters** in the context menu.

The Block Parameters dialog opens.

2 In the Permit hierarchical resolution field, select **ExplicitOnly** or **None**, and click **OK**.

Forcing Explicit Signal Resolution for an Output Data Signal

To force signal resolution for an output signal in an Embedded MATLAB Function block, follow these steps:

1 In the Simulink model, right-click the signal line connected to the output that you want to resolve and select **Signal Properties** from the context menu.

2 In the Signal Properties dialog, enter a name for the signal that corresponds to the signal object.

3 Select the **Signal name must resolve to Simulink signal object** check box and click **OK**.

Working with Structures and Bus Signals

About Structures in Embedded MATLAB™ Function Blocks

Embedded MATLAB™ Function blocks support MATLAB® structures (see "Structures" in the MATLAB Programming Fundamentals documentation). In Embedded MATLAB Function blocks, you can define structure data as inputs or outputs that interact with bus signals. You can also define structures inside Embedded MATLAB functions that are not part of Embedded MATLAB Function blocks (see "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`.	You can create structure data as inputs or outputs in the top-level Embedded MATLAB function for interfacing to other environments. See "Workflow for Creating Structures in Embedded MATLAB™ Function Blocks" on page 22-83.
Output	Create structure data with scope of `Output`.	
Local	Create a local variable implicitly in an Embedded MATLAB function.	See "Defining Local Structure Variables" in the Embedded MATLAB documentation.
Persistent	Declare a variable to be persistent in an Embedded MATLAB function.	See "Making Structures Persistent" in the Embedded MATLAB documentation.
Parameter	Create structure data with scope of `Parameter`	See "Working with Non-Tunable Structure Parameters in Embedded MATLAB™ Function Blocks" on page 22-87.

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 Structures in the Embedded MATLAB Subset" in the Embedded MATLAB documentation.

Example of Structures in an Embedded MATLAB™ Function Block

The following example shows how to use structures in an Embedded MATLAB Function block:

```
function [outbus, outbus1] = fcn(inbus)

substruct.a1 = inbus.ele3.a1;
substruct.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5, 'ele2', single(100), 'ele3', substruct);

outbus = mystruct;
outbus.ele3.a2 = 2*(substruct.a2);

outbus1 = inbus.ele3;
```
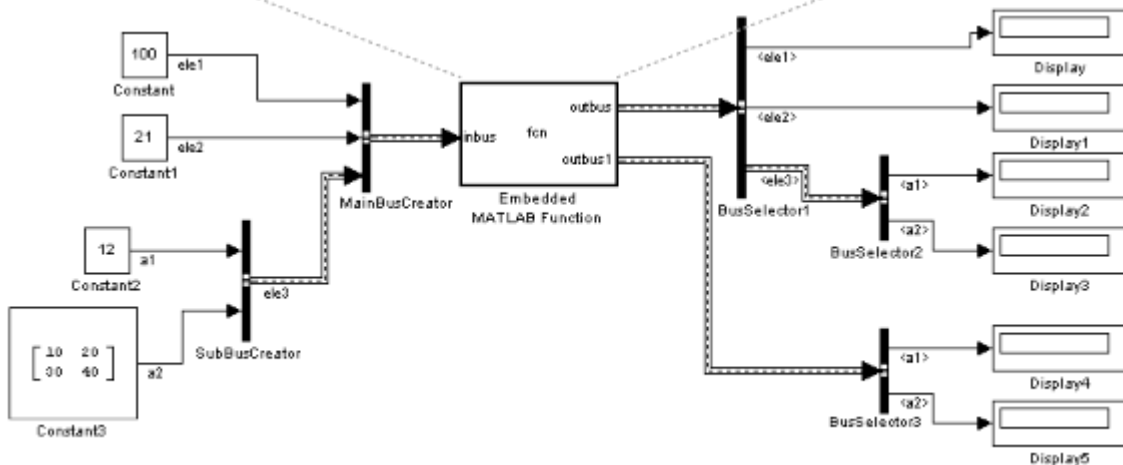

In this model, an Embedded MATLAB Function block receives a bus signal using the structure inbus at input port 1 and outputs two bus signals from the structures outbus at output port 1 and outbus1 at output port 2. The input signal comes from the Bus Creator block MainBusCreator, which bundles signals ele1, ele2, and ele3. The signal ele3 is the output of another Bus Creator block SubBusCreator, which bundles the signals a1 and a2. The structure outbus connects to a Bus Selector block BusSelector1; the structure outbus1 connects to another Bus Selector block BusSelector3.

Like other outputs in the Embedded MATLAB subset, structure outputs must be initialized. The Embedded MATLAB function in this example implicitly defines a local structure variable mystruct using the struct function, and

uses this local structure variable to initialize the value of the first output `outbus`. It initializes the second output `outbus1` to the value of field `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

Bus Objects Define Structure Inputs and Outputs

Each structure input and output must be defined by a `Simulink.Bus` object in the base workspace (see "Workflow for Creating Structures in Embedded MATLAB™ Function Blocks" on page 22-83). This means that the structure shares the same properties as the bus object, including number, name, and type of fields. In this example, the following bus objects define the structure inputs and outputs:
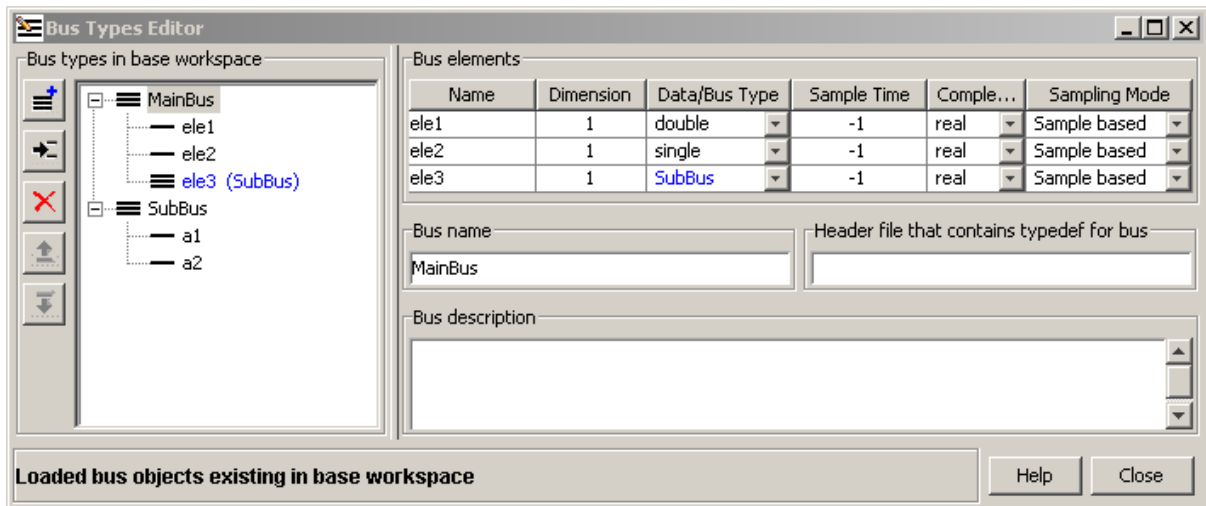
The `Simulink.Bus` object `MainBus` defines structure input `inbus` and structure output `outbus`. The `Simulink.Bus` object `SubBus` defines structure output `outbus1`. Based on these definitions, `inbus` and `outbus` have the same properties as `MainBus` and, therefore, reference their fields by the same names as the fields in `MainBus`, using dot notation (see "Indexing Substructures and Fields" on page 22-84). 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 Blocks" on page 22-83.

How Structure Inputs and Outputs Interface with Bus Signals

Buses in a Simulink® model appear inside the Embedded MATLAB Function block as structures; structure outputs from the Embedded MATLAB Function block appear as buses in Simulink models. When you create structure inputs, the Embedded MATLAB Function block determines the type, size, and complexity of the structure from the input signal. When you create structure outputs, you must define their type, size, and complexity in the Embedded MATLAB function.

You connect structure inputs and outputs from Embedded MATLAB Function blocks to any bus signal, including:

- Blocks that output bus signals — such as Bus Creator blocks

- Blocks that accept bus signals as input — such as Bus Selector and Gain blocks

- S-Function blocks

- Other Embedded MATLAB Function blocks

Working with Virtual and Nonvirtual Buses

Embedded MATLAB Function blocks supports nonvirtual buses only (see "Virtual and Nonvirtual Buses" on page 9-30 in the Simulink User's Guide). When models that contain Embedded MATLAB function inputs and outputs are built, hidden converter blocks are used to convert bus signals for use with Embedded MATLAB, as follows:

- Converts incoming virtual bus signals to nonvirtual buses for Embedded MATLAB structure inputs

- Converts outgoing nonvirtual bus signals from Embedded MATLAB to virtual bus signals

Rules for Defining Structures in Embedded MATLAB™ Function Blocks

Follow these rules when defining structures in Embedded MATLAB Function blocks:

- For each structure input or output in an Embedded MATLAB Function block, you must define a `Simulink.Bus` object in the base workspace to specify its type. For more information, see Simulink.Bus.

- Embedded MATLAB Function blocks support nonvirtual buses only (see "Working with Virtual and Nonvirtual Buses" on page 22-82).

Workflow for Creating Structures in Embedded MATLAB™ Function Blocks

Here is the workflow for creating a structure in Embedded MATLAB:

1 Decide on the type (or scope) of the structure (see "About Structures in Embedded MATLAB™ Function Blocks" on page 22-77).

2 Based on the scope, follow these guidelines for creating the structure:

For Structure Scope:	Follow These Steps:
Input	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 22-44. The data should have the following properties • **Scope** = `Input` • **Mode** = `Bus Object` • **Bus object** = name of the `Simulink.Bus` object that defines the structure input See "Rules for Defining Structures in Embedded MATLAB™ Function Blocks" on page 22-82.

For Structure Scope:	**Follow These Steps:**
Output	**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` • **Mode** = `Bus Object` • **Bus object** = 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.
Local	Define the structure implicitly as a local variable in the Embedded MATLAB function, as described in "Defining Local Structure Variables" in the Embedded MATLAB documentation. By default, local variables in Embedded MATLAB are temporary.
Persistent	Define the structure implicitly as a persistent variable in the Embedded MATLAB function, as described in "Making Structures Persistent" in the Embedded MATLAB documentation.
Parameter	**1** Create a structure variable in the base workspace. **2** Add data to the Embedded MATLAB Function block with the following properties: • **Name** = same name as the structure variable you created in step 1. • **Scope** = `Parameter` **3** Make sure the **Tunable** property is unchecked. See "Working with Non-Tunable Structure Parameters in Embedded MATLAB™ Function Blocks" on page 22-87.

Indexing Substructures and Fields

As in MATLAB, you index substructures and fields of Embedded MATLAB structures by using dot notation. Unlike MATLAB, you must reference field

values individually (see "Limitations with Structures" in the Embedded MATLAB documentation).

For example, in the model described in "Example of Structures in an Embedded MATLAB™ Function Block" on page 22-78, the Embedded MATLAB function uses dot notation to index fields and substructures:

```
function [outbus, outbus1] = fcn(inbus)

substruct.a1 = inbus.ele3.a1;
substruct.a2 = int8([1 2;3 4]);

mystruct = struct('ele1',20.5,'ele2',single(100),
                  'ele3',substruct);

outbus = mystruct;
outbus.ele3.a2 = 2*(substruct.a2);

outbus1 = inbus.ele3;
```

The following table shows how Embedded MATLAB resolves symbols in dot notation for indexing elements of the structures in this example:

Dot Notation	Symbol Resolution
substruct.a1	Field a1 of local structure substruct
inbus.ele3.a1	Value of field a1 of field ele3, a substructure of structure inputinbus
inbus.ele3.a2(1,1)	Value in row 1, column 1 of field a2 of field ele3, a substructure of structure input inbus

Assigning Values to Structures and Fields

You can assign values to any Embedded MATLAB structure, substructure, or field. Here are the guidelines:

Operation	Conditions
Assign one structure to another structure	You must define each structure with the same number, type, and size of fields, either as `Simulink.Bus` objects in the base workspace or locally as implicit structure declarations (see "Workflow for Creating Structures in Embedded MATLAB™ Function Blocks" on page 22-83).
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 22-78:

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 same `Simulink.Bus` object, `MainBus`.

Assignment	Valid or Invalid?	Rationale
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.

Working with Non-Tunable Structure Parameters in Embedded MATLAB™ Function Blocks

You can define non-tunable structure parameters in Embedded MATLAB Function blocks. Models that contain non-tunable structure parameters will compile both for simulation and code generation with Real-Time Workshop®.

Defining Non-Tunable Structure Parameters

To define non-tunable structure parameters in Embedded MATLAB Function blocks, follow these steps:

1 Define and initialize a structure variable

A common method is to create a structure in the base workspace. For other methods, see "Working with Block Parameters" on page 6-8.

2 In the Ports and Data Manager or Model Explorer, add data in the Embedded MATLAB Function block with the following properties:

Property	What to Specify
Name	Enter same name as the structure variable you defined in the base workspace
Scope	Select Parameter
Tunable	Uncheck this check box
Type	Select Inherit: Same as Simulink

For example, the data dialog box fields should look something like this:

Caution Embedded MATLAB Function blocks do not support tunable structure parameters. If you define a structure parameter as tunable, you receive a run-time error:

```
Parameter is a tunable structure parameter
which is not supported. Set the parameter
to be non-tunable, or do not use a structure value.
```

---

3 Click **Apply**.

FIMATH Properties of Non-Tunable Structure Parameters

FIMATH properties for non-tunable structure parameters containing fixed-point values are based on the initial values of the structure. They do *not* come from the FIMATH properties specified for fixed-point input signals to the parent Embedded MATLAB Function block. (These FIMATH properties appear in the properties dialog box for Embedded MATLAB Function blocks.)

Rules for Defining Non-Tunable Structure Parameters

Non-tunable structure parameters in Embedded MATLAB Function blocks have the same limitations as structures in Embedded MATLAB functions. See "Limitations with Structures" in the Embedded MATLAB documentation.

Example: Using a Non-Tunable Structure Parameter to Initialize a Matrix

The following simple example uses a non-tunable structure parameter input to initialize a matrix output. The model looks like this:
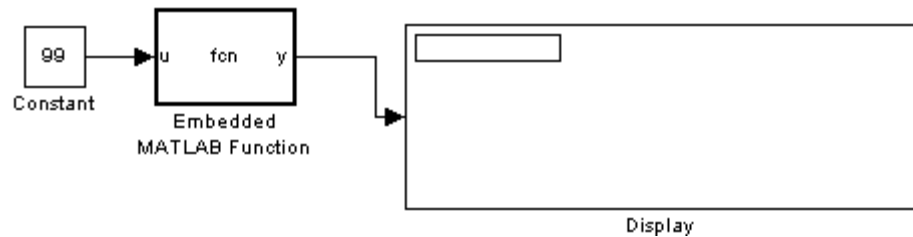
This model defines a structure variable p in its pre-load callback function, as follows:
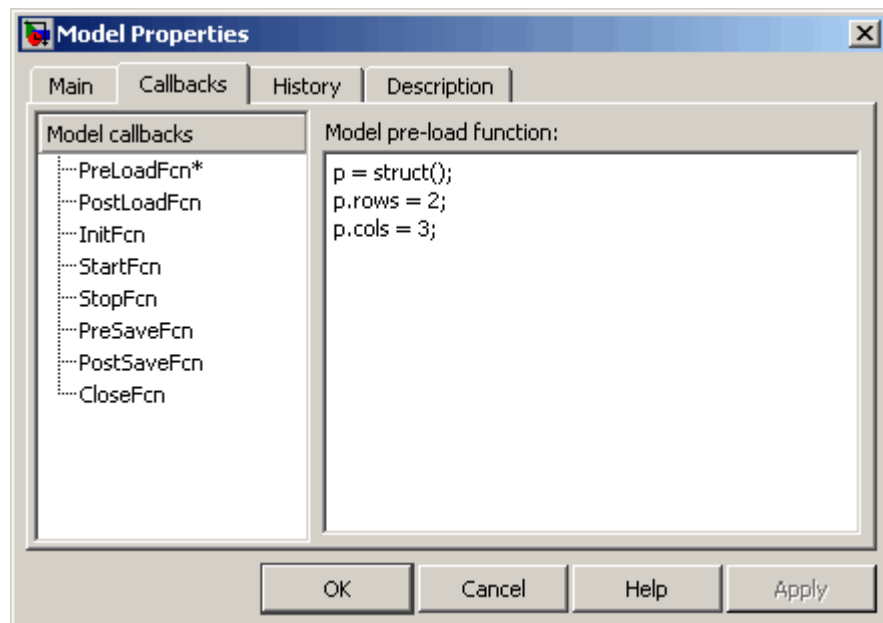
The structure p has two fields, rows and cols, which specify the dimensions of a matrix. The Embedded MATLAB Function block uses a constant input u to initialize the matrix output y. Here is the code:

```
function y = fcn(u, p)
y = zeros(p.rows,p.cols) + u;
```

Running the model initializes each element of the 2-by-3 matrix y to 99, the value of u:
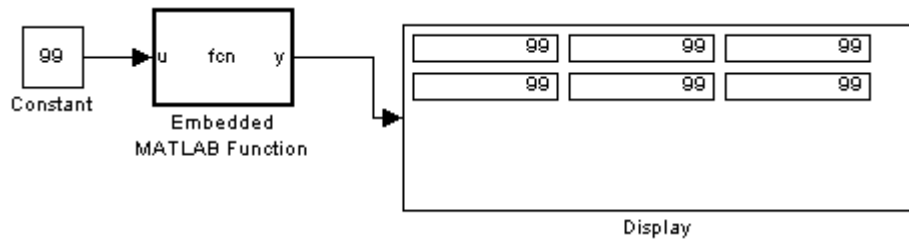
Limitations of Structures in Embedded MATLAB™ Function Blocks

Structures in Embedded MATLAB Function blocks support a subset of the operations available for MATLAB structures (see "Limitations with Structures" in the Embedded MATLAB documentation).

Working with Frame-Based Signals

In this section...
"About Frame Based Signals" on page 22-91
"Supported Types for Frame-Based Data" on page 22-92
"Adding Frame-Based Data in Embedded MATLAB™ Function Blocks" on page 22-92
"Examples of Frame-Based Signals in Embedded MATLAB™ Function Blocks" on page 22-93

About Frame Based Signals

Embedded MATLAB Function blocks can input and output frame-based signals in Simulink® models. A frame of data is a collection of sequential samples from a single channel or multiple channels. To generate frame-based signals, you must install Signal Processing Blockset. For more information about using frame-based signals, see "Frame-Based Signals" in the Signal Processing Blockset documentation.

Embedded MATLAB Function blocks automatically convert incoming frame-based signals as follows:

- Converts single-channel frame-based signals to MATLAB® column vectors

- Converts multichannel frame-based signals to two-dimensional MATLAB matrices

An M-by-N frame-based signal represents M consecutive samples from each of N independent channels. N-Dimensional signals are not supported for frames.

To convert matrix or vector data to a frame-based output, Embedded MATLAB™ provides a data property called **Sampling mode** that lets you specify whether your output is a frame-based or sample-based signal for downstream processing.

Supported Types for Frame-Based Data

Embedded MATLAB Function blocks accept frame-based signals of any data type *except* bus objects. For a list of supported types, see "Supported Variable Types" in the Embedded MATLAB documentation.

Adding Frame-Based Data in Embedded MATLAB™ Function Blocks

To add frame-based data to an Embedded MATLAB Function block, follow these steps:

1 Add an input or output, as described in "Adding Data to an Embedded MATLAB™ Function Block" on page 22-44.

2 If your data is an output, set **Sampling mode** to Frame based.
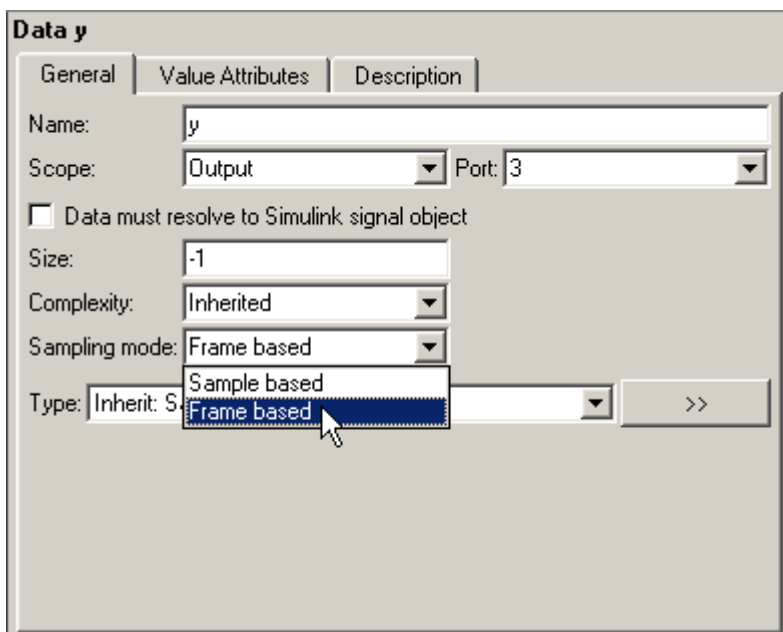
Note If your data is an input, **Sampling mode** is not an option.

> **Note** For more information on how to set data properties, see "Defining Data in the Model Explorer" on page 22-46.

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 22-93
- "Adding a Channel to a Frame-Based Signal" on page 22-95

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:
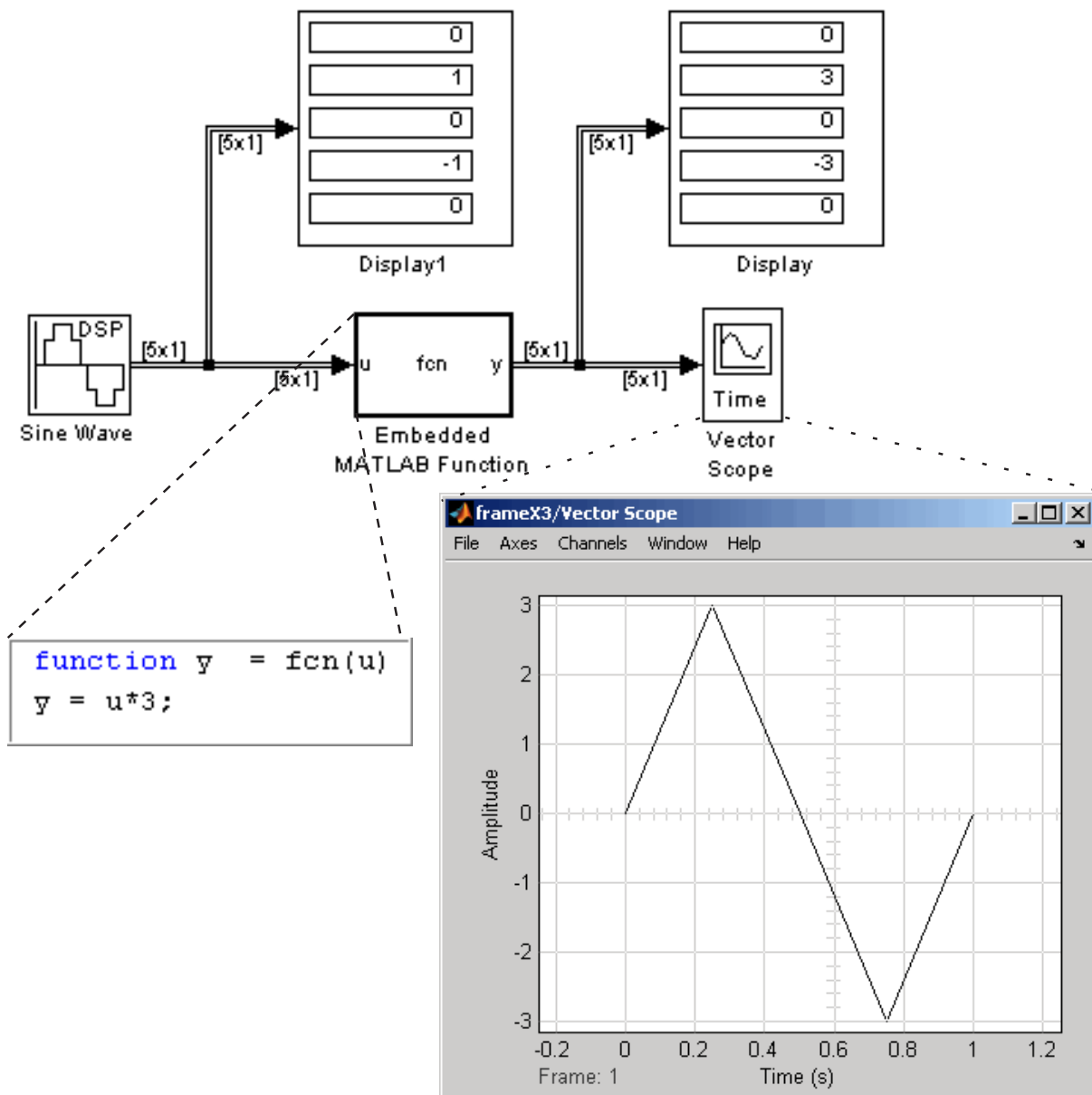
```
function y = fcn(u)
y = u*3;
```

In the Embedded MATLAB Function block, input u and output y inherit size, complexity, and data type from the input sine wave signal, a 5-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set **Sampling mode** to Frame based (see "Adding Frame-Based Data in Embedded MATLAB™ Function Blocks" on page 22-92). 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:
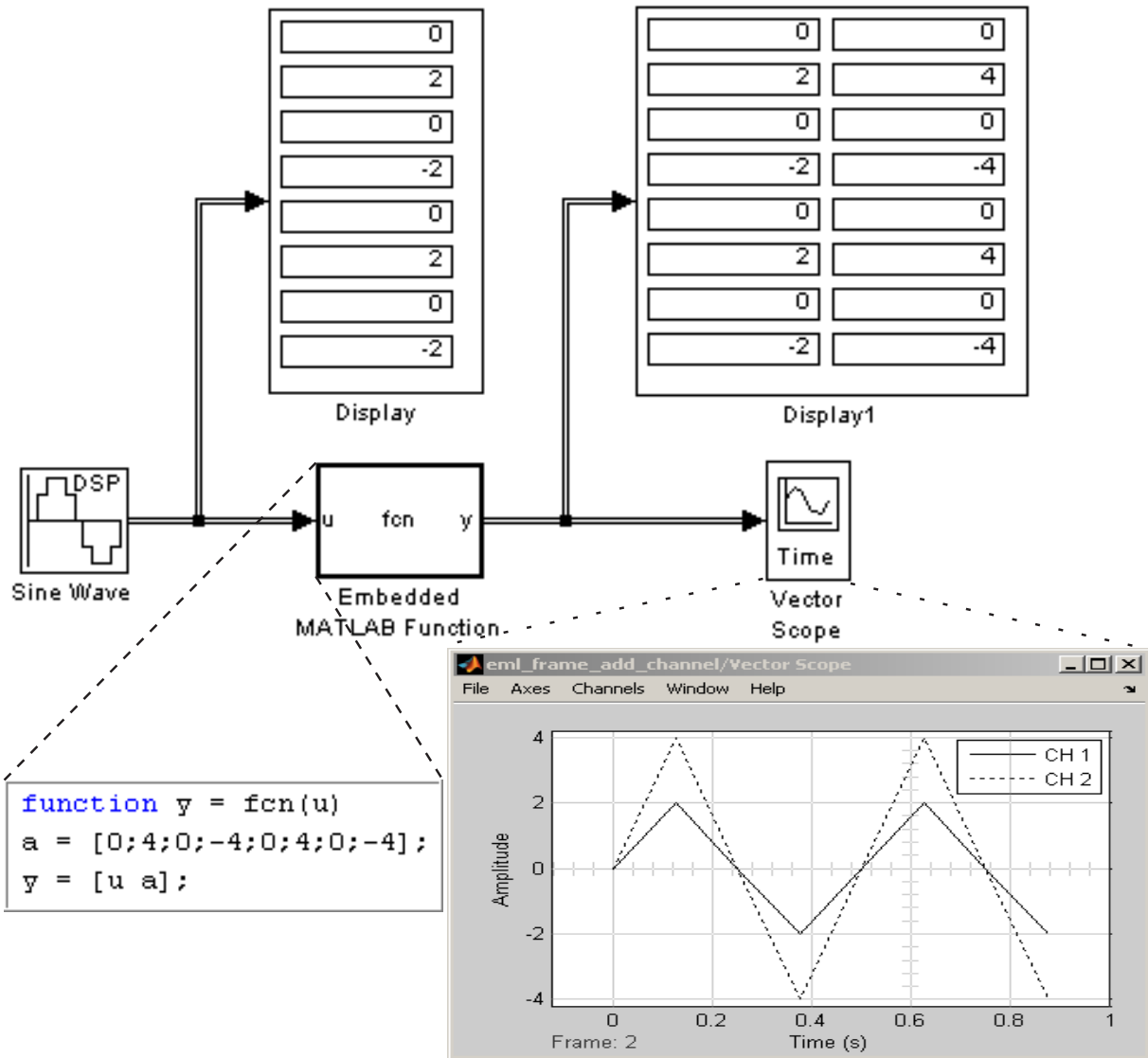
```
function y = fcn(u)
a = [0;4;0;-4;0;4;0;-4];
y = [u a];
```

In the Embedded MATLAB Function block, input u and output y inherit size, complexity, and data type from the input sine wave signal, an 8-by-1 vector of signed, generalized fixed-point values. For y to output a frame of data, you must explicitly set **Sampling mode** to Frame based(see "Adding

Frame-Based Data in Embedded MATLAB™ Function Blocks" on page 22-92).
Local variable a defines a second column on the matrix which will be output
as a frame and interpreted as a second channel by downstream blocks. When
you simulate this model, the Embedded MATLAB Function block outputs
the new multichannel signal.

Using Traceability in Embedded MATLAB™ Function Blocks

In this section...
"Extent of Traceability in Embedded MATLAB™ Function Blocks" on page 22-98
"Traceability Requirements" on page 22-98
"Basic Workflow for Using Traceability" on page 22-99
"Tutorial: Using Traceability in an Embedded MATLAB™ Function Block" on page 22-100

Extent of Traceability in Embedded MATLAB™ Function Blocks

Like other Simulink® blocks, Embedded MATLAB™ Function blocks support bidirectional traceability, but extend navigation to lines of source code. That is, you can navigate between a line of generated code and its corresponding line of source code. In other Simulink blocks, you can navigate between a line of generated code and its corresponding object.

For information about how traceability works in Simulink blocks, see "Tracing Generated Code Back to Your Simulink Model" in the Real-Time Workshop® User's Guide.

Traceability Requirements

To enable traceability comments in your code, you must have a license for Real-Time Workshop® Embedded Coder™ software. These comments appear only in code that you generate for an embedded real-time (`ert`) based target.

Note Traceability is not supported for M-files that you call from an Embedded MATLAB Function block.

Basic Workflow for Using Traceability

The workflow for using traceability is described in "Creating and Using a Code Generation Report" in the Real-Time Workshop Embedded Coder User's Guide.

Here are the basic steps:

1 Open the Embedded MATLAB Function block in your Simulink model.

2 Define your system target file to be an embedded real-time (`ert`) target.

> **How?**
>
> **a** In the model, open **Simulation > Configuration Parameters**.
>
> **b** In the Real-Time Workshop pane, enter `ert.tlc` for the system target file.

3 Enable traceability options.

> **How?**
>
> In the Real-Time Workshop Report pane, enable options, as shown:
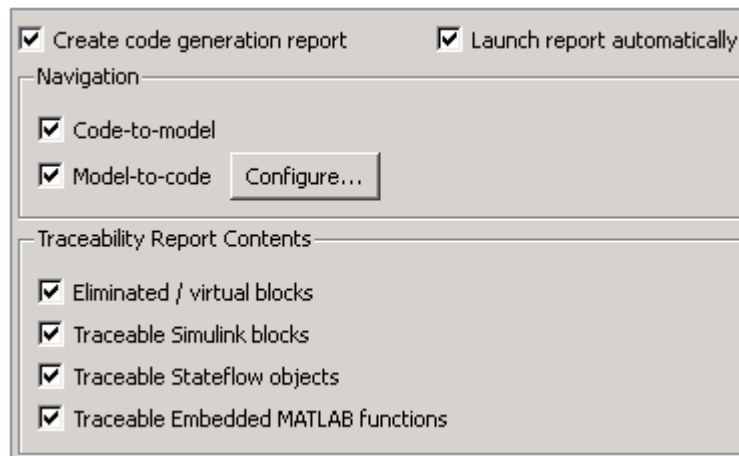
4 Generate the source code and header files for your model.

5 Trace a line of code:

To Trace:	Do This:
Line of source code to line of generated code	Right-click in a line in your source code and select **Real-Time Workshop > Navigate to Code** from the context menu
Line of generated code to line of source code	Click a hyperlink in the traceability comment in your generated code

To learn how to complete each step in this workflow, see "Tutorial: Using Traceability in an Embedded MATLAB™ Function Block" on page 22-100

Tutorial: Using Traceability in an Embedded MATLAB™ Function Block

This example shows how to trace between source code and generated code in an Embedded MATLAB Function block in the eml_fire demo model. Follow these steps:

1 Type eml_fire at the MATLAB® prompt.

2 In the Simulink model window, double-click the flame block to open the Embedded MATLAB Editor.

3 In the Simulink model window, select **Simulation > Configuration Parameters**.

4 In the **Real-Time Workshop** pane, go to the **Target selection** section and enter ert.tlc for the system target file. Then click **Apply**.

Note Traceability comments appear hyperlinked in generated code only for embedded real-time (ert) targets.

---

5 In the **Real-Time Workshop > Report** pane, select the **Create code generation report** option.

This action automatically selects the **Launch report automatically** and **Code-to-model** options.

6 Select the **Model-to-code** option in the **Navigation** section. Then click **Apply**.

This action automatically selects all options in the **Traceability Report Contents** section.

Note For large models that contain over 1000 blocks, disable the **Model-to-code** option to speed up code generation.

7 Go to the **Real-Time Workshop > Interface** pane. In the **Software environment** section, select the **continuous time** option. Then click **Apply**.

Note Because this demo model contains a block with a continuous sample time, you must perform this step before generating code.

8 In the **Real-Time Workshop** pane, click **Build** in the lower right corner.

This action generates source code and header files for the eml_fire model that contains the flame block. After the code generation process is complete, the code generation report appears automatically.
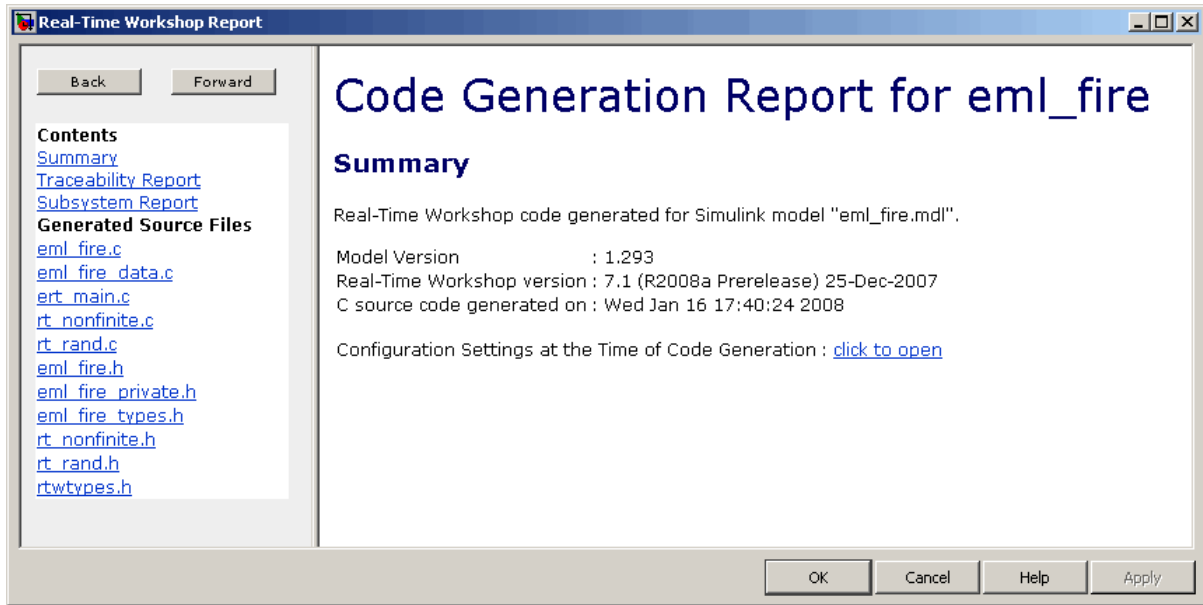
9 Click the `eml_fire.c` hyperlink in the report.

10 Scroll down through the code to see the traceability comments.

```
61      for (eml_y = 0; eml_y < 60; eml_y += 2) {
62        /* '<S2>:1:18' */
63        for (eml_x = 0; eml_x < 256; eml_x++) {
64          /* '<S2>:1:19' */         Traceability comment for a
65          /* '<S2>:1:21' */         line of code in an Embedded
                                       MATLAB function
66          eml_yb = eml_y + 3;
67
68          /* '<S2>:1:22' */         Traceability comment for a
                                       line of code in an Embedded
69          eml_xb1 = eml_x;          MATLAB function
```

Note The line numbers shown above may differ from the numbers that appear in your code generation report.

11 Click the <S2>:1:19 hyperlink in this traceability comment:

```
/* '<S2>:1:19' */
```

Line 19 of the Embedded MATLAB function appears highlighted in the Embedded MATLAB Editor.

Highlighted line of function

12 You can also trace a line in an Embedded MATLAB function to a line of generated code. For example, right-click in line 21 of your function and select **Real-Time Workshop > Navigate to Code** from the context menu.

The code location for line 21 appears highlighted in `eml_fire.c`.

```
61      for (eml_y = 0; eml_y < 60; eml_y += 2) {
62          /* '<S2>:1:18' */
63          for (eml_x = 0; eml_x < 256; eml_x++) {
64              /* '<S2>:1:19' */
65              /* '<S2>:1:21' */
66              eml_yb = eml_y + 3;
```

Highlighted
line of code

23

PrintFrame Editor

PrintFrame Editor Overview

In this section...
"About the Print Frame Editor" on page 23-2
"What PrintFrames Are" on page 23-3
"Starting the PrintFrame Editor" on page 23-6
"Getting Help for the PrintFrame Editor" on page 23-7
"Closing the PrintFrame Editor" on page 23-7
"Print Frame Process" on page 23-7

About the Print Frame Editor

The PrintFrame Editor is a graphical user interface you use to create and edit print frames for block diagrams created with the Simulink® software and the Stateflow® product. This chapter outlines the PrintFrame Editor, accessible with the frameedit command.

The following figure describes the general layout of the PrintFrame Editor.

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.

What PrintFrames Are

Print frames are borders containing information relevant to the block
diagram, for example, the name of the block diagram. After creating a print

frame, you can use the Simulink software or the Stateflow product to print a block diagram with a print frame.

This illustration shows an example of a print frame with the major elements labeled.

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)

1.0

Atmospheric
Pressure, Pa
(bar)

Atmospheric Pressure, Pa (bar)

Throttle Flow, mdot (g/s)

Throttle

mdot Input (g/s)

mdot to Cylinder (g/s)

1

mass(k+1)

2

Engine Speed, N

N (rad/sec)

Manifold Pressure, Pm (bar)

Intake Manifold

$\frac{1}{s}$

3

trigger

Intake

sldemo_engine/Throttle & Manifold

1 of 3

Variable information included
in print frame

Simulink block diagram

See the "Example" on page 23-26 for specific instructions to create this print
frame.

Starting the PrintFrame Editor

Type `frameedit` at the MATLAB® prompt. The **PrintFrame Editor** window appears. The **PrintFrame Editor** window opens with the default print frame.

You can use `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.
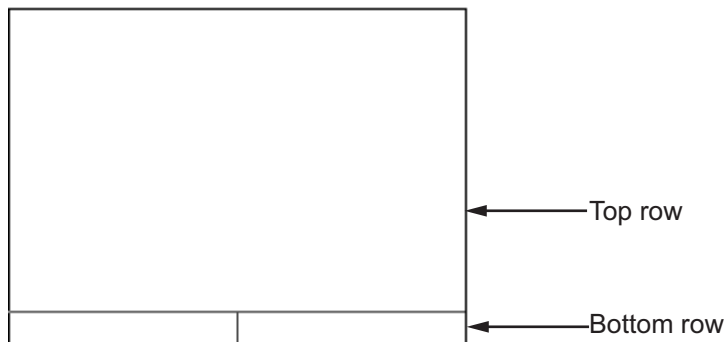
You can add information entries to these cells. You can also add new rows and cells and add information in them, or change entries to different ones.

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.
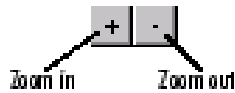
The area is magnified.

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 23-8
- "Specifying the Print Frame Page Setup" on page 23-9
- "Creating Borders (Rows and Cells)" on page 23-11
- "Adding Information to Cells" on page 23-14
- "Changing Information in Cells" on page 23-18
- "Saving and Opening Print Frames" on page 23-22
- "Printing Block Diagrams with Print Frames" on page 23-23

See also the "Example" on page 23-26.

Designing the Print Frame

In this section...
"Before you Begin" on page 23-8
"Variable and Static Information" on page 23-8
"Single Use or Multiple Use Print Frames" on page 23-8

Before you Begin

Before you create a print frame using the PrintFrame Editor, consider the type of information you want to include in it and how you want the information to appear. You might want to make a sketch of how you want the print frame to look, and note the wording you want to use.

Variable and Static Information

In a print frame, you can include variable and static information. Variable information is automatically supplied at the time of printing, for example, the date the block diagram is being printed. Static information always prints exactly as you entered it, for example, the name and address of your organization.

Single Use or Multiple Use Print Frames

You can design a print frame for one particular block diagram, or you can design a more generic print frame for printing with different block diagrams.

Specifying the Print Frame Page Setup

After you have an idea of the design of your print frame, specify the page setup for the print frame.

Note Always begin creating a new print frame with **PrintFrame Page Setup**. If, instead, you begin by creating borders and adding information, and then later change the page setup, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the page setup paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

---

1 In the **PrintFrame Editor** window, select **Page Setup** from the **File** menu.

The **PrintFrame Page Setup** dialog box appears.
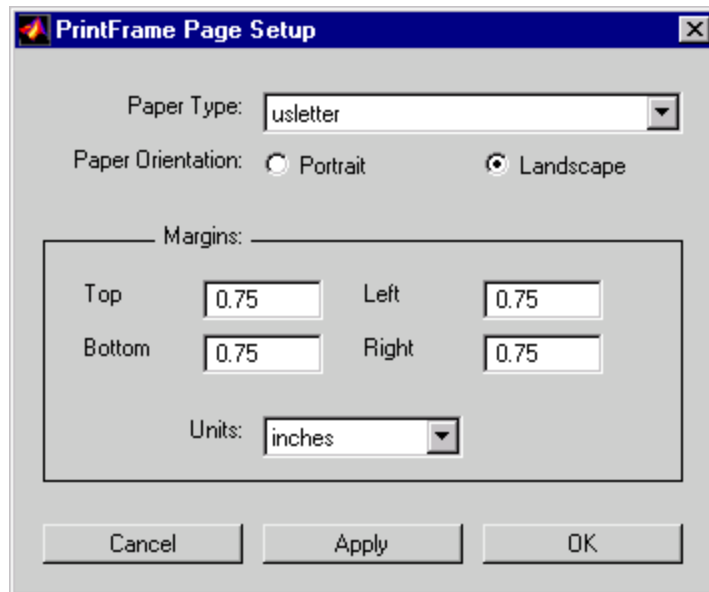
2 In the dialog box, specify:

- **Paper Type** – for example, usletter

- **Paper Orientation** – portrait or landscape

- **Margins** for the print frame and the **Units** in which to specify the margins

3 Click **Apply** to see the effects of the changes you made. Then click **OK** to close the dialog box.

Creating Borders (Rows and Cells)

First Steps

Once you have set up the page, use the PrintFrame Editor to specify borders (cells) in which the block diagram and information will be placed.

Important Always specify the PrintFrame page setup before creating borders and adding information (see "Specifying the Print Frame Page Setup" on page 23-9). Otherwise, you might have to correct the borders and placement of the information. For example, if you add information to cells and then change the paper orientation from landscape to portrait, the information you added might not fit in the cells, given the new orientation.

Adding and Removing Rows

You can add and remove rows in a print frame.

1 Click within an existing row to select it. If a row consists of multiple cells, click in any of the cells in the row to select that row.

When a row is selected, handles appear on all four corners. If handles appear on only two corners, you clicked on and only selected the line, not the row.

Cell (or row) selected - note 4 handles.

Line selected - note 2 handles.
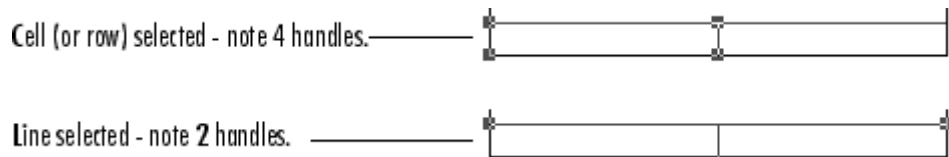
2 Click the **add row** button to create a new row.

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 23-9.

Adding Information to Cells

In this section...
"Procedure for Adding Information to Cells" on page 23-14
"Text Information" on page 23-15
"Variable Information" on page 23-15
"Multiple Entries in a Cell" on page 23-16

Procedure for Adding Information to Cells

Use the following steps to add information to cells.

1 Select the cell where you want to add information.

2 From the list box, select the type of information you want to add.

3 Click the **Add** button.

An edit box containing that information appears in the cell. (The edit boxes for your platform might look slightly different from those in the figure below.)

4 Click outside of the edit box to end editing mode.

> **Note** If you click the **Add** button and nothing happens, it might be because you did not select a cell first.

Text Information

For **Text**, type the text you want to include in that cell, for example, the name of your organization. Press the **Enter** key if you want to type additional text on a new line. Note that you can type special characters, for example, superscripts and subscripts, Greek letters, and mathematical symbols. For special characters, use embedded TeX sequences (see the `text` command `String` property (in Text Properties of the online documentation) for a list of allowable sequences). Click outside of the edit box when you are finished to end editing mode.

Variable Information

All of the items in the information list box, except for the **Text** item, are for adding variable information, which is supplied at the time of printing. When you print a block diagram with a print frame that contains variable information, the information for that particular block diagram prints in those fields.

Types of Variable Information

The variable entries you can include are:

- **Block Diagram** — This entry indicates where the block diagram is to be printed. **Block Diagram** is a mandatory entry. If **Block Diagram** is not in one of the cells, you cannot save the print frame and therefore cannot print a block diagram with it.

- **Date** — The date that the block diagram and print frame are printed, in `dd-mmm-yyyy` format, for example, `05-Dec.-1997`.

- **Time** — The time that the block diagram and print frame are printed, in `hh:mm` format, for example, `14:22`.

- **Page Number** — The page of the block diagram being printed.

- **Total Pages** — The total number of pages being printed for the block diagram, which depends on the printing options specified.

- **System Name** — The name of the block diagram being printed.

- **Full System Name** — The name of the block diagram being printed, including its position from the root system through the current system, for example, `engine/Throttle & Manifold`.

- **File Name** — The filename of the block diagram, for example, `sldemo_engine.mdl`.

- **Full File Name** — The full path and filename for the block diagram, for example, `\\matlab\toolbox\simulink\simdemos\automotive\sldemo_engine.mdl`.

Note: Adding the system name or filename does not mean that you can then specify a filename for the Simulink® software or the StateFlow product in the PrintFrame Editor. It means that when you print a block diagram and specify that it print with a print frame, the system name or filename of the Simulink software or the Stateflow® product block diagram is printed in the specified cell of the print frame.

Format for Variable Information

When you add a variable entry, a percent sign, `%`, is automatically included to identify the entry as variable information rather than a text string. In addition, the type of entry, for example, `page`, appears in angle brackets, `< >`. The entry consists of the entire string, for example, `%<page>`, for **Page Number**.

Multiple Entries in a Cell

You can include multiple entries in one cell.

1 Select the cell.

2 Add another item from the list box.

   The new entry is added after the last entry in that cell.

You can also type descriptive text to any of the variable entries without using the **Text** item in the information list box.

1 Double-click in the cell.

An edit box appears around the entry.

2 Type text in the edit box before or after the entry.

3 Click anywhere outside of the edit box to end editing mode.

Note You cannot include multiple entries or text in the cell that contains the block diagram entry. `%<blockdiagram>` must be the only information in that cell. If there is any other information in that cell, you cannot save the print frame and therefore cannot print it with a block diagram.

Changing Information in Cells

Aligning the Information in a Cell

To align the information within a cell:

1 Click within the cell to select it.

2 Click on one of the **Align** buttons for left, center, or right alignment.

The information aligns within the cell.

Alignment does not apply to the cell that contains the `%<blockdiagram>` entry. The block diagram is automatically scaled and centered to fit in that cell at the time of printing.

Editing Text Strings

You can change text you typed in a cell:

1 Double-click the information you want to edit.

An edit box appears around all of the information in that cell.

2 Click at the start of the text you want to change and drag to the end of the text to be changed.

This highlights the text.

3 Type the replacement text.

It automatically replaces the highlighted text.

4 Click anywhere outside of the edit box to end editing mode.

Note Be careful not to edit the text of a variable entry, because then the variable information will not print. For example, if you accidentally remove the `%` from the `%<page>` entry, the text `<page>` will print instead of the actual page number.

Removing and Copying Entries

You can cut, copy, paste, or delete an entry:

1 Double-click the information you want to remove or copy.

An edit box appears around all of the information in that cell.

2 Click at the start of the entry you want to edit and drag to the end of that entry. This highlights the entry.

For variable information, be sure to include the entire string, for example, `%<page>`.

Note that for computers running the Microsoft® Windows® operating system, you can select all of the entries in a cell by right-clicking the information and choosing **Select All** from the pop-up menu.

3 Use the standard editing techniques for your platform to cut, copy, or delete the highlighted information.

- For computers running the Microsoft Windows operating system, right-click in the edit box and select **Cut**, **Copy**, or **Delete** from the pop-up menu.

- For UNIX® based systems, highlighting the information automatically copies it to the clipboard. If you want to remove it, press the **Delete** key.

If you make a mistake, use your platform's standard undo technique. For example, for computers running the Microsoft Windows operating system, right-click in the edit box and select **Undo** from the pop-up menu.

23-19

4 If you cut or copied the information to the clipboard and want to paste it, double-click the entry where you want to paste it and position the cursor at the new location in that edit box. Then use the standard paste technique for your platform.

- For computers running the Microsoft Windows operating system, right-click at the new location and select **Paste** from the pop-up menu.

- For UNIX based systems, click at the new location and then click the middle mouse button.

5 Click somewhere outside of the edit box to end editing mode.

Changing the Font Characteristics

You can change the font characteristics for the information in any cell. Specifically, you can specify the font size, style, color, and family.

1 Right-click the information in the cell.

The information in the cell is selected and the pop-up menu for changing font characteristics appears.

```
String...
Font Size      ▶
Font Style     ▶
Color...

Properties...
```
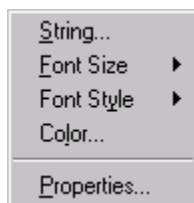
If this pop-up menu does not appear, it is because you were in edit mode. To get the font pop-up menu, click somewhere outside of the edit box surrounding the information and then right-click.

2 Select an item from the pop-up menu. Choose **Properties** if you want to change the font family or if you want to change multiple characteristics at once.

Note that you can also select **String** from the pop-up menu, which allows you to edit the text string.

3 Select the new font characteristic(s) for that cell. For example, for **Font Size**, select the new size from its pop-up menu.

Note that changing the font characteristics for the `%<blockdiagram>` entry is not relevant and does nothing.

Saving and Opening Print Frames

In this section...
"Saving a Print Frame" on page 23-22
"Opening a Print Frame" on page 23-22

Saving a Print Frame

You must save a print frame to print a block diagram with that print frame. To save a print frame:

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

The **Save As** dialog box appears.

2 Type a name for the print frame in the **File name** edit box.

3 Click the **Save** button.

The print frame is saved as a figure file, which has the .fig extension. A figure file is a binary file used for print frames.

Opening a Print Frame

You can open a saved print frame in the PrintFrame Editor, make changes to it, and save it under the same or a different name. To open an existing print frame:

1 Select **Open** from the **File** menu.

2 Select the print frame you want to open.

All print frames are figure files.

Alternatively, you can open a print frame from the MATLAB® prompt. Type `frameedit filename` and the PrintFrame Editor opens with the print frame file you specified.

Printing Block Diagrams with Print Frames

When using the Simulink® software or Stateflow® product, you can print a block diagram with the print frame.

1 Select **Print** from the **File** menu.

The **Print Model** dialog box appears. The dialog box shown below is for computers running the Microsoft® Windows® operating system. The dialog box for your platform might look slightly different.
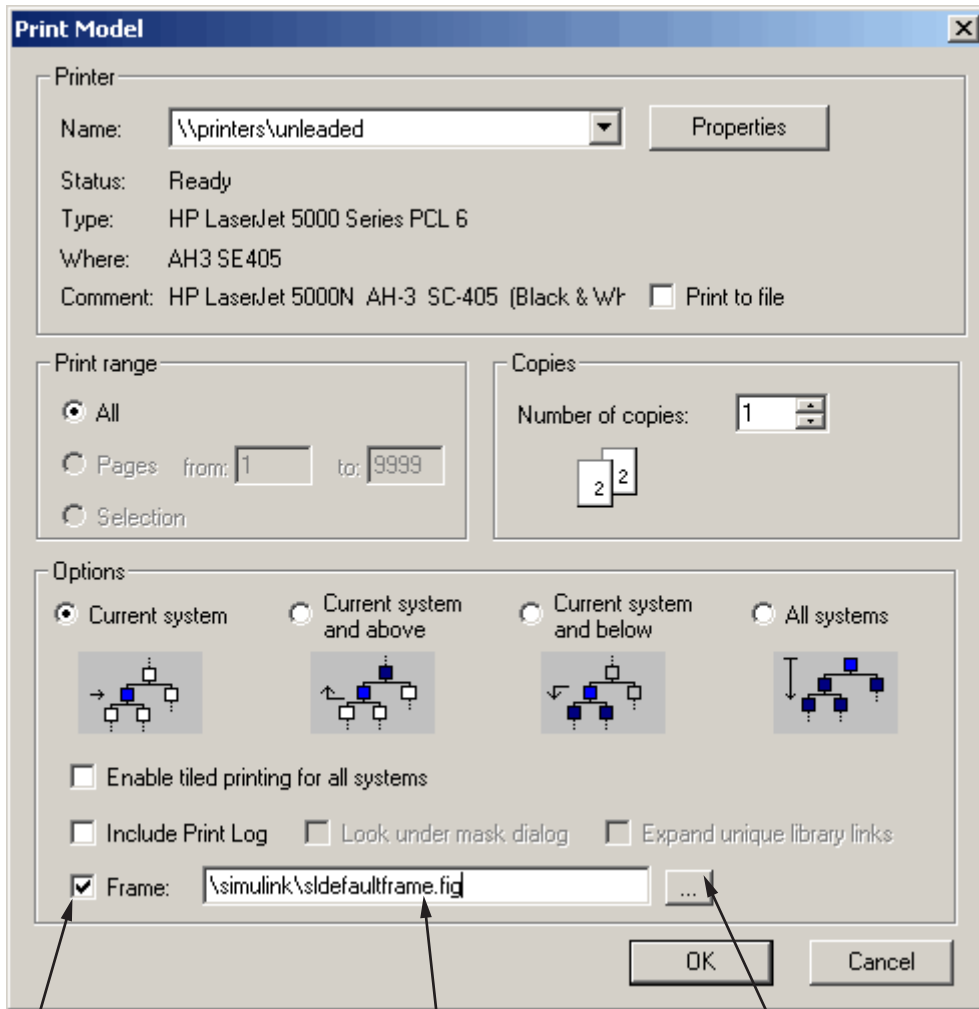
Check this box to print the block diagram with a print frame.

Type the path and filename of the print frame you want the block diagram to print with,

or click the ... button and then select the print frame file.

In the **Print Model** dialog box:

1 Select the **Frame** check box.

2 Supply the filename for the print frame you want to use. Either type the path and filename directly in the edit box, or click the **...** button and select the print frame file you saved using the PrintFrame Editor.

Note that the default print frame filename, `sldefaultframe.fig`, appears in the filename edit box until you specify a different filename.

3 Specify other printing options in the **Print** dialog box. For example, for computers running the Microsoft Windows operating system, specify options under **Properties**.

Note Specify the paper orientation for printing the way you normally would. The paper orientation you specified in the PrintFrame Editor's **PrintFrame Page Setup** dialog box is not the same as the paper orientation used for printing. For example, assume you specified a landscape-oriented print frame in the PrintFrame Editor. If you want the printed page to have a landscape orientation, you must specify that at the time of printing. For example, for computers running the Microsoft Windows operating system, click the **Properties** button in the **Print Model** dialog box, and for **Page Setup**, specify the **Orientation** as **Landscape**.

---

4 Click **OK** in the **Print** dialog box.

The block diagram prints with the print frame you specified.

See also the example, "Print the Block Diagram with the Print Frame" on page 23-30.

Example

In this section...
"About the Example" on page 23-26
"Create the Print Frame" on page 23-27
"Print the Block Diagram with the Print Frame" on page 23-30

About the Example

This example uses a Simulink® software demo engine model. It involves two parts — first creating a print frame, and then printing the engine model with that print frame. The result looks similar to the figure below.
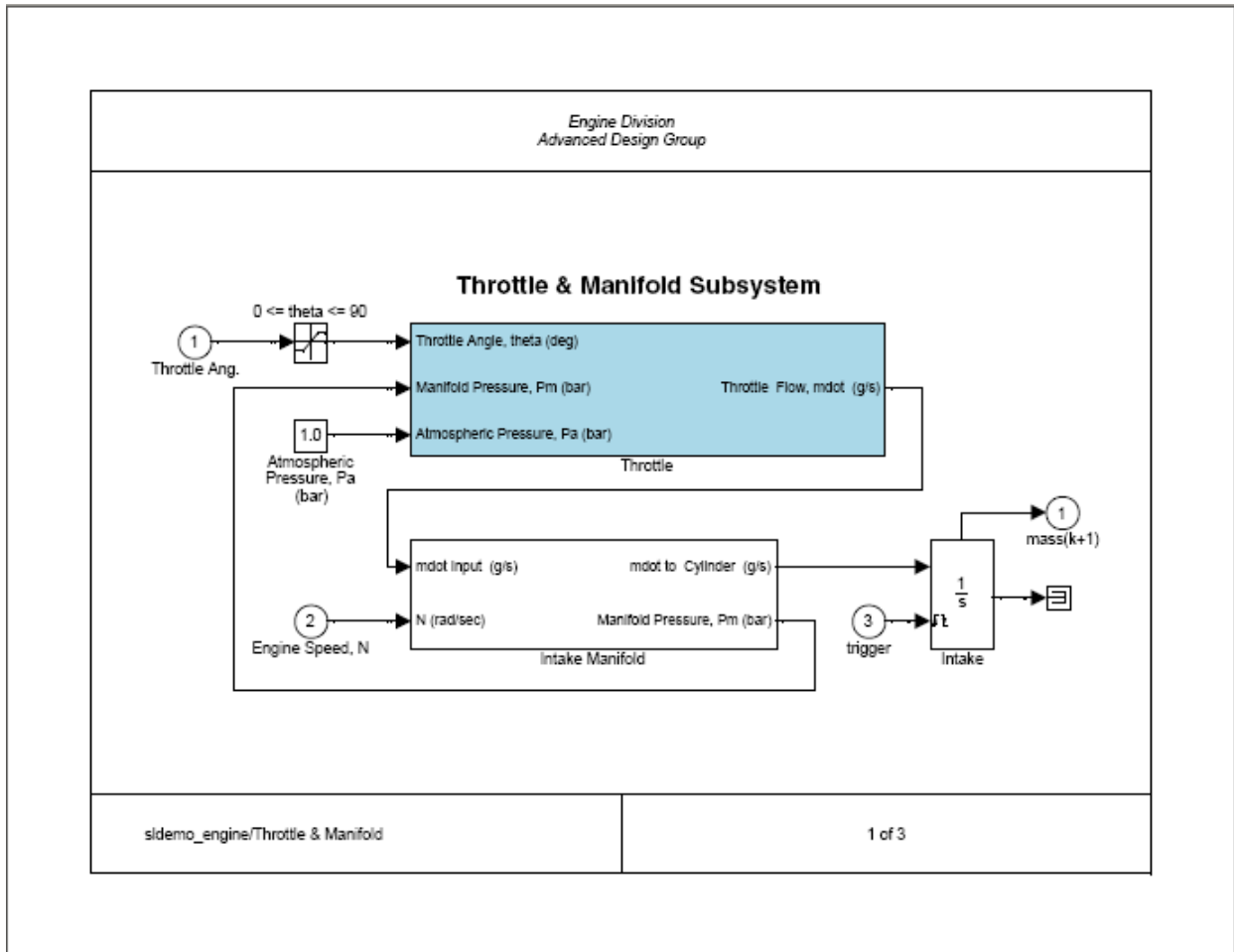
Create the Print Frame

1 At the MATLAB® prompt, type `frameedit`.

The **PrintFrame Editor** window appears.

2 Set up the page:

   a Select **Page Setup** from the **File** menu.

The **PrintFrame Page Setup** dialog box opens.

b For **Paper Type**, select a size that is appropriate for your printer.

c For this example, keep the **Paper Orientation** as **Landscape** and the **Margins** set to 0.75 **inches**.

d Click the **OK** button.

The dialog box closes. The print frame you see in the **PrintFrame Editor** window will reflect your changes.

3 Add the information entries %<blockdiagram>, %<fullsystem>, and %<page>.

a Click within the upper row in the print frame.

b From the bottom right list box, select Block Diagram, then click **Add**. The %<blockdiagram> appears in the row.

c Click within the lower left cell in the print frame.

d From the bottom right list box, select Full System Name, then click **Add**. The %<fullsystem> appears in the cell.

e Click within the lower right cell in the print frame.

f From the bottom right list box, select Page Number, then click **Add**. The %<page> appears in the cell.

4 Add a row at the top.

a Click within the upper row in the print frame, the row that contains the %<blockdiagram> entry.

Be sure that handles appear on all four corners of the row.

b Click the **add row** button.

A new row appears at the top, above the row you selected.

5 Make the new row shorter.

a Click on the horizontal line that separates the top row (the row you just added) from the row beneath it (the row containing the %<blockdiagram> entry).

Be sure that only two handles appear, one at each end of the line. If you see four handles in either row, click directly on the horizontal line and the other two handles disappear.

b Drag the line up until the top row is about the same height as the row at the bottom of the print frame.

6 Add information in the top row.

a Click anywhere within the top row (the row you just added).

b Select **Text** from the information list box.

c Click the **Add** button.

An edit box appears in the cell.

d Type `Engine Division`, press the **Enter** key to advance the cursor to the next line, and then type `Advanced Design Group`.

Click the zoom in button if you need to magnify the entry.

e Click outside of the edit box to end editing mode.

7 In the left cell of the bottom row, align the information on the left.

a Click the zoom out button if you need to.

b Click within the left cell of the bottom row to select it.

c Click the left alignment button.

The entry moves to the left.

8 Make the information in the top row appear in italics.

a Right-click on the entry in the row.

b Select **Font Style** from the pop-up menu.

If the pop-up menu for font properties does not appear, you are in editing mode. Click outside of the edit box to end editing mode and then right-click the text to access the pop-up menu.

c From the **Font Style** pop-up menu, select **italic**.

The entry in the cell appears in italics and the information will appear in italics when the print frame is printed with a Simulink diagram.

9 Add the total number of pages to the right cell in the bottom row.

   a Click within the cell to select it.

   b Add the total pages entry: select **Total Pages** from the information list box and click the **Add** button.

    The %<npages> entry appears after the %<page> entry. If you need to, zoom in to see the entry.

   c Add the text of after the page number entry. Click the cursor after the %<page> entry, and then type of (type a space before and after the word).

   The information in the cell now is: %<page> of %<npages>.

10 Save the print frame: select **Save As** from the **File** menu. In the **Save Frame** dialog box, type engdiv1 for the **File name**. Click the **Save** button.

The print frame is saved as a figure file.

11 You can close the **PrintFrame Editor** window by clicking the close box.

Print the Block Diagram with the Print Frame

1 To view the Simulink engine model, type sldemo_engine at the MATLAB prompt.

The engine model appears in a Simulink window.

2 Double-click the Throttle & Manifold block.
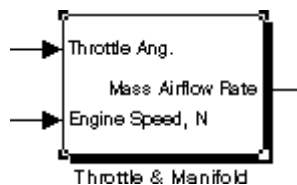
The Throttle & Manifold subsystem opens in a new window.

3 In the **Throttle & Manifold** window, select **Print** from the **File** menu.

The **Print Model** dialog box opens with the default settings as shown here.
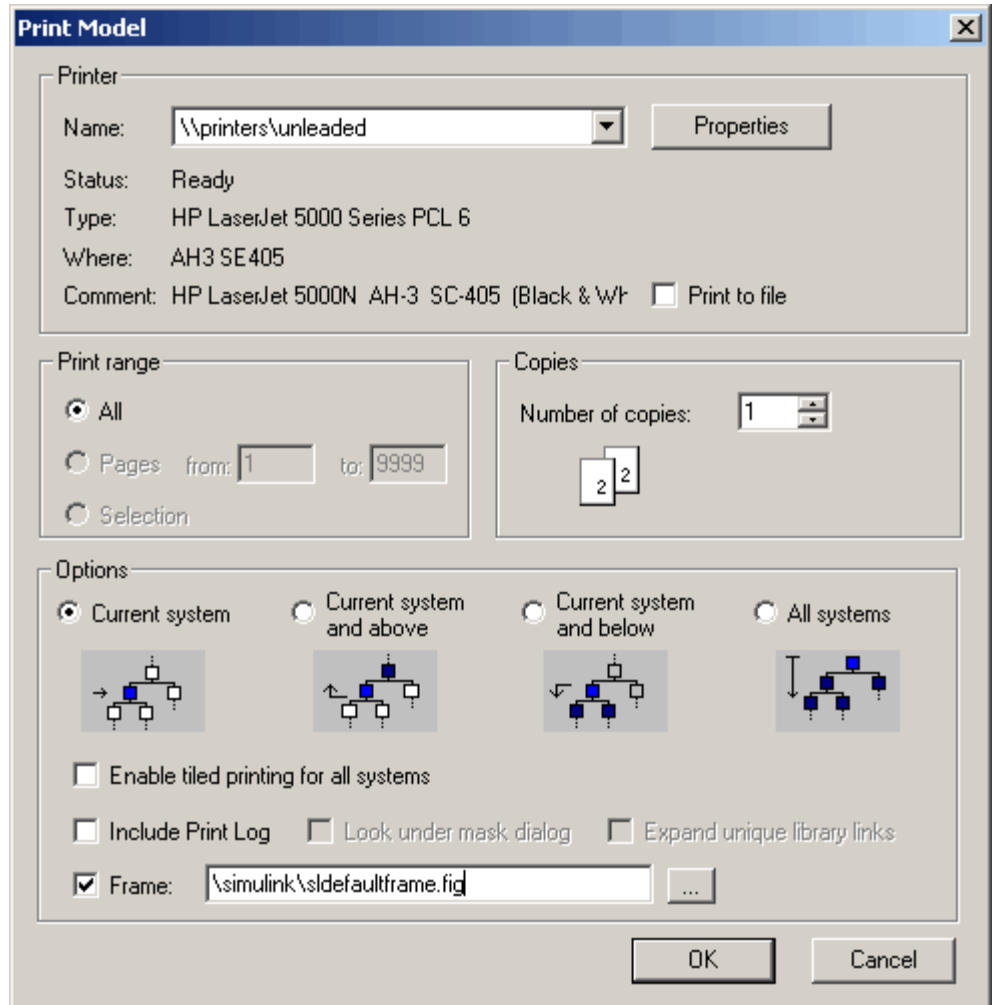
4 In the **Print Model** dialog box, set the page orientation to landscape. This example uses the techniques for computers running the Microsoft®

Windows® operating system. Use the methods for your own platform to change the page orientation for printing.

a Click the **Properties** button.

The **Document Properties** dialog box opens.

b Go to the **Page Setup** tab.

c For **Orientation**, select **Landscape**.

d Click **OK**.

The **Document Properties** dialog box closes.

5 In the **Print Model** dialog box, under **Options**, select **Current system and below**.

This specifies that the Throttle & Manifold block diagram and its subsystems will print.

6 Check the **Frame** check box.

7 Specify the print frame to use.

a Click the **...** button.

b In the **Frame File Selection** dialog box, find the filename of the print frame you just created, 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.

library

A collection of blocks intended to serve as prototypes for creating instances of block types in models. Simulink® software uses a special type of model file to store block libraries.

library block

A block in a library that serves as a prototype for creating instances of the block's type in models. You create an instance of the block type by copying the library block into a model.

library link

A connection between a library reference block and a library block that allows Simulink to update the reference block when the library block changes. Simulink creates the link when you create the reference block.
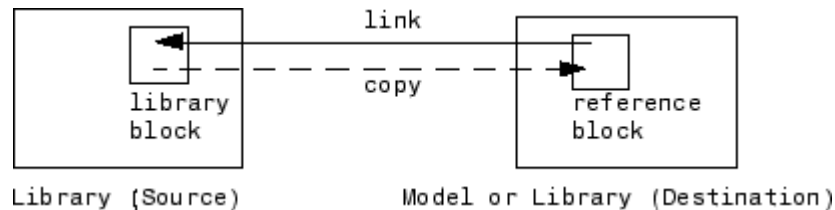
parameterized link

A link to a library block from a reference block whose parameter values differ from those of the library block. Parameterized links reference block parameter values to differ from library block parameter values without losing the advantage of automatic updating of reference blocks when library blocks change.

reference block

An instance of a block type in a model linked to a library block that serves as the block type's prototype.

sublibrary

A block library included in another library.

Examples

Use this list to find examples in the documentation.

Simulink Basics

How Simulink Works

Creating a Model

Working with Blocks

Working with Lookup Tables

Creating Block Masks

Creating Custom Simulink Blocks

Index

Symbols and Numerics