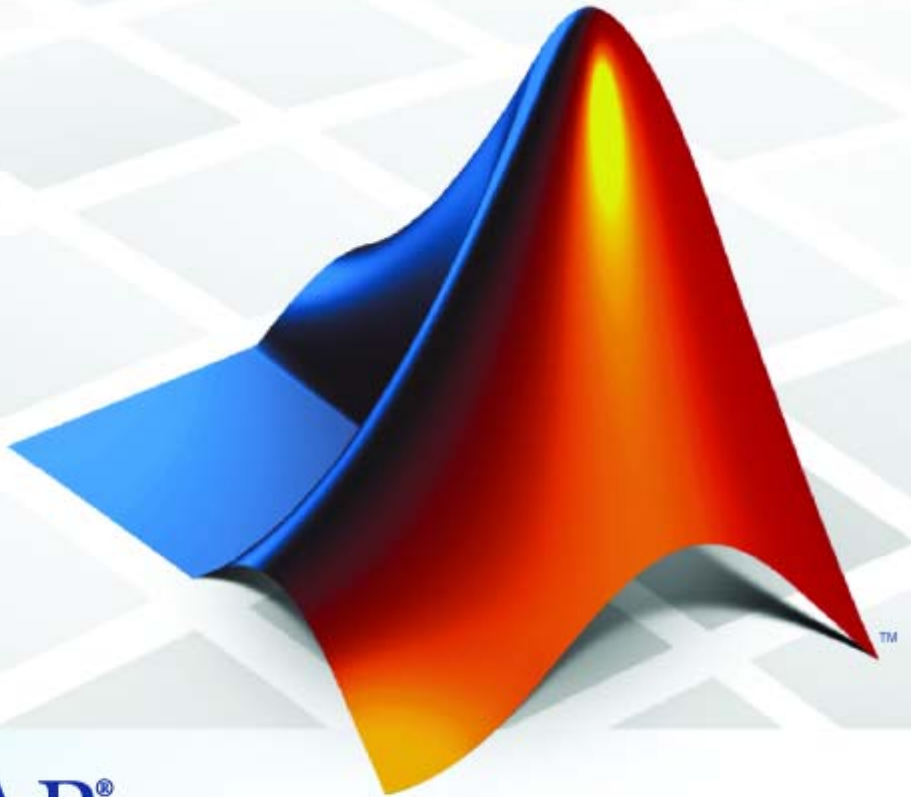


SimEvents[®] 3

User's Guide



MATLAB[®]
& **SIMULINK[®]**

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.

SimEvents® User's Guide

© COPYRIGHT 2005–2010 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to 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.

Revision History

November 2005	Online only	New for Version 1.0 (Release 14SP3+)
March 2006	Online only	Revised for Version 1.1 (Release 2006a)
September 2006	Online only	Revised for Version 1.2 (Release 2006b)
March 2007	Online only	Revised for Version 2.0 (Release 2007a)
September 2007	Online only	Revised for Version 2.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.2 (Release 2008a)
October 2008	Online only	Revised for Version 2.3 (Release 2008b)
March 2009	Online only	Revised for Version 2.4 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)

Working with Entities

1

Generating Entities When Events Occur	1-2
Overview	1-2
Sample Use Cases for Event-Based Generation of Entities	1-2
Specifying Generation Times for Entities	1-5
Overview	1-5
Procedure for Generating Entities at Specified Times	1-5
Setting Attributes of Entities	1-7
Role of Attributes in SimEvents Models	1-7
Blocks That Set Attributes	1-8
Attribute Value Support	1-9
Example: Setting Attributes	1-10
Manipulating Attributes of Entities	1-13
Writing Functions to Manipulate Attributes	1-13
Using Block Diagrams to Manipulate Attributes	1-15
Accessing Attributes of Entities	1-18
Counting Entities	1-20
Counting Departures Across the Simulation	1-20
Counting Departures per Time Instant	1-20
Resetting a Counter Upon an Event	1-22
Associating Each Entity with Its Index	1-24
Combining Entities	1-25
Overview of the Entity-Combining Operation	1-25
Example: Waiting to Combine Entities	1-25
Example: Copying Timers When Combining Entities	1-27
Example: Managing Data in Composite Entities	1-28

Replicating Entities on Multiple Paths	1-33
Sample Use Cases	1-33
Modeling Notes	1-33

Working with Events

2

Supported Events in SimEvents Models	2-2
Types of Supported Events	2-2
Signal-Based Events	2-4
Function Calls	2-8
Example: Event Calendar Usage for a Queue-Server	
Model	2-10
Overview of Example	2-10
Start of Simulation	2-11
Generation of First Entity	2-11
Generation of Second Entity	2-12
Completion of Service Time	2-13
Generation of Third Entity	2-14
Generation of Fourth Entity	2-15
Completion of Service Time	2-16
Observing Events	2-18
Techniques for Observing Events	2-18
Example: Observing Service Completions	2-22
Example: Detecting Collisions by Comparing Events	2-25
Generating Function-Call Events	2-28
Role of Explicitly Generated Events	2-28
Generating Events When Other Events Occur	2-28
Generating Events Using Intergeneration Times	2-30
Manipulating Events	2-32
Reasons to Manipulate Events	2-32
Blocks for Manipulating Events	2-34
Creating a Union of Multiple Events	2-34
Translating Events to Control the Processing Sequence ..	2-37
Conditionalizing Events	2-39

Managing Simultaneous Events

3

Overview of Simultaneous Events	3-2
Exploring Simultaneous Events	3-4
Using Nearby Breakpoints to Focus on a Particular Time	3-5
For Further Information	3-5
Choosing an Approach for Simultaneous Events	3-7
Assigning Event Priorities	3-8
Procedure for Assigning Event Priorities	3-8
Tips for Choosing Event Priority Values	3-8
Procedure for Specifying Equal-Priority Behavior	3-9
Example: Choices of Values for Event Priorities	3-11
Overview of Example	3-11
Arbitrary Resolution of Signal Updates	3-12
Selecting a Port First	3-12
Generating Entities First	3-19
Randomly Selecting a Sequence	3-24
Example: Effects of Specifying Event Priorities	3-26
Overview of the Example	3-26
Default Behavior	3-27
Deferring Gate Events	3-28
Deferring Subsystem Execution to the Event Calendar ...	3-30

Working with Signals

4

Role of Event-Based Signals in SimEvents Models	4-2
Overview of Event-Based Signals	4-2
Comparison with Time-Based Signals	4-2

Generating Random Signals	4-4
Generating Random Event-Based Signals	4-4
Examples of Random Event-Based Signals	4-5
Generating Random Time-Based Signals	4-6
Using Data Sets to Create Event-Based Signals	4-9
Behavior of the Event-Based Sequence Block	4-9
Generating Sequences Based on Arbitrary Events	4-10
Manipulating Signals	4-12
Specifying Initial Values of Event-Based Signals	4-12
Example: Resampling a Signal Based on Events	4-13
Sending Data to the MATLAB Workspace	4-15
Behavior of the Discrete Event Signal to Workspace Block	4-15
Example: Sending Queue Length to the Workspace	4-15
Using the To Workspace Block with Event-Based Signals	4-18
Working with Multivalued Signals	4-19
Zero-Duration Values of Signals	4-19
Importance of Zero-Duration Values	4-20
Detecting Zero-Duration Values	4-20

Modeling Queues and Servers

5

Using a LIFO Queuing Discipline	5-2
Overview of LIFO Queues	5-2
Example: Waiting Time in LIFO Queue	5-2
Sorting by Priority	5-5
Behavior of the Priority Queue Block	5-5
Example: FIFO and LIFO as Special Cases of a Priority Queue	5-5
Example: Serving Preferred Customers First	5-8

Preempting an Entity in a Server	5-11
Definition of Preemption	5-11
Criteria for Preemption	5-11
Residual Service Time	5-12
Queuing Disciplines for Preemptive Servers	5-12
Example: Preemption by High-Priority Entities	5-12
Determining Whether a Queue Is Nonempty	5-17
Modeling Multiple Servers	5-18
Blocks that Model Multiple Servers	5-18
Example: M/M/5 Queuing System	5-18
Modeling the Failure of a Server	5-20
Server States	5-20
Using a Gate to Implement a Failure State	5-20
Using Stateflow Charts to Implement a Failure State	5-21

Routing Techniques

6

Output Switching Based on a Signal	6-2
Specifying an Initial Port Selection	6-2
Using the Storage Option to Prevent Latency Problems ..	6-2
Example: Cascaded Switches with Skewed Distribution	6-6
Example: Compound Switching Logic	6-7

Using Logic

7

Role of Logic in SimEvents Models	7-2
--	------------

Using Embedded MATLAB Function Blocks for Logic	7-3
Overview of Use of Embedded MATLAB Function Blocks	7-3
Example: Choosing the Shortest Queue	7-3
Example: Varying Fluid Flow Rate Based on Batching Logic	7-6
Using Logic Blocks	7-10
Overview of Use of Logic Blocks	7-10
Example: Using Servers in Shifts	7-11
Example: Choosing the Shortest Queue Using Logic Blocks	7-14
For Further Examples	7-15

Regulating Arrivals Using Gates

8

Role of Gates in SimEvents Models	8-2
Overview of Gate Behavior	8-2
Types of Gate Blocks	8-3
Keeping a Gate Open Over a Time Interval	8-4
Behavior of Enabled Gate Block	8-4
Example: Controlling Joint Availability of Two Servers ..	8-4
Opening a Gate Instantaneously	8-6
Behavior of Release Gate Block	8-6
Example: Synchronizing Service Start Times with the Clock	8-6
Example: Opening a Gate Upon Entity Departures	8-7
Combining Gates	8-9
Effect of Combining Gates	8-9
Example: First Entity as a Special Case	8-11

Forcing Departures Using Timeouts

9

Role of Timeouts in SimEvents Models	9-2
Basic Example Using Timeouts	9-3
Basic Procedure for Using Timeouts	9-4
Schematic Illustrating Procedure	9-4
Step 1: Designate the Entity Path	9-5
Step 2: Specify the Timeout Interval	9-5
Step 3: Specify Destinations for Timed-Out Entities	9-6
Defining Entity Paths on Which Timeouts Apply	9-7
Linear Path for Timeouts	9-7
Branched Path for Timeouts	9-8
Feedback Path for Timeouts	9-8
Handling Entities That Time Out	9-10
Common Requirements for Handling Timed-Out Entities	9-10
Techniques for Handling Timed-Out Entities	9-10
Example: Dropped and Timed-Out Packets	9-11
Example: Rerouting Timed-Out Entities to Expedite Handling	9-12
Example: Limiting the Time Until Service Completion	9-14

Controlling Timing with Subsystems

10

Timing Issues in SimEvents Models	10-2
Overview of Timing Issues	10-2
Timing for the End of the Simulation	10-2
Timing for a Statistical Computation	10-3
Timing for Choosing a Port Using a Sequence	10-4

Role of Discrete Event Subsystems in SimEvents	
Models	10-7
Overview of Discrete Event Subsystems	10-7
Purpose of Discrete Event Subsystems	10-8
Processing Sequence for Events in Discrete Event Subsystems	10-8
Blocks Inside Discrete Event Subsystems	10-10
Working with Discrete Event Subsystem Blocks	10-11
Setting Up Signal-Based Discrete Event Subsystems	10-11
Signal-Based Events That Control Discrete Event Subsystems	10-14
Examples Using Discrete Event Subsystem Blocks	10-18
Example: Comparing the Lengths of Two Queues	10-18
Example: Normalizing a Statistic to Use for Routing	10-19
Example: Ending the Simulation Upon an Event	10-21
Example: Sending Unrepeated Data to the MATLAB Workspace	10-22
Example: Focusing on Events, Not Values	10-23
Example: Detecting Changes from Empty to Nonempty ..	10-24
Example: Logging Data About the First Entity on a Path	10-25
Creating Entity-Departure Subsystems	10-27
Overview of Entity-Departure Subsystems	10-27
Accessing Blocks for Entity-Departure Subsystems	10-28
Setting Up Entity-Departure Subsystems	10-29
Examples Using Entity-Departure Subsystems	10-30
Example: Using Entity-Based Timing for Choosing a Port	10-30
Example: Performing a Computation on Selected Entity Paths	10-31
Using Function-Call Subsystems	10-33
Using Function-Call Subsystems in Discrete-Event Simulations	10-33
Use Cases for Function-Call Subsystems	10-33

Setting Up Function-Call Subsystems in SimEvents Models	10-34
--	-------

Plotting Data

11

Choosing and Configuring Plotting Blocks	11-2
Sources of Data for Plotting	11-2
Inserting and Connecting Scope Blocks	11-3
Connections Among Points in Plots	11-4
Varying Axis Limits Automatically	11-5
Caching Data in Scopes	11-6
Examples Using Scope Blocks	11-6
 Working with Scope Plots	11-8
Customizing Plots	11-8
Exporting Plots	11-9
 Using Plots for Troubleshooting	11-10
 Example: Plotting Entity Departures to Verify	
Timing	11-11
Overview of Example	11-11
Model Exhibiting Correct Timing	11-11
Model Exhibiting Latency	11-13
 Example: Plotting Event Counts to Check for	
Simultaneity	11-15
 Comparison with Time-Based Plotting Tools	11-17

Role of Statistics in Discrete-Event Simulation	12-2
Overview	12-2
Statistics for Data Analysis	12-2
Statistics for Run-Time Control	12-3
Accessing Statistics from SimEvents Blocks	12-5
Statistics-Related Parameters in Block Dialog Boxes	12-5
Accessing Statistics Throughout the Simulation	12-6
Accessing Statistics When Stopping or Pausing Simulation	12-7
Deriving Custom Statistics	12-8
Overview of Approaches to Custom Statistics	12-8
Graphical Block-Diagram Approach	12-8
Coded Approach	12-9
Post-Simulation Analysis	12-9
Example: Fraction of Dropped Messages	12-9
Example: Computing a Time Average of a Signal	12-11
Example: Resetting an Average Periodically	12-13
Using Timers	12-20
Overview of Timers	12-20
Basic Example Using Timer Blocks	12-21
Basic Procedure for Using Timer Blocks	12-22
Timing Multiple Entity Paths with One Timer	12-23
Restarting a Timer from Zero	12-24
Timing Multiple Processes Independently	12-26
Varying Simulation Results by Managing Seeds	12-28
Connection Between Random Numbers and Seeds	12-28
Making Results Repeatable by Storing Sets of Seeds	12-29
Setting Seed Values Programmatically	12-30
Sharing Seeds Among Models	12-30
Working with Seeds Not in SimEvents Blocks	12-31
Choosing Seed Values	12-34
Regulating the Simulation Length	12-35
Overview	12-35

Setting a Fixed Stop Time	12-35
Stopping Upon Processing a Fixed Number of Entities ...	12-36
Stopping Upon Reaching a Particular State	12-37

Using Stateflow Charts in SimEvents Models

13

Role of Stateflow Charts in SimEvents Models	13-2
Guidelines for Using Stateflow and SimEvents	
Blocks	13-3
Examples Using Stateflow Charts and SimEvents	
Blocks	13-4
Failure State of Server	13-4
Go-Back-N ARQ Model	13-4

Debugging Discrete-Event Simulations

14

Overview of Debugging Resources	14-2
Overview of the SimEvents Debugger	14-3
Starting the SimEvents Debugger	14-5
The Debugger Environment	14-7
Debugger Command Prompt	14-7
Simulation Log in the Debugger	14-8
Identifiers in the Debugger	14-18
Independent Operations and Consequences in the	
Debugger	14-20
Significance of Independent Operations	14-20

Independent Operations	14-20
Consequences of Independent Operations	14-21
Stopping the Debugger	14-24
How to End the Debugger Session	14-24
Comparison of Simulation Control Functions	14-24
Stepping Through the Simulation	14-26
Overview of Stepping	14-26
How to Step	14-27
Choosing the Granularity of a Step	14-28
Tips for Stepping Through the Simulation	14-29
Inspecting the Current Point in the Debugger	14-31
Viewing the Current Operation	14-31
Obtaining Information Associated with the Current Operation	14-32
Inspecting Entities, Blocks, and Events	14-34
Inspecting Entities	14-34
Inspecting Blocks	14-36
Inspecting Events	14-38
Obtaining Identifiers of Entities, Blocks, and Events	14-38
Working with Debugging Information in Variables ...	14-41
Comparison of Variables with Inspection Displays	14-41
Functions That Return Debugging Information in Variables	14-41
How to Create Variables Using State Inspection Functions	14-42
Tips for Manipulating Structures and Cell Arrays	14-43
Example: Finding the Number of Entities in Busy Servers	14-43
Viewing the Event Calendar	14-46
For Further Information	14-46
Customizing the Debugger Simulation Log	14-47
Customizable Information in the Simulation Log	14-47
Tips for Choosing Appropriate Detail Settings	14-48
Effect of Detail Settings on Stepping	14-49

How to View Current Detail Settings	14-51
How to Change Detail Settings	14-52
How to Save and Restore Detail Settings	14-53
Debugger Efficiency Tips	14-54
Executing Commands Automatically When the Debugger Starts	14-54
Creating Shortcuts for Debugger Commands	14-55
Defining a Breakpoint	14-56
What Is a Breakpoint?	14-56
Identifying a Point of Interest	14-56
Setting a Breakpoint	14-58
Viewing All Breakpoints	14-60
Using Breakpoints During Debugging	14-62
Running the Simulation Until the Next Breakpoint	14-62
Ignoring or Removing Breakpoints	14-63
Enabling a Disabled Breakpoint	14-64
Block Operations Relevant for Block Breakpoints	14-65
Common Problems in SimEvents Models	14-72
Unexpectedly Simultaneous Events	14-72
Unexpectedly Nonsimultaneous Events	14-73
Unexpected Processing Sequence for Simultaneous Events	14-73
Time-Based Block Not Recognizing Certain Trigger Edges	14-74
Incorrect Timing of Signals	14-74
Unexpected Use of Old Value of Signal	14-76
Effect of Initial Condition on Signal Loops	14-80
Loops in Entity Paths Without Sufficient Storage Capacity	14-83
Unexpected Timing of Random Signal	14-86
Unexpected Correlation of Random Processes	14-88
Recognizing Latency in Signal Updates	14-90

Running Discrete-Event Simulations Programmatically

15

Accelerating Discrete-Event Simulations Using Rapid Simulation	15-3
Objective	15-3
Prerequisites	15-3
Procedure	15-3
See Also	15-4
Sharing Executables for Discrete-Event Simulations ..	15-5
Objective	15-5
Model Designer and Model User Roles	15-5
Procedure for Model Designer	15-5
Procedure for Model User	15-7
Prerequisites for Using Generated Code	15-9
Choice of Environment for Varying Parameters	
Between Simulation Runs	15-10
Comparison of Approaches	15-10
Adapting Existing Models and MATLAB Code to Work with Rapid Simulation	15-11
Varying Parameters Between Simulation Runs Using MATLAB Code	15-13
Objective	15-13
Procedure	15-13
See Also	15-14
Varying Parameters Between Rapid Simulation Runs	15-15
Objective	15-15
Prerequisites	15-15
Procedure	15-15
For Further Information	15-17
Designing Models to Accept Event-Based Data During Rapid Simulation	15-18

Event-Based Data for Rapid Simulations	15-18
Varying the Stream of Random Numbers	15-21
Varying Intergeneration Times	15-21
Varying Attribute Values	15-22
Varying Service Times	15-23
Varying Timeout Intervals	15-24
Varying Event-Based Signals	15-25
Example: Computing an Ensemble Average Using	
MATLAB Code	15-26
Example: Varying the Number of Servers Using	
MATLAB Code	15-29
Example: Computing an Ensemble Average Using	
Rapid Simulation	15-32
Example Scenario	15-32
Prerequisites	15-32
Steps in the Example	15-32
Results	15-35
Example: Varying Attribute Values Between Runs	
Using Rapid Simulation	15-37
Example Scenario	15-37
Prerequisites	15-37
Steps in the Example	15-37
Results	15-42
Example: Varying Queue Capacity Between Runs Using	
Rapid Simulation	15-43
Example Scenario	15-43
Prerequisites	15-43
Steps in the Example	15-43
Results	15-47
Limitations of Rapid Simulation for Discrete-Event	
Simulations	15-48

Event Sequencing	16-2
Processing Sequence for Simultaneous Events	16-2
Role of the Event Calendar	16-3
For Further Information	16-5
Choosing How to Resolve Simultaneous Signal Updates	16-7
Resolution Sequence for Input Signals	16-8
Detection of Signal Updates	16-8
Effect of Simultaneous Operations	16-9
Resolving the Set of Operations	16-10
Specifying Event Priorities to Resolve Simultaneous Signal Updates	16-10
Resolving Simultaneous Signal Updates Without Specifying Event Priorities	16-12
For Further Information	16-15
Livelock Prevention	16-16
Overview	16-16
Permitting Large Finite Numbers of Simultaneous Events	16-16
Notifications and Queries Among Blocks	16-18
Overview of Notifications and Queries	16-18
Querying Whether a Subsequent Block Can Accept an Entity	16-18
Notifying Blocks About Status Changes	16-19
Notifying, Monitoring, and Reactive Ports	16-21
Overview of Signal Input Ports of SimEvents Blocks	16-21
Notifying Ports	16-21
Monitoring Ports	16-22
Reactive Ports	16-23
Interleaving of Block Operations	16-25
Overview of Interleaving of Block Operations	16-25

How Interleaving of Block Operations Occurs	16-25
Example: Sequence of Departures and Statistical Updates	16-26
Zero-Duration Values and Time-Based Blocks	16-31
Techniques for Working with Multivalued Signals	16-31
Example: Using a #n Signal as a Trigger	16-32
Update Sequence for Output Signals	16-34
Determining the Update Sequence	16-34
Example: Detecting Changes in the Last-Updated Signal	16-35
Storage and Nonstorage Blocks	16-37
Storage Blocks	16-37
Nonstorage Blocks	16-37

Examples

A

Attributes of Entities	A-2
Counting Entities	A-2
Working with Events	A-2
Queuing Systems	A-2
Working with Signals	A-3
Server States	A-3
Routing Entities	A-3
Batching	A-3

Gates	A-3
Timeouts	A-4
Discrete Event Subsystems	A-4
Troubleshooting	A-4
Statistics	A-5
Timers	A-5
Rapid Simulation	A-5

Index

Working with Entities

- “Generating Entities When Events Occur” on page 1-2
- “Specifying Generation Times for Entities” on page 1-5
- “Setting Attributes of Entities” on page 1-7
- “Manipulating Attributes of Entities” on page 1-13
- “Accessing Attributes of Entities” on page 1-18
- “Counting Entities” on page 1-20
- “Combining Entities” on page 1-25
- “Replicating Entities on Multiple Paths” on page 1-33

Generating Entities When Events Occur

In this section...
“Overview” on page 1-2
“Sample Use Cases for Event-Based Generation of Entities” on page 1-2

Overview

The Event-Based Entity Generator block enables you to generate entities in response to signal-based events that occur during the simulation. The **Generate entities upon** parameter of the block determines:

- The kind of signal input port the block has
- The kinds of events that cause the block to generate an entity

Event times and the time intervals between pairs of successive entities are not necessarily predictable in advance.

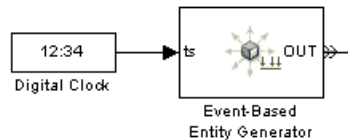
Note The Event-Based Entity Generator block can respond to triggers and function calls. However, do not place the block inside a triggered or function-call subsystem. Like other blocks that possess entity ports, the Event-Based Entity Generator block is not valid inside a triggered or function-call subsystem.

To specify intergeneration times between pairs of successive entities instead of using an event-based approach, use the Time-Based Entity Generator block. For more information, see “Creating Entities Using Intergeneration Times” in the SimEvents® getting started documentation.

Sample Use Cases for Event-Based Generation of Entities

Generating entities when events occur is appropriate if you want the dynamics of your model to determine when to generate entities. For example,

- To generate an entity each time the length of a queue changes, configure the queue to output a signal indicating the queue length. Then configure the Event-Based Entity Generator block to react to changes in the value of that signal.
- To generate entities periodically, configure the Event-Based Entity Generator block to react to sample time hits of a discrete-time signal with an explicit **Sample time** parameter. Regardless of the value of the signal, the block generates an entity periodically, according to the sample time of the driving block.

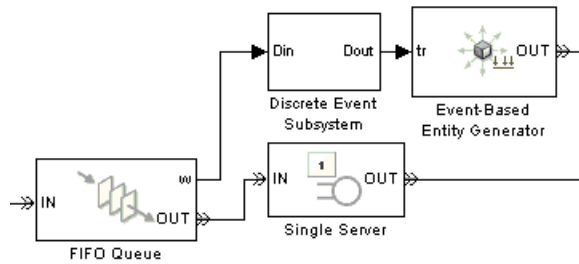


Another way to generate entities periodically is to use a Time-Based Entity Generator block with **Distribution** set to **Constant**. The advantage of using the Event-Based Entity Generator is that it more directly synchronizes entity generation events with sample time hits and avoids possible roundoff errors.

For other examples that use this entity-generation method to create desired simultaneity of events, see “Example: Choices of Values for Event Priorities” on page 3-11.

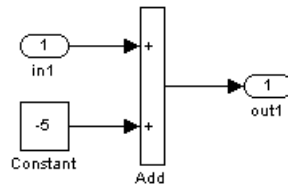
- To generate an entity each time a Stateflow® chart transitions from state A to state B, configure the chart to output a function call upon such a transition. Then configure the Event-Based Entity Generator block to react to each function call by generating an entity.
- To generate an entity each time a real-valued statistical signal crosses a threshold, configure the Event-Based Entity Generator block to react to triggers. Connect the **tr** signal input port of the block to the statistical signal minus the threshold. If the statistical signal is an event-based signal, then perform the subtraction in a discrete event subsystem; see “Timing Issues in SimEvents Models” on page 10-2 for details.

In the following figure, the Event-Based Entity Generator block generates a new entity each time the average waiting time of the queue crosses a threshold. The threshold is 5 s.



Top-Level Model

The subsystem between the queue and the entity generator adds -5 to the average waiting time value to translate the threshold from 5 to 0.



Subsystem Contents

- To generate multiple entities simultaneously, configure the Event-Based Entity Generator block to react to function calls. Then connect its **fcn** input port to a signal that represents multiple function calls. For an example, see the Preload Queue with Entities demo.

Note If you generate multiple entities simultaneously, then consider the appropriateness of other blocks in the model. For example, if three simultaneously generated entities advance to a single server, then consider inserting a queue between the generator and the server. As a result, entities (in particular, the second and third entities) have a place to wait for the server to become available.

Specifying Generation Times for Entities

In this section...

“Overview” on page 1-5

“Procedure for Generating Entities at Specified Times” on page 1-5

Overview

If you have a list of unique times, you can configure the Time-Based Entity Generator block to generate entities at these times. Explicit entity-generation times are useful if you want to

- Recreate an earlier simulation whose intergeneration times you saved using a Discrete Event Signal to Workspace block.
- Study the behavior of your model under unusual circumstances and have created a series of entity generation times that you expect to produce unusual circumstances.
- Verify simulation behavior that you or someone else observed elsewhere, such as a result reported in a paper.

Procedure for Generating Entities at Specified Times

To generate entities at specified times, follow this procedure:

- 1** In the Time-Based Entity Generator block, set the **Generate entities with** parameter to Intergeneration time from port t . A signal input port labeled t appears on the block.
- 2** Depending on whether you want to generate an entity at $T=0$, either select or clear the **Generate entity at simulation start** option.
- 3** Create a column vector, `gentimes`, that lists 0 followed by the nonzero times at which you want to create entities, in strictly ascending sequence. You can create this vector using one of these techniques:
 - Enter the definition in the MATLAB® Command Window
 - Load a MAT-file that you previously created

- Manipulate a variable that a To Workspace or Discrete Event Signal to Workspace block previously created

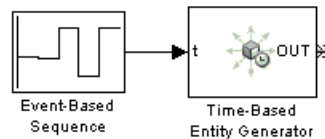
An example of a column vector listing generation times is:

```
gentimes = [0; 0.9; 1.7; 3.8; 3.9; 6];
```

- 4 Apply the `diff` function to the vector of generation times, thus creating a vector of intergeneration times.

```
intergentimes = diff(gentimes);
```

- 5 Insert an Event-Based Sequence block in the model and connect it to the `t` input port of the Time-Based Entity Generator block.



- 6 In the Event-Based Sequence block, set **Vector of output values** to `intergentimes`. Set the **Form output after final data value by** parameter to `Setting to infinity`. The `Setting to infinity` option halts the generation process if the simulation time exceeds your maximum generation time.

Setting Attributes of Entities

In this section...

“Role of Attributes in SimEvents Models” on page 1-7

“Blocks That Set Attributes” on page 1-8

“Attribute Value Support” on page 1-9

“Example: Setting Attributes” on page 1-10

Role of Attributes in SimEvents Models

You can attach data to an entity using one or more *attributes* of the entity. Each attribute has a name and a numeric value. You can read or change the values of attributes during the simulation.

For example, suppose your entities represent a message that you are transmitting across a communication network. You can attach the length of each particular message to the message itself, using an attribute named *length*.

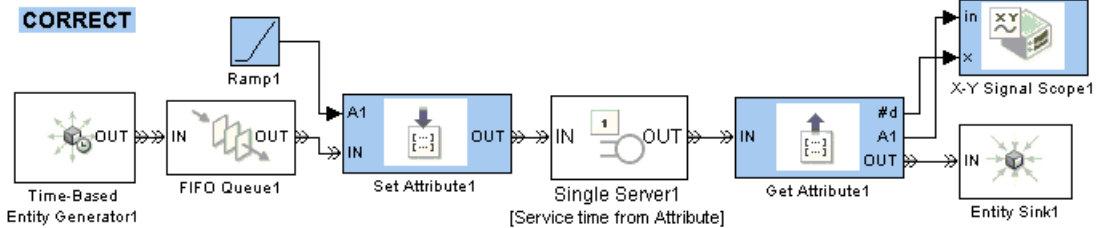
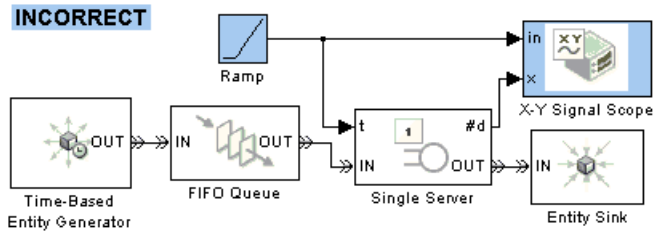
In some modeling situations, it is important to *attach* data to an entity instead of merely creating the data as the content of a signal. The next example shows the importance of considering not only the topology of your block diagrams, but also the timing of data signals.

Example: Reusing Data

Consider a queue-server example with varying service times. Suppose you want to plot the service time against entity count for each entity departing from the server. A signal specifies the service time to use for each entity. Although connecting the same signal to the Signal Scope block appears correct topologically, the timing in such an arrangement is incorrect. The incorrectness arises from the delay at the server. That is, the signal has one value when a given entity arrives at the server and another value when the same entity arrives at the scope.

To implement the example correctly, attach the service time to each entity using an attribute and retrieve the attribute value when needed from each entity. That way, the scope receives the service time associated with each

entity, regardless of the delay between arrival times at the server and the scope.



Blocks That Set Attributes

To assign attributes on each arriving entity, use one of the blocks in the next table.

Block	Attribute Value	For More Information
Attribute Function	Value of output argument from a function you write	“Writing Functions to Manipulate Attributes” on page 1-13
Set Attribute	Signal value or information you enter in the block dialog box	Set Attribute block reference page

Block	Attribute Value	For More Information
Entity Departure Counter	Entity count, only if you set Write count to attribute to On.	“Associating Each Entity with Its Index” on page 1-24
Single Server	Residual service time, only for preempted entities	“Preempting an Entity in a Server” on page 5-11

Assignments can create new attributes or change the values of existing attributes.

To learn how to query entities for attribute values, see “Accessing Attributes of Entities” on page 1-18. To learn how to aggregate attributes from distinct entities, see “Combining Entities” on page 1-25.

Attribute Value Support

These lists summarize the characteristics of attribute values.

Permitted Characteristics of Attribute Values

- Real or complex
- Array of any dimension
- double data type

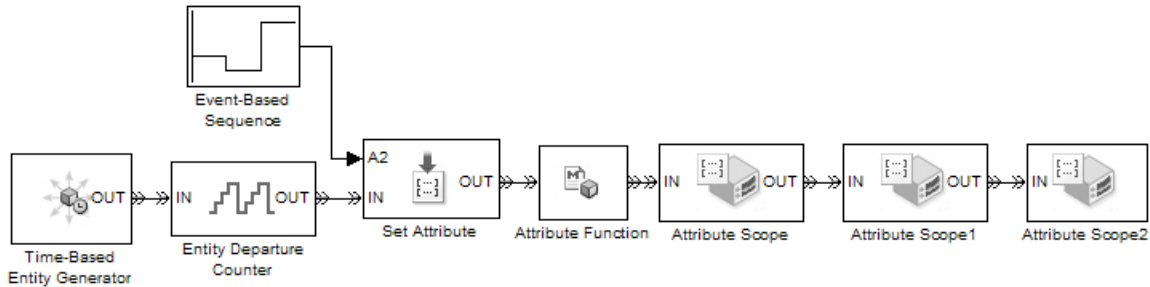
For a given attribute, the characteristics of the value must be consistent throughout the model.

Not Permitted as Attribute Values

- Frame
- Structure
- Data types other than double. In particular, strings are not valid as attribute values.
- Buses

Example: Setting Attributes

This example illustrates different ways of assigning attribute values to entities.

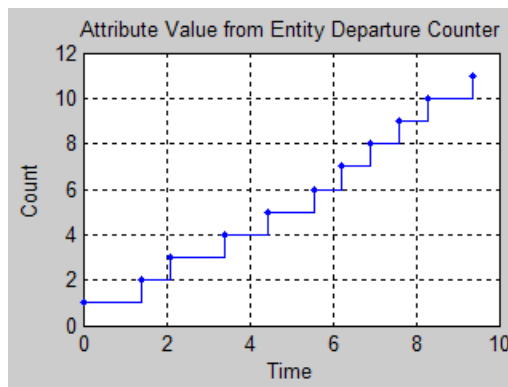


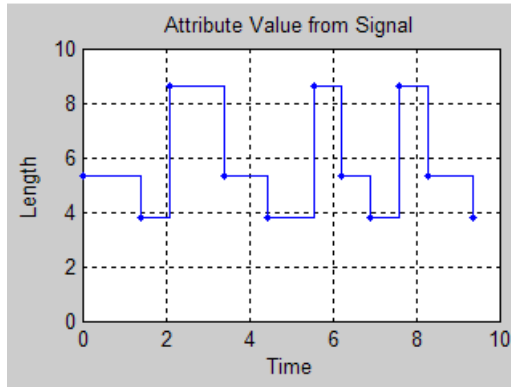
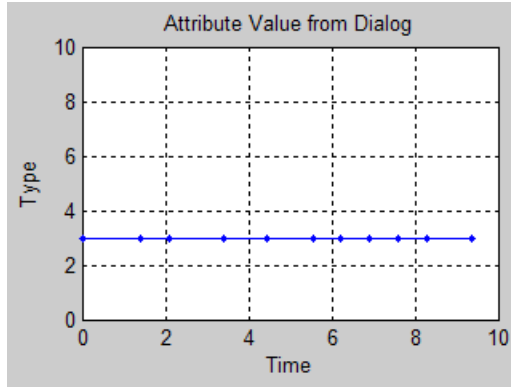
After each entity departs from the Attribute Function block, it possesses the attributes listed in the table.

Attribute Name	Attribute Value	Method for Setting Attribute Value
Count	N, for the Nth entity departing from the Time-Based Entity Generator block	In Entity Departure Counter dialog box: Write count to attribute = On Attribute name = Count Actually, the entity generator creates the Count attribute with a value of 0. The Entity Departure Counter block sets the attribute value according to the entity count.
Type	Constant value of 3	A1 row of table in Set Attribute dialog box: Name = Type Value From = Dialog Value = 3

Attribute Name	Attribute Value	Method for Setting Attribute Value
Length	Next number in the sequence produced by Event-Based Sequence block	Event-Based Sequence block connected to Set Attribute block in which A2 row of table in dialog box is configured as follows: Name = Length Value From = Signal port
LengthInt	floor(Length)	Attribute Function block whose function is function [out_LengthInt] = fcn(Length) out_LengthInt = floor(Length);

In this example, each Attribute Scope block plots values of a different attribute over time. Notice from the vertical axes of the plots below that the Count values increase by 1 with each entity, the Type values are constant, and the Length values show cyclic repetition of a sequence. For brevity, the example does not plot LengthInt values, which would look similar to Length values.





Manipulating Attributes of Entities

In this section...
“Writing Functions to Manipulate Attributes” on page 1-13
“Using Block Diagrams to Manipulate Attributes” on page 1-15

Writing Functions to Manipulate Attributes

To manipulate attributes via code, use the Attribute Function block. The block lets you access existing attributes of the arriving entity, modify the values of existing attributes, or create new attributes.

For examples that use the block, see

- “Example: Setting Attributes” on page 1-10
- “Attribute Names Unique and Accessible in Composite Entity” on page 1-29.
- Entity Combiner for Assembling Components demo
- Distributed Processing for Multi-Class Jobs demo, within the Distribution Center subsystem and its Service History Monitor subsystem

Procedure for Using the Attribute Function Block

The Attribute Function block has one entity input port and one entity output port. The block manipulates attributes of each arriving entity. To edit the computation, use this procedure:

- 1** Display the block’s associated function in an editor window by double-clicking the Attribute Function block.
- 2** Write the first line of the function using arguments that reflect the attributes that the function manipulates. The arguments do not reflect input or output signals.

Argument-Naming Rules

- Input argument names must match the entity's attribute names.
 - Output argument names must be `out_` followed by the names of attributes to assign to the departing entity.
-

The entity must possess any attributes named as input arguments of the function. If the entity does not possess an attribute named using an output argument of the function, then the block creates the attribute.

The default function definition, below, indicates that the function called `fcn` uses the value of the attribute called `Attribute1` to compute values of the attributes called `Attribute1` and `Attribute2`.

```
function [out_Attribute1, out_Attribute2] = fcn(Attribute1)
```

- 3 Write the function to implement your specific computation. The value of each attribute can be a real- or complex-valued array of any dimension and double data type, but cannot be a structure or frame. For each attribute, the dimensions and complexity must be consistent throughout the model. Also, the function must use the Embedded MATLAB[®] subset. For more information, see “Overview of the Embedded MATLAB Subset” in the Embedded MATLAB documentation.

Note If you try to update the diagram or run the simulation for a model that contains the Attribute Function block, then you must have write access to the current folder because the application creates files there.

Example: Incorporating Legacy Code

Suppose you already have a file that defines a function that:

- Represents your desired attribute manipulation
- Uses the Embedded MATLAB subset

The function might or might not satisfy the argument-naming rules for the Attribute Function block. This example illustrates a technique that can help you satisfy the naming rule while preserving your legacy code.

- 1 The top level of the function associated with the Attribute Function block must satisfy the argument-naming rule. In this example, the function reads the entity's `x` attribute, so the function's input argument must be called `x`. The function assigns a new value to the entity's `x` attribute, so the function's output argument must be called `out_x`.

```
function out_x = update_x_attribute(x)
```

- 2 Using the variable names `x` and `out_x`, invoke your legacy code. The usage below assumes that you have called the function `mylegacyfcn`.

```
out_x = mylegacyfcn(x);
```

- 3 Include your legacy code as either a subfunction or an extrinsic function; see “Calling Functions” in the Embedded MATLAB documentation for details.

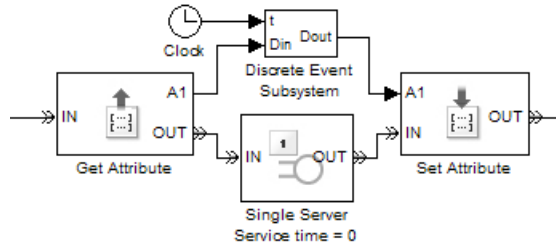
In your legacy code, the function name and the number of input and output arguments must be compatible with the invocation above. However, the legacy code does not need to follow the argument-naming rules for the Attribute Function block. Below is an example of a function that is compatible with the invocation above.

```
function outp = mylegacyfcn(inp)

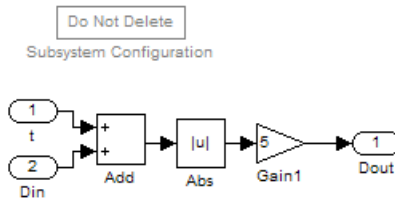
if inp < 0
    outp = 0;
else
    if ((inp >= 0) && (inp < 1))
        outp = 0.25;
    else
        outp = 5;
    end
end
```

Using Block Diagrams to Manipulate Attributes

To manipulate attributes using blocks to perform the computation, use an arrangement as in the following figure.



Top-Level Model



Subsystem Contents

In your own model, you can vary:

- The number of outputs of the Get Attribute block
- The number of inputs of the Set Attribute block
- The connections or contents of the Discrete Event Subsystem block

The key components are in the table.

Block	Role
Get Attribute	Queries the entity for its attribute value.
Discrete Event Subsystem	Ensures correct timing of the computation inside the subsystem, given that the output signal from the Get Attribute block is an event-based signal. Other signals, if any, in the computation can be event-based or time-based. To learn more about discrete event subsystems, see

Block	Role
	Chapter 10, “Controlling Timing with Subsystems”.
Content of the discrete event subsystem	Models your specific computation. The earlier figure shows one example: $5 Attr+t $, where <i>Attr</i> is an attribute value and <i>t</i> is a signal value unrelated to attributes. The configuration of the subsystem causes it to execute if and only if <i>Attr</i> has a sample time hit.
Single Server with Service time set to 0	Ensures that the Set Attribute block uses the up-to-date results of the computation. For details, see “Interleaving of Block Operations” on page 16-25.
Set Attribute	Assigns new value to the attribute.

Accessing Attributes of Entities

The table describes some ways you can access and use data that you have attached to an entity.

Use Attribute Values To...	Technique
Create a signal	<p>Use the Get Attribute block.</p> <p>For example, see the subsystem of the model described in “Adding Event-Based Behavior” in the SimEvents getting started documentation.</p>
Create a plot	<p>Use the Attribute Scope block and name the attribute in the Y attribute name parameter of the block.</p> <p>Alternatively, use the X-Y Attribute Scope block and name two attributes in the X attribute name and Y attribute name parameters of the block.</p> <p>For example, see the reference page for the X-Y Attribute Scope block.</p>
Compute a different attribute value	<p>Use the Attribute Function block.</p> <p>For example, see “Attribute Names Unique and Accessible in Composite Entity” on page 1-29.</p>
Help specify behavior of a block that supports the use of attribute values for block parameters. Examples are the service time for a server and the selected port for an output switch.	<p>Name the attribute in the block dialog box as applicable.</p> <p>For example, see “Example: Using an Attribute to Select an Output Port” in the SimEvents getting started documentation.</p>

Tip Suppose your entity possesses an attribute containing one of these quantities:

- Service time
- Switching criterion
- Another quantity that a block can obtain from either an attribute or signal

Use the attribute directly. This is better than creating a signal with the attribute value and ensuring that the signal is up-to-date when the entity arrives. For a comparison of the two approaches, see “Example: Using a Signal or an Attribute” on page 14-78.

Counting Entities

In this section...
“Counting Departures Across the Simulation” on page 1-20
“Counting Departures per Time Instant” on page 1-20
“Resetting a Counter Upon an Event” on page 1-22
“Associating Each Entity with Its Index” on page 1-24

Counting Departures Across the Simulation

Use the **#d** or **#a** output signal from a block to learn how many entities have departed from (or arrived at) a particular block. The output signal also indicates when departures occurred. This method of counting is cumulative throughout the simulation. These examples use the **#d** output signal to count departures:

- “Building a Simple Discrete-Event Model” in the SimEvents getting started documentation
- “Example: First Entity as a Special Case” on page 8-11
- “Stopping Upon Processing a Fixed Number of Entities” on page 12-36

Counting Departures per Time Instant

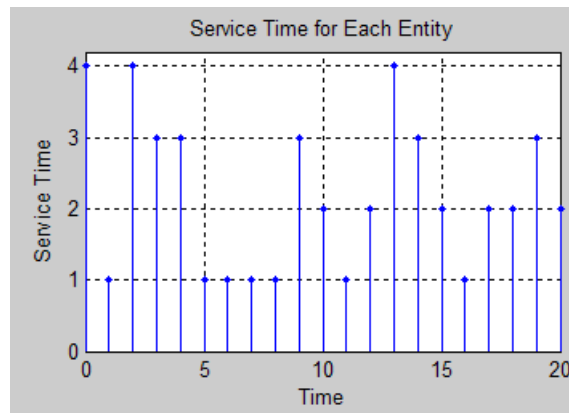
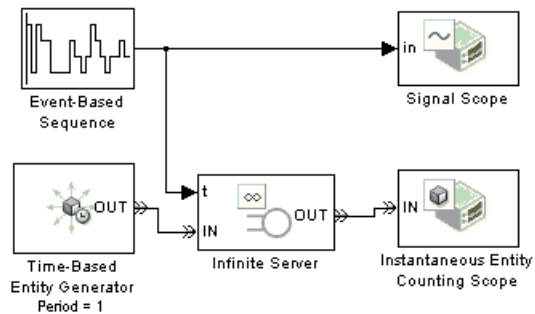
Suppose you want to visualize entity departures from a particular block, and you want to reset (that is, restart) the counter at each time instant. Visualizing departures per time instant can help you:

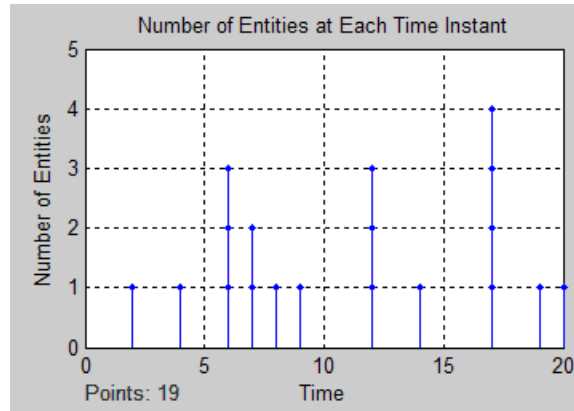
- Detect simultaneous departures
- Compare the number of simultaneous departures at different time instants
- Visualize the departure times while keeping the plot axis manageable

Use the Instantaneous Entity Counting Scope to plot the number of entities that have arrived at each time instant. The block restarts the count from 1 when the time changes. As a result, the count is cumulative for a given time instant, but not cumulative across the entire simulation.

Example: Counting Simultaneous Departures from a Server

In this example, the Infinite Server block sometimes completes service on multiple entities simultaneously. The Instantaneous Entity Counting Scope indicates how many entities departed from the server at each fixed time instant during the simulation.





Resetting a Counter Upon an Event

Suppose you want to count entities that depart from a particular block, and you want to reset the counter at arbitrary times during the simulation. Resetting the counter can help you compute statistics for evaluating your system over portions of the simulation.

The Entity Departure Counter block counts entity departures via a resettable counter. To configure the block:

- 1 Decide which type of events you want to reset the counter. The choices are:
 - Sample time hits
 - Trigger edges
 - Value changes
 - Function calls

These resources can help you choose which type of events:

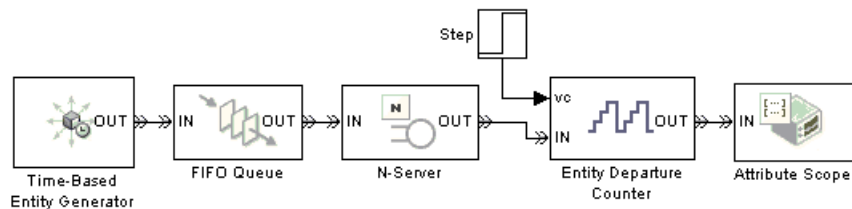
- To learn the difference among the kinds of signal-based events, see “Signal-Based Events” on page 2-4 and “Function Calls” on page 2-8.
- To build a signal that has events at the times that you need them, see “Manipulating Events” on page 2-32.
- To visualize the events of a signal, use the Instantaneous Event Counting Scope block.

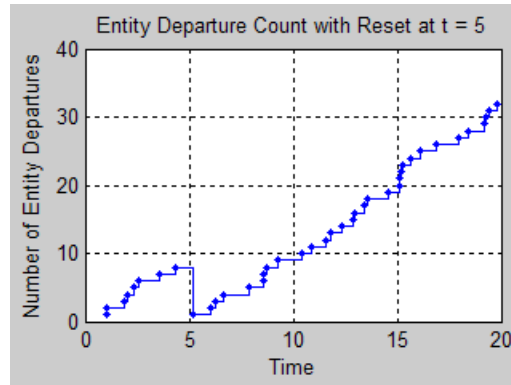
- 2 In the Entity Departure Counter block, indicate which signal-based events cause the counter to reset:
 - To reset upon sample time hits of an input signal, set **Reset counter upon** to Sample time hit from port ts.
 - To reset based on a trigger edge in an input signal, set **Reset counter upon** to Trigger from port tr. Then set **Trigger type** to Rising, Falling, or Either.
 - To reset based on a change in the value of an input signal, set **Reset counter upon** to Change in signal from port vc. Then set **Type of change in signal value** to Rising, Falling, or Either.
 - To reset upon function calls, set **Reset counter upon** to Function call from port fcn.
- 3 If you want to specify an explicit priority for the reset event, select **Resolve simultaneous signal updates according to event priority**. Then enter the priority using the **Event priority** parameter.
- 4 Click **OK** or **Apply**. The block now has a signal input port.
- 5 Connect a signal to the signal input port.

During the simulation, the block counts departing entities and resets its counter whenever the input signal satisfies your specified event criteria.

Example: Resetting a Counter After a Transient Period

This example counts entity departures from a queuing system, but resets the counter after an initial transient period.





Associating Each Entity with Its Index

Use the Entity Departure Counter block with **Write count to attribute** set to **On** to associate entity counts with the entities that use a particular path. The Nth entity departing from the Entity Departure Counter block has an attribute value of N.

For an example, see “Example: Setting Attributes” on page 1-10.

For an example in which the Entity Departure Counter block is more straightforward than storing the **#d** output signal in an attribute, see “Example: Sequence of Departures and Statistical Updates” on page 16-26.

Combining Entities

In this section...
“Overview of the Entity-Combining Operation” on page 1-25
“Example: Waiting to Combine Entities” on page 1-25
“Example: Copying Timers When Combining Entities” on page 1-27
“Example: Managing Data in Composite Entities” on page 1-28

Overview of the Entity-Combining Operation

You can combine entities from different paths using the Entity Combiner block. The entities you combine, called component entities, might represent different parts within a larger item, such as a header, payload, and trailer that are parts of a packet.

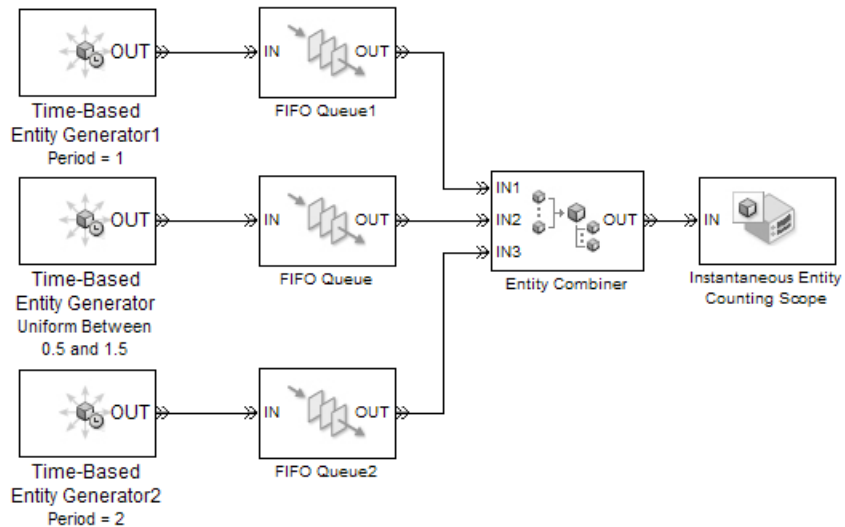
The Entity Combiner block and its surrounding blocks automatically detect when all necessary component entities are present and the result of the combining operation would be able to advance to a storage block or a block that destroys entities. “Example: Waiting to Combine Entities” on page 1-25 illustrates these automatic interactions among blocks.

The Entity Combiner block provides options for managing information (attributes and timers) associated with the component entities, as illustrated in “Example: Managing Data in Composite Entities” on page 1-28. You can also configure the Entity Combiner block to make the combining operation reversible via the Entity Splitter block.

For details about individual blocks, see the reference pages for the Entity Combiner and Entity Splitter blocks.

Example: Waiting to Combine Entities

The model below illustrates the synchronization of entities’ advancement by the Entity Combiner block and its preceding blocks.



The combining operation proceeds when all of these conditions are simultaneously true:

- The top queue has a pending entity.
- The middle queue has a pending entity.
- The bottom queue has a pending entity.
- The entity input port of the Instantaneous Entity Counting Scope block is available, which is true throughout the simulation.

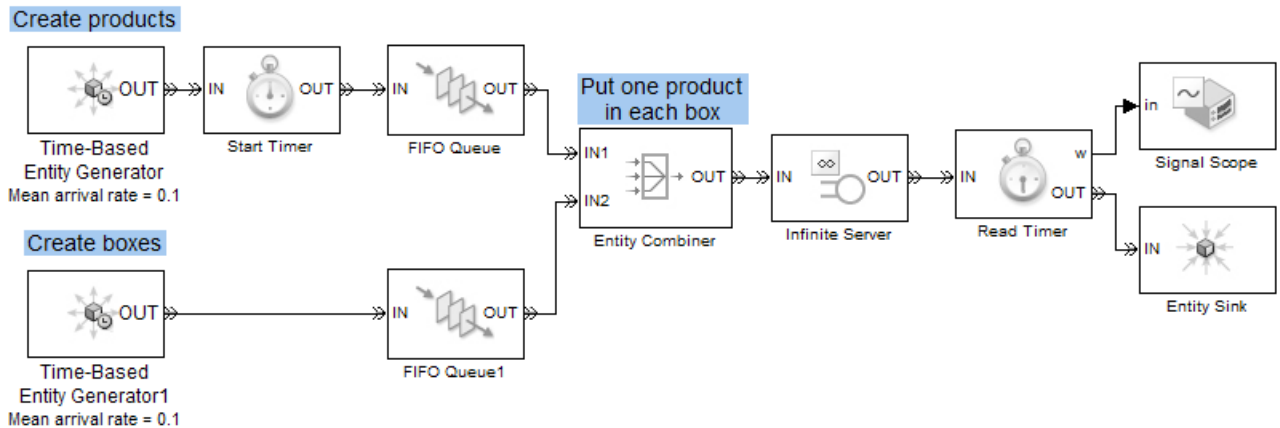
The bottom entity generator has the largest intergeneration time among the three entity generators, and is the limiting factor that determines when the Entity Combiner block accepts one entity from each queue. The top and middle queues store pending entities while waiting for the bottom entity generator to generate its next entity.

If you change the uniform distribution in the middle entity generator to produce intergeneration times between 0.5 and 3, then the bottom entity generator is not consistently the slowest. Nevertheless, the Entity Combiner block automatically permits the arrival of one entity from each queue as soon as each queue has a pending entity.

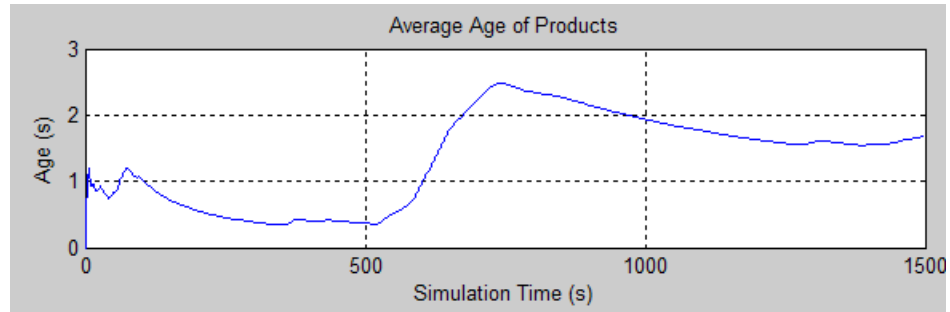
While you could alternatively synchronize the departures from the three queues using appropriately configured gates, it is simpler and more intuitive to use the Entity Combiner block as shown.

Example: Copying Timers When Combining Entities

The model below combines an entity representing a product with an entity representing a box, thus creating an entity that represents a boxed product. The Entity Combiner block copies the timer from the product to the boxed product.



The model plots the products' average age, which is the sum of the time that a product might wait for a box and the service time for boxed products in the Infinite Server block. In this simulation, some products wait for boxes, while some boxes wait for products. The generation of products and boxes are random processes with the same exponential distribution, but different seeds for the random number generator.

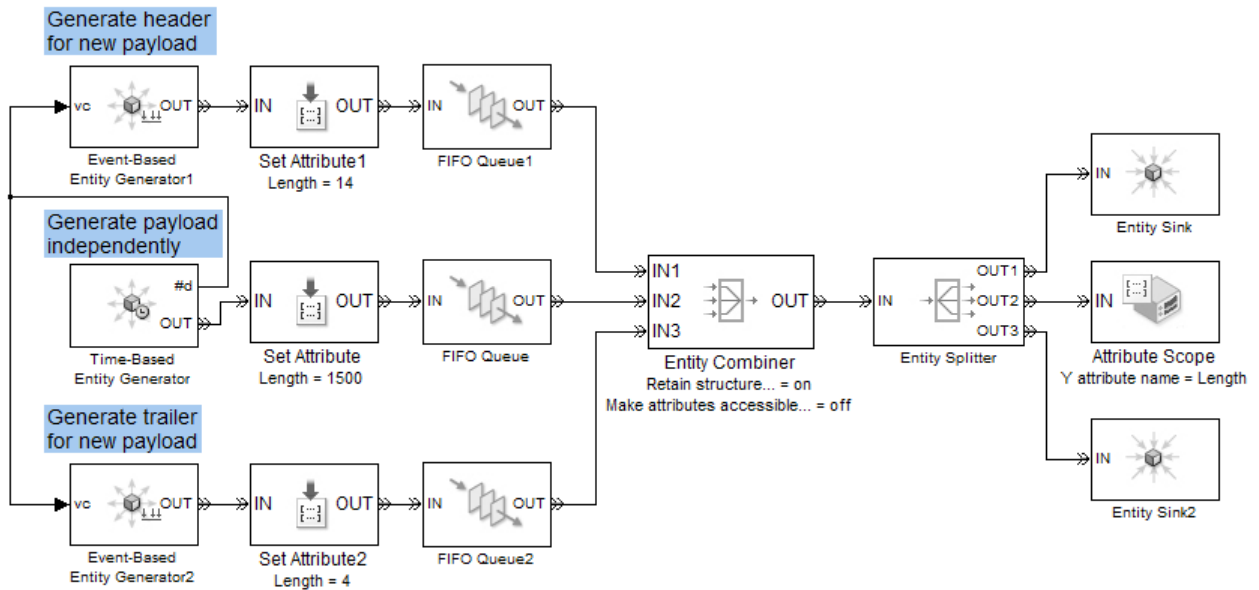


Example: Managing Data in Composite Entities

This section illustrates the connection between access to a component entity's attributes in a composite entity and uniqueness of the attribute names across all component entities.

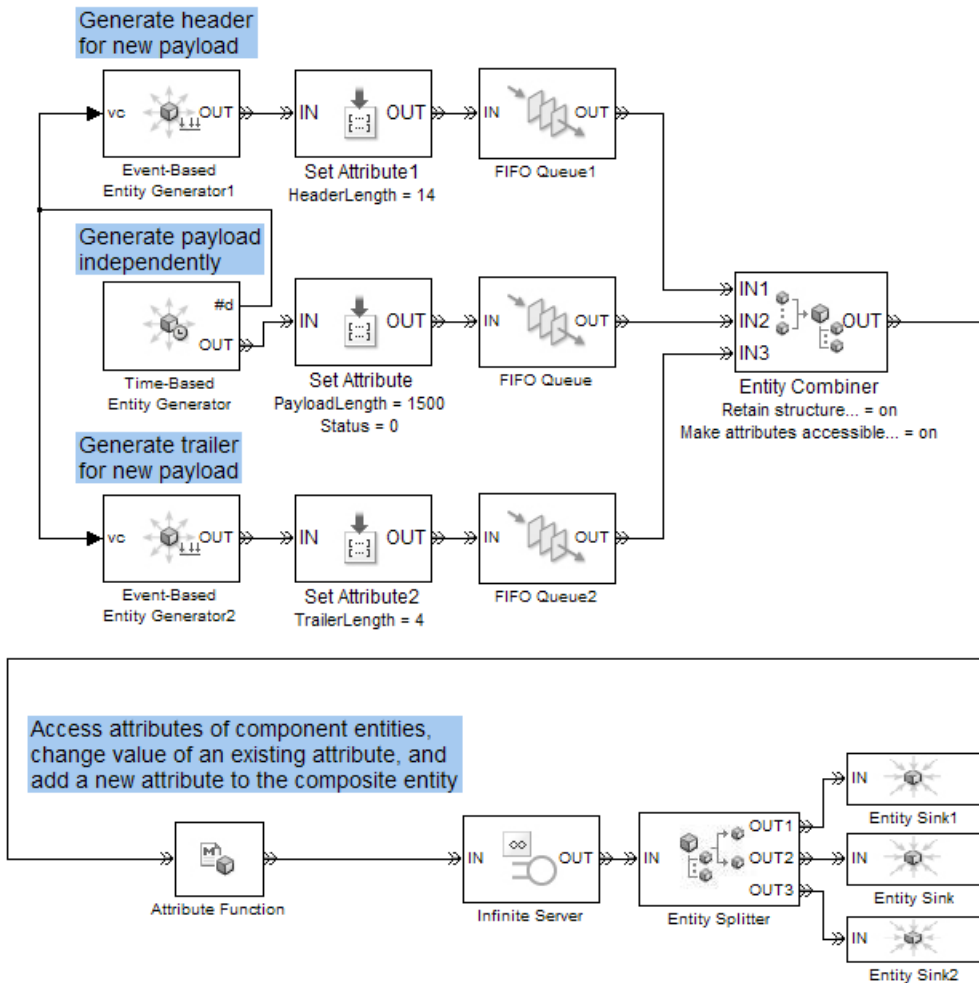
Attribute Names Nonunique and Inaccessible in Composite Entity

The model below combines component entities representing a header, payload, and trailer into a composite entity representing a packet. Each component entity has a `Length` attribute that the packet stores. When the Entity Splitter block divides the packet into separate header, payload, and trailer entities, each has the appropriate attribute. However, `Length` is not accessible in the packet (that is, after combining and before splitting). If it were, the name would be ambiguous because all component entities have an attribute by that name.



Attribute Names Unique and Accessible in Composite Entity

The model below uniquely names all attributes of the components and makes them accessible in the packet. If your primary focus is on data rather than the entities that carry the data, then you can think of the Entity Combiner block as aggregating data from different sources.



The model uses the Attribute Function block and its underlying function to illustrate these ways of accessing attributes via the composite entity.

Attribute Operation	Attribute Names	Use of Attribute Function Block
Read existing attributes of the component entities.	HeaderLength PayloadLength TrailerLength	Naming these attributes as input arguments of the function causes the block to read these attributes of the arriving entity.
Change the value of an existing attribute of a component. The new value persists when the Entity Splitter block divides the packet into its component entities.	Status	Naming out_Status as an output argument of the function causes the block to update the Status attribute of the entity.
Create a new attribute on the composite entity, not associated with any of the component entities. The new attribute does not persist beyond the Entity Splitter block.	PacketLength	Naming out_PacketLength as an output argument causes the block to create the PacketLength attribute on the entity.

Function Underlying the Attribute Function Block.

```
function [out_PacketLength, out_Status] = packet_length(HeaderLength,...
    PayloadLength,TrailerLength)
%PACKET_LENGTH Sum the component lengths and set Status to 1.

out_PacketLength = HeaderLength + PayloadLength + TrailerLength;
out_Status = 1;
```

Note The code does not distinguish between output arguments that define new attributes and output arguments that define new values for existing attributes. Only by examining other blocks in the model could you determine that `Status` is an existing attribute and `PacketLength` is not.

Replicating Entities on Multiple Paths

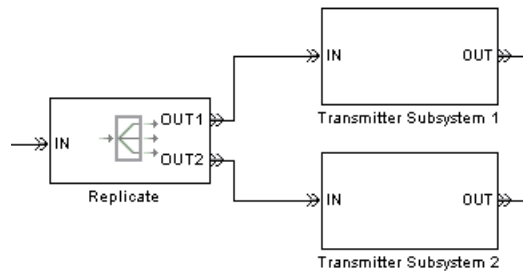
In this section...

“Sample Use Cases” on page 1-33

“Modeling Notes” on page 1-33

Sample Use Cases

The Replicate block enables you to distribute copies of an entity on multiple entity paths. Replicating entities might be a requirement of the situation you are modeling. For example, copies of messages in a multicasting communication system can advance to multiple transmitters or multiple recipients, as in the fragment below.



Similarly, copies of computer jobs can advance to multiple computers in a cluster so that the jobs can be processed in parallel on different platforms.

In some cases, replicating entities is a convenient modeling construct. For example, the MAC Controller subsystems in the Ethernet Local Area Network demo send one copy of an entity for processing and retain another copy of the same entity for the purpose of observing the state of the channel.

Modeling Notes

- Unlike the Output Switch block, the Replicate block has departures at all of its entity output ports that are not blocked, not just a single selected entity output port.

- If your model routes the replicates such that they use a common entity path, then be aware that blockages can occur during the replication process. For example, connecting all ports of a Replicate block, Path Combiner block, and Single Server block in that sequence can create a blockage because the server can accommodate at most one of the replicates at a time. The blockage causes fewer than the maximum number of replicates to depart from the block.
- Each time the Replicate block replicates an entity, the copies depart in a sequence whose start is determined by the **Departure port precedence** parameter. Although all copies depart at the same time instant, the sequence might be significant in some modeling situations. For details, see the reference page for the Replicate block.

Working with Events

- “Supported Events in SimEvents Models” on page 2-2
- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-10
- “Observing Events” on page 2-18
- “Generating Function-Call Events” on page 2-28
- “Manipulating Events” on page 2-32

To learn about working with sets of simultaneous events, see Chapter 3, “Managing Simultaneous Events”. To view information about events during the simulation, see Chapter 14, “Debugging Discrete-Event Simulations”.

Supported Events in SimEvents Models

In this section...
“Types of Supported Events” on page 2-2
“Signal-Based Events” on page 2-4
“Function Calls” on page 2-8

Types of Supported Events

An event is an instantaneous discrete incident that changes a state variable, an output, and/or the occurrence of other events. SimEvents software supports the events listed below.

Event	Description
Counter reset	Reinitialization of the counter in the Entity Departure Counter block.
Delayed restart	Causes a pending entity in the Time-Based Entity Generator block to attempt to depart. The block uses delayed restart events only when you set Response when unblocked to Delayed restart.
Entity advancement	Departure of an entity from one block and arrival at another block.
Entity destruction	Arrival of an entity at a block that has no entity output port.
Entity generation	Creation of an entity, except in the case of an Event-Based Entity Generator block that has suspended the generation of entities.
Entity request	Notification that an entity input port has become available. A preceding block’s response to the notification might result in an entity advancement. After each entity advancement, an Enabled Gate block or a switch block reissues the notification until no further entity advancement can occur.

Event	Description
Function call	Discrete invocation request carried from block to block by a special signal called a function-call signal. For more information, see “Function Calls” on page 2-8.
Gate (opening or closing)	Opening or closing of the gate represented by the Enabled Gate block.
Memory read	Reading of memory in the Signal Latch block.
Memory write	Writing of memory in the Signal Latch block.
Port selection	Selection of a particular entity port in the Input Switch, Output Switch, or Path Combiner block. In the case of the Path Combiner block, the selected entity input port is the port that the block notifies first, whenever its entity output port changes from blocked to unblocked.
Preemption	Replacement of an entity in a server by a higher priority entity.
Release	Opening of the gate represented by the Release Gate block.
Sample time hit	Update in the value of a signal that is connected to a block configured to react to signal updates.
Service completion	Completion of service on an entity in a server.
Storage completion	Change in the state of the Output Switch block, making it attempt to advance the stored entity.
Subsystem	Execution of Discrete Event Subsystem block contents caused by an appropriate signal-based event at one of the subsystem’s inports.
Timeout	Departure of an entity that has exceeded a previously established time limit.

Event	Description
Trigger	Rising or falling edge of a signal connected to a block that is configured to react to relevant trigger edges. A rising edge is an increase from a negative or zero value to a positive value (or zero if the initial value is negative). A falling edge is a decrease from a positive or a zero value to a negative value (or zero if the initial value is positive).
Value change	Change in the value of a signal connected to a block that is configured to react to relevant value changes.

During a simulation, the application maintains a list, called the *event calendar*, of selected upcoming events that are scheduled for the current simulation time or future times. By referring to the event calendar, the application executes events at the correct simulation time and in an appropriate sequence.

Signal-Based Events

Sample time hits, value changes, and triggers are collectively called *signal-based events*. Signal-based events can occur with respect to time-based or event-based signals. Signal-based events provide a mechanism for a block to respond to selected state changes in a signal connected to the block. The kind of state change to which the block responds determines the specific type of signal-based event.

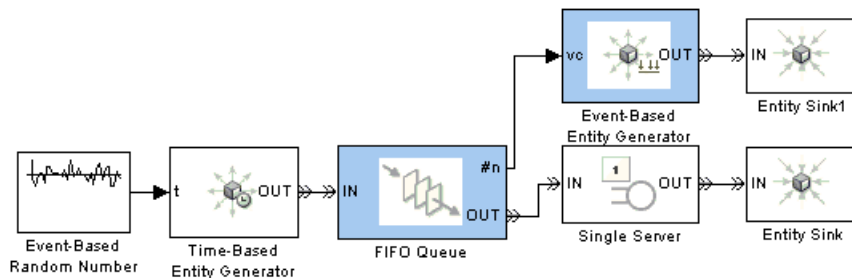
When comparing the types of signal-based events, note that

- The updated value that results in a sample time hit could be the same as or different from the previous value of the signal.
- Event-based signals do not necessarily undergo an update at the beginning of the simulation.
- Every change in a signal value is also an update in that signal's value. However, the opposite is not true because an update that merely reconfirms the same value is not a change in the value.

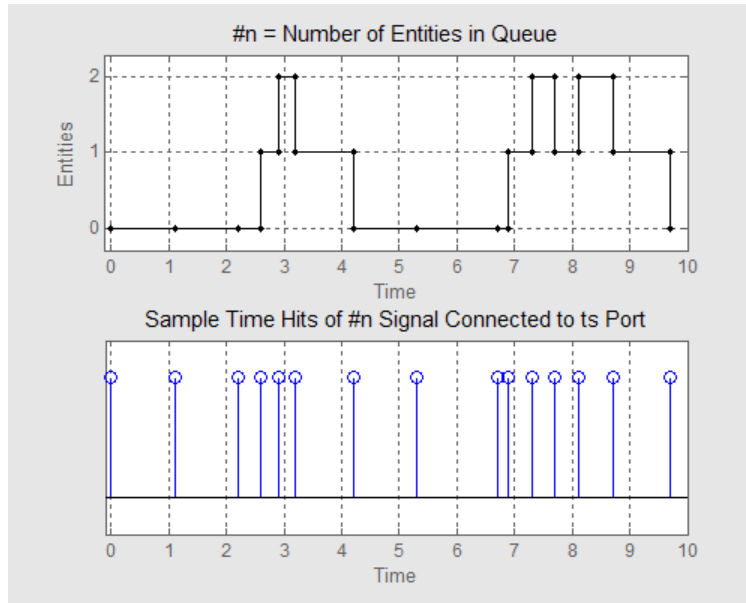
- Every rising or falling edge is also a change in the value of the signal. However, the opposite is not true because a change from one positive value to another (or from one negative value to another) is not a rising or falling edge.
- Triggers and value changes can be rising or falling. You configure a block to determine whether the block considers rising, falling, or either type to be a relevant occurrence.
- Blocks in the Simulink® libraries are more restrictive than blocks in the SimEvents libraries regarding trigger edges that rise or fall from zero. Simulink blocks in discrete-time systems do not consider a change from zero to be a trigger edge unless the signal remained at zero for more than one time step; see “Triggered Subsystems” in the Simulink documentation. SimEvents blocks configured with **tr** ports consider any change from zero to a nonzero number to be a trigger edge.

Example: Comparing Types of Signal-Based Events

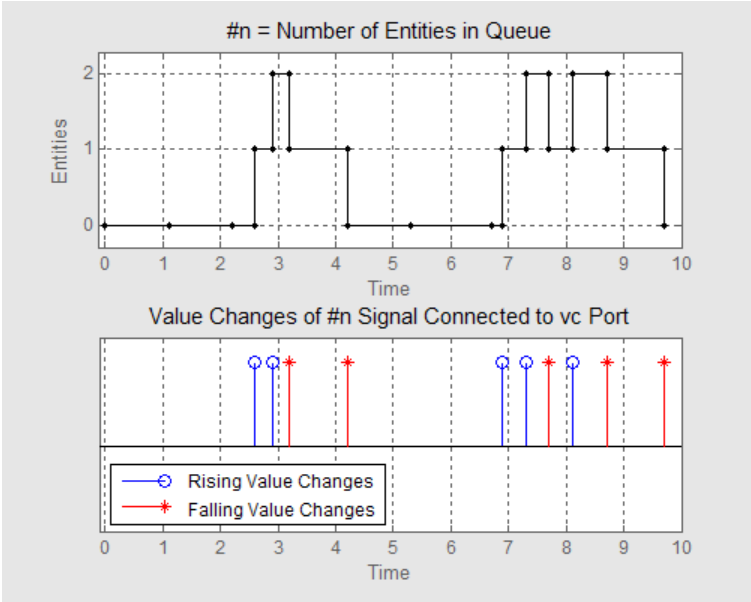
Consider the signal representing the number of entities stored in a FIFO queue. This is the **#n** output signal from the FIFO Queue block in the model below. The **#n** signal is connected to the Event-Based Entity Generator block, which reacts to different types of signal-based events. Parameters in its dialog box determine whether the block has a **ts**, **vc**, or **tr** input port, as well as the types of events to which the block reacts.



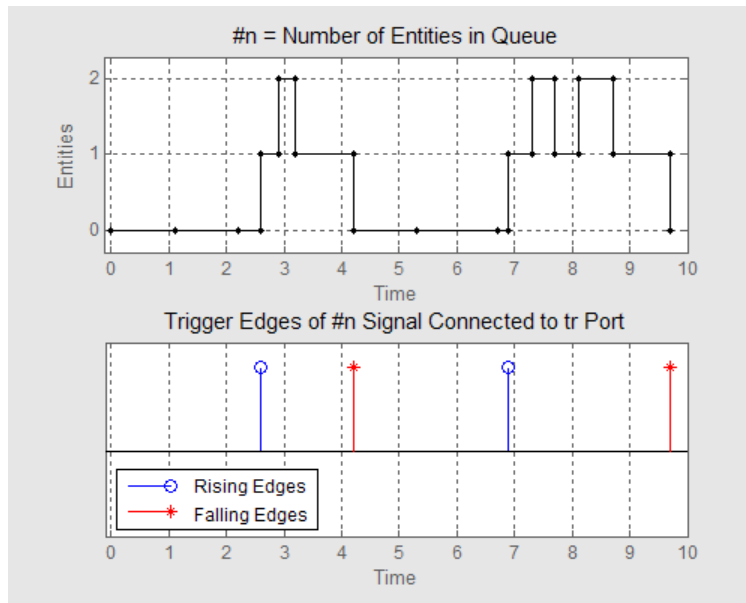
In the following figures, a staircase plot shows the values of the **#n** signal, starting from the first entity’s arrival at $T=0$. Based on those values of **#n**, the stem plots indicate when signal-based events occur at the Event-Based Entity Generator block if it is configured with a **ts**, **vc**, or **tr** signal input port.



Sample Time Hits of #n Signal Connected to ts Port



Value Changes of #n Signal Connected to vc Port



Trigger Edges of #n Signal Connected to tr Port

Function Calls

Function calls are discrete invocation requests carried from block to block by a special signal called a function-call signal. A function-call signal appears as a dashed line, not a solid line. A function-call signal carries a function call at discrete times during the simulation and does not have a defined value at other times. A function-call signal is capable of carrying multiple function calls at the same value of the simulation clock, representing multiple simultaneous events.

In SimEvents models, function calls are the best way to make Stateflow blocks and blocks in the Simulink libraries respond to asynchronous state changes.

Function calls can come from these kinds of blocks:

- Event generator blocks or Function-Call Generator blocks
- Event translation blocks
- Stateflow blocks

- Embedded MATLAB Function blocks

You can combine or manipulate function-call signals. To learn more, see “Manipulating Events” on page 2-32.

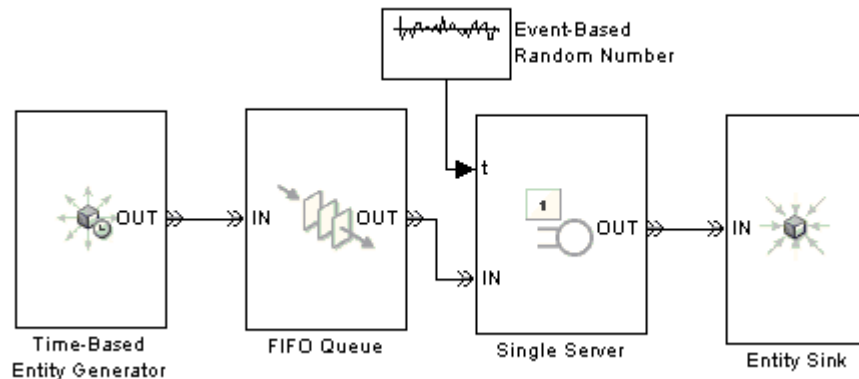
Example: Event Calendar Usage for a Queue-Server Model

In this section...

“Overview of Example” on page 2-10
 “Start of Simulation” on page 2-11
 “Generation of First Entity” on page 2-11
 “Generation of Second Entity” on page 2-12
 “Completion of Service Time” on page 2-13
 “Generation of Third Entity” on page 2-14
 “Generation of Fourth Entity” on page 2-15
 “Completion of Service Time” on page 2-16

Overview of Example

To see how the event calendar drives the simulation of a simple event-based model, consider the queue-server model depicted below.



Assume that the blocks are configured so that:

- The Time-Based Entity Generator block generates an entity at $T = 0.9, 1.7, 3.8, 3.9,$ and 6 .

- The queue has infinite capacity.
- The server uses service times of 1.3, 2.0, and 0.7 seconds for the first three entities.

The sections below indicate how the event calendar and the system's states change during the simulation.

Start of Simulation

When the simulation starts, the queue and server are empty. The entity generator schedules an event for $T = 0.9$. The event calendar looks like the table below.

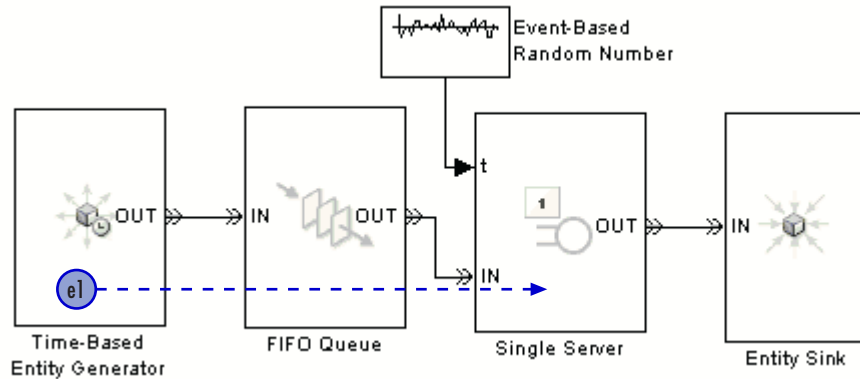
Time of Event (s)	Type of Event
0.9	Time-Based Entity Generator block generates an entity.

Generation of First Entity

At $T = 0.9$,

- The entity generator generates an entity and attempts to output it.
- The queue is empty, so the entity advances from the entity generator to the queue.
- The newly generated entity is the only one in the queue, so the queue attempts to output the entity. It queries the server to determine whether the server can accept the entity.
- The server is empty, so the entity advances from the queue to the server.
- The server's entity input port becomes temporarily unavailable to future entities.
- The server schedules an event that indicates when the entity's service time is completed. The service time is 1.3 seconds, so service is complete at $T = 2.2$.
- The entity generator schedules its next entity-generation event, at $T = 1.7$.

In the schematic below, the circled notation “e1” depicts the first entity and the dashed arrow is meant to indicate that this entity advances from the entity generator through the queue to the server.



The event calendar looks like this.

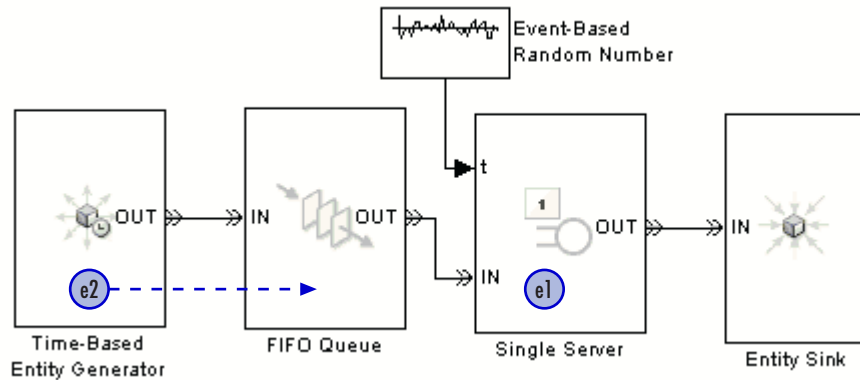
Time of Event (s)	Event Description
1.7	Time-Based Entity Generator block generates second entity.
2.2	Single Server block completes service on the first entity.

Generation of Second Entity

At $T = 1.7$,

- The entity generator generates an entity and attempts to output it.
- The queue is empty, so the entity advances from the entity generator to the queue.
- The newly generated entity is the only one in the queue, so the queue attempts to output the entity. However, the server’s entity input port is unavailable, so the entity stays in the queue. The queue’s entity output port is said to be blocked because an entity has tried and failed to depart via this port.

- The entity generator schedules its next entity-generation event, at $T = 3.8$.



Time of Event (s)	Event Description
2.2	Single Server block completes service on the first entity.
3.8	Time-Based Entity Generator block generates the third entity.

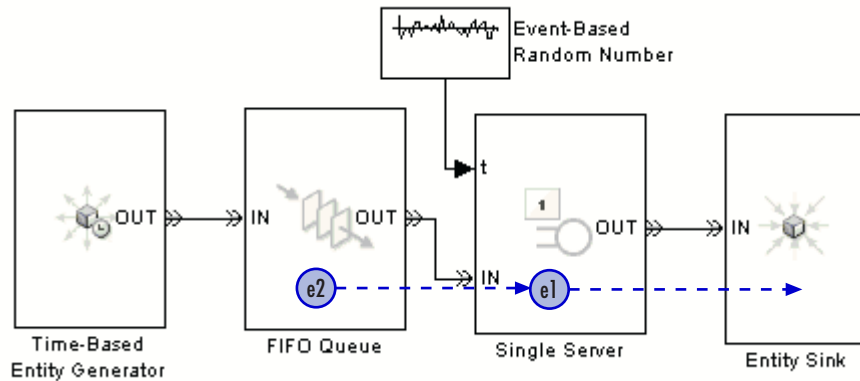
Completion of Service Time

At $T = 2.2$,

- The server finishes serving its entity and attempts to output it. The server queries the next block to determine whether it can accept the entity.
- The sink block accepts all entities by definition, so the entity advances from the server to the sink.
- The server's entity input port becomes available. The server executes an event to notify the queue about the newly available entity input port. This is called an entity request event.
- The queue is now able to output the second entity to the server. As a result, the queue becomes empty and the server becomes busy again.
- The server's entity input port becomes temporarily unavailable to future entities.

- The server schedules an event that indicates when the second entity's service time is completed. The service time is 2.0 seconds.

Note The server's entity input port started this time instant in the unavailable state, became available (when the first entity departed from the server), and then became unavailable once again (when the second entity arrived at the server). It is not uncommon for a state to change more than once in the same time instant.



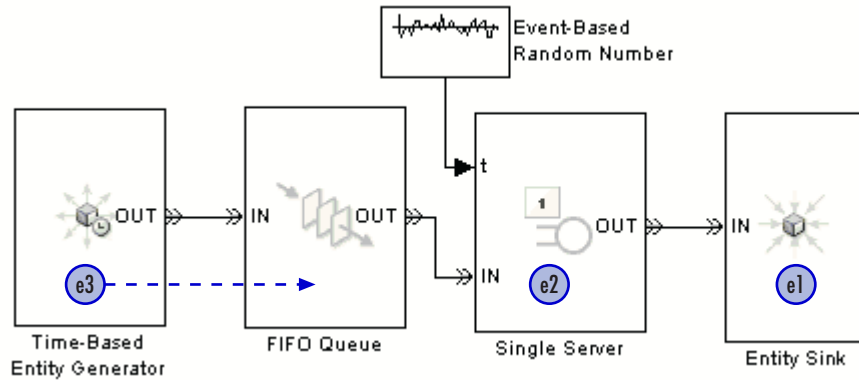
Time of Event (s)	Event Description
3.8	Time-Based Entity Generator block generates the third entity.
4.2	Single Server block completes service on the second entity.

Generation of Third Entity

At $T = 3.8$,

- The entity generator generates an entity and attempts to output it.
- The queue is empty, so the entity advances from the entity generator to the queue.

- The newly generated entity is the only one in the queue, so the queue attempts to output the entity. However, the server's entity input port is unavailable, so the entity stays in the queue.
- The entity generator schedules its next entity-generation event, at $T = 3.9$.

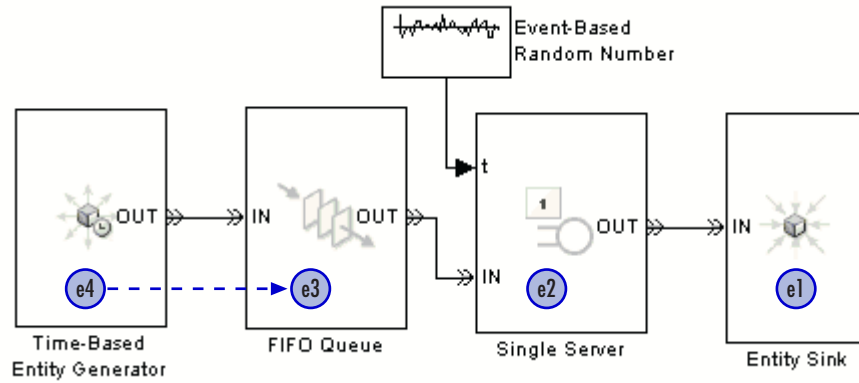


Time of Event (s)	Event Description
3.9	Time-Based Entity Generator block generates the fourth entity.
4.2	Single Server block completes service on the second entity.

Generation of Fourth Entity

At $T = 3.9$,

- The entity generator generates an entity and attempts to output it.
- The queue is not full, so the entity advances from the entity generator to the queue.
- The server's entity input port is still unavailable, so the queue cannot output an entity. The queue length is currently two.
- The entity generator schedules its next entity-generation event, at $T = 6$.



Time of Event (s)	Event Description
4.2	Single Server block completes service on the second entity.
6	Time-Based Entity Generator block generates the fifth entity.

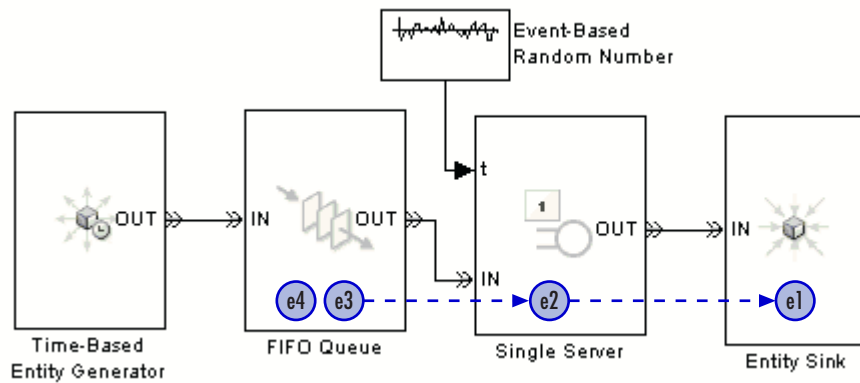
Completion of Service Time

At $T = 4.2$,

- The server finishes serving its entity and attempts to output it.
- The sink block accepts all entities by definition, so the entity advances from the server to the sink.
- The server's entity input port becomes available, so it executes an entity request event. The queue's entity output port becomes unblocked. The queue is now able to output the third entity to the server. As a result, the queue length becomes one, and the server becomes busy.
- The server's entity input port becomes temporarily unavailable to future entities.
- The server schedules an event that indicates when the entity's service time is completed. The service time is 0.7 seconds.

- The queue attempts to output the fourth entity. However, the server's entity input port is unavailable, so this entity stays in the queue. The queue's entity output port becomes blocked.

Note The queue's entity output port started this time instant in the blocked state, became unblocked (when it sensed that the server's entity input port became available), and then became blocked once again (when the server began serving the third entity).



Time of Event (s)	Event Description
4.9	Single Server block completes service on the third entity.
6	Time-Based Entity Generator block generates the fifth entity

Observing Events

In this section...
“Techniques for Observing Events” on page 2-18
“Example: Observing Service Completions” on page 2-22
“Example: Detecting Collisions by Comparing Events” on page 2-25

Techniques for Observing Events

The SimEvents debugger can help you observe events and the relative sequencing of simultaneous events. For details, see “Overview of the SimEvents Debugger” on page 14-3.

The next table describes some additional observation techniques. Each technique focuses on a particular kind of event. These techniques indicate the simulation time at which events occur but do not indicate relative sequencing of simultaneous events. Key tools are the Instantaneous Event Counting Scope block, Signal Scope block, and Discrete Event Signal to Workspace block. You can also build a discrete event subsystem that counts events and creates a signal, as illustrated in “Example: Focusing on Events, Not Values” on page 10-23.

Event	Observation Technique
Counter reset	In the counter block #d output signal, observe falling edges. Alternatively, use a branch line to connect the input signal to an Instantaneous Event Counting Scope block.
Delayed restart	
Entity advancement	In the block from which the entity departs, observe increases in the #d output signal.
Entity destruction	In the Entity Sink block, observe increases in the #a output signal. The Instantaneous Entity Counting Scope block provides a plot in place of a #a signal.

Event	Observation Technique
Entity generation	<p>In the entity generator block, observe values of the pe and #d output signals. Upon entity generation, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The generated entity departs immediately. #d increases and pe does not change. • The generated entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique described in “Example: Observing Service Completions” on page 2-22.</p>
Entity request	
Function call	<p>If the block issuing the function call provides a #f1 output signal, observe its increases. Otherwise, configure a Signal-Based Function-Call Event Generator block by enabling the #f1 output port and setting Generate function call only upon to Function call from port fcn. Insert the Signal-Based Function-Call Event Generator block between the block issuing the function call and the block reacting to the function call.</p>
Gate (opening or closing)	<p>In the Enabled Gate block, use a branch line to connect the en input signal to an Instantaneous Event Counting Scope block. Rising trigger edges of the input signal indicate gate-opening events, while falling trigger edges of the input signal indicate gate-closing events.</p>
Memory read	<p>In the Signal Latch block, observe sample time hits in the out output signal.</p>
Memory write	<p>In the Signal Latch block, observe sample time hits in the mem output signal..</p>
Port selection	<p>If the block has a p input signal, use a branch line to connect the p signal to an Instantaneous Event Counting Scope block that is configured to plot value changes. For the Input Switch or Output Switch block, an alternative is to observe the last output signal.</p>

Event	Observation Technique
Preemption	In the server block, observe increases in the #p output signal.
Release	In the Release Gate block, use a branch line to connect the input signal to an Instantaneous Event Counting Scope block.
Sample time hit	Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.
Service completion	<p>For Single Server blocks, observe values of the pe and #d output signals. Upon service completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and pe does not change. • The entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique described in “Example: Observing Service Completions” on page 2-22.</p>
	<p>For Infinite Server and N-Server blocks, observe values of the #pe and #d output signals. Upon service completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and #pe does not change. • The entity cannot depart immediately. #pe increases. <p>For a concrete example, see “Example: Observing Service Completions” on page 2-22.</p>

Event	Observation Technique
Storage completion	<p>In the switch block, observe values of the pe and #d output signals. Upon storage completion, you see one of the following outcomes:</p> <ul style="list-style-type: none"> • The entity departs immediately. #d increases, and pe does not change. • The entity cannot depart immediately. pe increases. <p>To build a concrete example, adapt the technique in “Example: Observing Service Completions” on page 2-22.</p>
Subsystem	<p>In any output signal from the subsystem, observe sample time hits. Alternatively, connect a Discrete Event Signal to Workspace block to any signal inside the subsystem. Then observe the times at which the variable in the workspace indicates a sample time hit of the signal.</p>
Timeout	<p>In the storage block from which the entity times out, observe increases in the #to output signal.</p>
Trigger	<p>Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.</p>
Value change	<p>Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.</p>

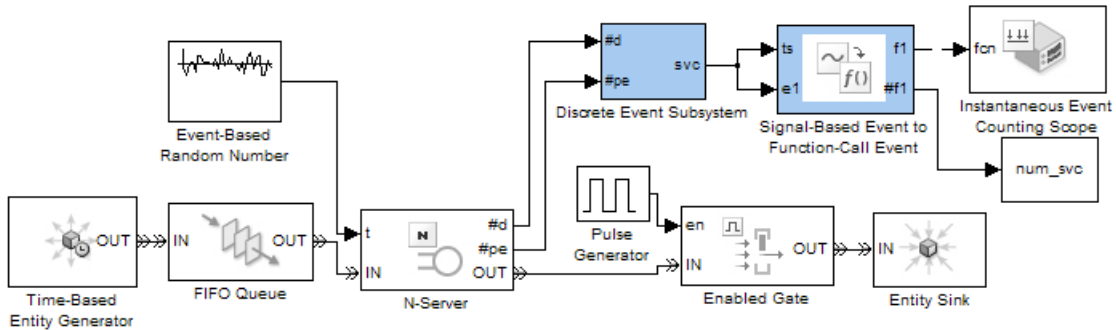
For examples that use one or more of these techniques, see:

- “Example: Plotting Event Counts to Check for Simultaneity” on page 11-15
- “Example: Observing Service Completions” on page 2-22
- “Example: Focusing on Events, Not Values” on page 10-23

Also, “Example: Detecting Collisions by Comparing Events” on page 2-25 describes how to use a Signal Latch block to observe which of two types of events are more recent.

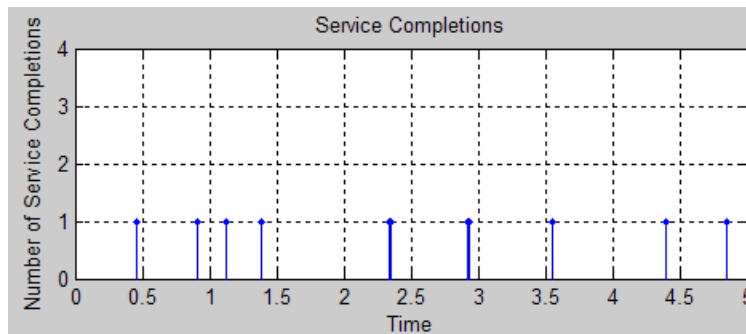
Example: Observing Service Completions

The following example creates a stem plot showing when an N-Server block completes service on each entity. The example also writes a signal, `num_svc`, to the MATLAB workspace that indicates when each service completion occurred.



Example Results

The example produces a plot in which each stem indicates a service completion. The timing depends on the entity generation, service completion, gate opening, and gate closing times in the model.



After the simulation is over, to form a vector of service completion times, enter the following code:

```
t_svcpc = num_svc.time
```

The output is:

```
t_svcp =  
0.4542  
0.9077  
1.1218  
1.3868  
2.3430  
2.3570  
2.9251  
2.9370  
3.5592  
4.3933  
4.8554
```

Computation Details

In the model, these blocks jointly determine when service completions occurred:

- Discrete Event Subsystem
- Signal-Based Event to Function-Call Event

As inputs, the subsystem uses the **#d** and **#pe** output signals from the server block.

Tip To adapt this technique to blocks that have a **pe** but not a **#pe** output signal, use **pe**.

The subsystem executes when either **#d** or **#pe** increases because a service completion has one of these consequences:

- The entity departs immediately. **#d** increases, while **#pe** does not change.
- The entity cannot depart immediately, so it becomes a pending entity. **#pe** increases, while **#d** does not change.

Inside the subsystem is the Embedded MATLAB Function block. It contains this code:

```
function svc = svc_completion(d, pe_sig)
    %#eml

    %SVC_COMPLETION Output 1 upon each service completion.
    % SVC = SVC_COMPLETION(D, PE_SIG) outputs 1 if output signals from a
    % server block indicate that a service completion occurred. The function
    % outputs 0 otherwise. D is the #d output signal from a server block.
    % PE_SIG is the #pe output signal from an N-Server or Infinite Server
    % block, or the pe output signal from a Single Server block.

    % Declare variables that must retain values between iterations.
    persistent last_d last_pe_sig;

    % Initialize persistent variables in the first iteration.
    if isempty(last_d)
        last_d = 0;
        last_pe_sig = 0;
    end

    % Compute the output. A service completion occurred if either is true:
    % * d increases and pe_sig remains the same.
    % * pe_sig increases.
    if ((d > last_d && isequal(pe_sig,last_pe_sig)) || (pe_sig > last_pe_sig))
        svc = 1;
    else
        svc = 0;
    end

    % Update the persistent variables for the next iteration.
    last_d = d;
    last_pe_sig = pe_sig;
```

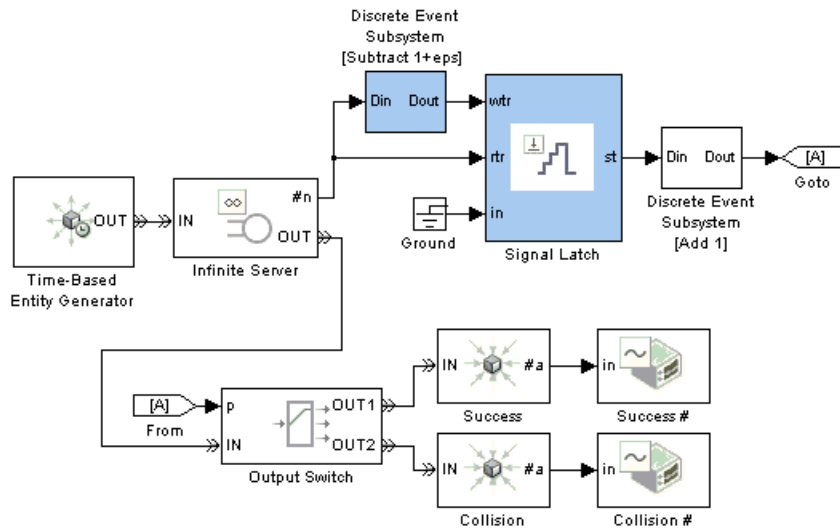
Note The subsystem does not execute upon decreases in **#pe**. A decrease in **#pe** means that a pending entity has departed. The departure causes a simultaneous increase in **#d**, but the block updates **#pe** before **#d**. Executing the subsystem upon decreases in **#pe** would be incompatible with the preceding code because **#pe** would reflect the departure but **#d** would not.

The output signal from the subsystem, **svc**, has a sample time hit of 1 whenever there is a service completion. The signal also has sample time hits of 0 when the subsystem executes but there is no service completion. To remove the sample time hits of 0, the example connects the **svc** signal to the **ts** and **e1** input ports of the Signal-Based Event to Function-Call Event block. Each time a service completion occurs, the block:

- Generates a function call at the **f1** output port
- Increases the value of the **#f1** output signal

Example: Detecting Collisions by Comparing Events

The example below aims to determine whether an entity is the only entity in an infinite server for the entire duration of service. The model uses the Signal Latch block to compare the times of two kinds of events and report which kind occurred more recently. This usage of the Signal Latch block relies on the block's status output signal, **st**, rather than the default **in** and **out** ports.



In the model, entities arrive at an infinite server, whose **#n** output signal indicates how many entities are in the server. The Signal Latch block responds to these signal-based events involving the integer-valued **#n** signal:

- If **#n** increases from 0 to a larger integer, then
 - **rtr** has a rising edge.
 - The Signal Latch block processes a read event.
 - The Signal Latch block's **st** output signal becomes 0.
- If **#n** increases from 1 to a larger integer, then
 - **wtr** has a rising edge.
 - The Signal Latch block processes a write event.
 - The Signal Latch block's **st** output signal becomes 1.
- If **#n** increases from 0 to 2 at the same value of the simulation clock, then it also assumes the value 1 as a zero-duration value. As a result,
 - **rtr** and **wtr** both have rising edges, in that sequence.
 - The Signal Latch block processes a read event followed by a write event.
 - The Signal Latch block's **st** output signal becomes 1.

By the time the entity departs from the Infinite Server block, the Signal Latch block's **st** signal is 0 if and only if that entity has been the only entity in the server block for the entire duration of service. This outcome is considered a success for that entity. Other outcomes are considered collisions between that entity and one or more other entities.

This example is similar to the CSMA/CD subsystem in the Ethernet Local Area Network demo.

Generating Function-Call Events

In this section...
“Role of Explicitly Generated Events” on page 2-28
“Generating Events When Other Events Occur” on page 2-28
“Generating Events Using Intergeneration Times” on page 2-30

Role of Explicitly Generated Events

You can generate an event and use it to

- Invoke a discrete event subsystem, Embedded MATLAB Function block, or Stateflow block
- Cause certain events, such as the opening of a gate or the reading of memory in a Signal Latch block
- Generate an entity

For most purposes, a function call is an appropriate type of event to generate.

Note While you can invoke triggered subsystems, Embedded MATLAB Function blocks, and Stateflow blocks upon trigger edges, trigger usage has limitations in discrete-event simulations. In particular, you should use function calls instead of triggers if you want the invocations to occur asynchronously, to be prioritized among other simultaneous events, or to occur more than once in a fixed time instant.

Generating Events When Other Events Occur

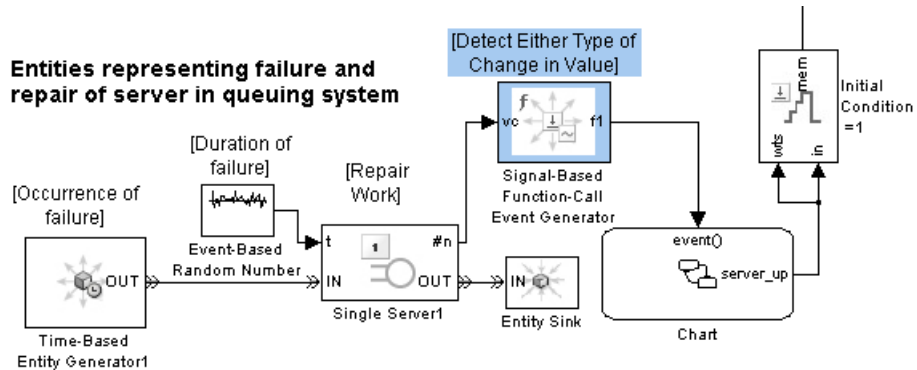
The table below indicates which blocks generate function calls when other events occur.

Event Upon Which to Generate Another Event	Block
Entity advancement	Entity-Based Function-Call Event Generator
Signal-based event	Signal-Based Function-Call Event Generator
Function call	Signal-Based Function-Call Event Generator

Example: Calling a Stateflow Block Upon Changes in Server Contents

The fragment below, which is part of an example in “Using Stateflow Charts to Implement a Failure State” on page 5-21, uses entities to represent failures and repairs of a server elsewhere in the model:

- A failure of the server is modeled as an entity’s arrival at the block labeled Repair Work. When the Repair Work block’s #n signal increases to reflect the entity arrival, the Signal-Based Function-Call Event Generator block generates a function call that calls the Stateflow block to change the state of the server from up to down.
- A completed repair of the server is modeled as an entity’s departure from the Repair Work block. When the Repair Work block’s #n signal decreases to reflect the entity departure, the Signal-Based Function-Call Event Generator block generates a function call that calls the Stateflow block to change the state of the server from down to up.



One reason to use function calls rather than triggers to call a Stateflow block in discrete-event simulations is that an event-based signal can experience a trigger edge due to a zero-duration value that a time-based block would not recognize. The Signal-Based Function-Call Event Generator can detect signal-based events that involve zero-duration values.

Generating Events Using Intergeneration Times

To generate events using intergeneration times from a signal or a statistical distribution, use this procedure:

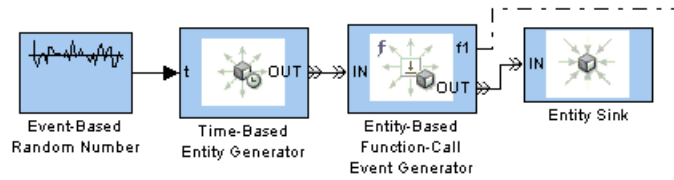
- 1** Use the signal or statistical distribution with the Time-Based Entity Generator block to generate entities.
- 2** Use the Entity-Based Function-Call Event Generator block to generate an event associated with each entity.
- 3** Terminate the entity path with an Entity Sink block.

In the special case when the intergeneration time is constant, a simpler alternative is to use the Function-Call Generator block in the Simulink Ports & Subsystems library.

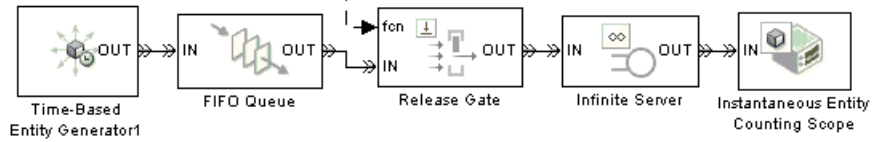
Example: Opening a Gate Upon Random Events

The example below uses the top entity generator to generate entities whose sole purpose is to cause the generation of events with intergeneration times from a statistical distribution. The bottom entity generator generates entities that enter a gated queuing system.

Generating Events with Random Intergeneration Times



Using Events to Open a Gate



Manipulating Events

In this section...
“Reasons to Manipulate Events” on page 2-32
“Blocks for Manipulating Events” on page 2-34
“Creating a Union of Multiple Events” on page 2-34
“Translating Events to Control the Processing Sequence” on page 2-37
“Conditionalizing Events” on page 2-39

Reasons to Manipulate Events

You can manipulate events to accomplish any of these goals:

- To invoke a function-call subsystem, Embedded MATLAB Function block, or Stateflow block upon entity departures or signal-based events.

Note You can invoke triggered subsystems, Embedded MATLAB Function blocks, and Stateflow blocks upon trigger edges, which are a type of signal-based event. However, you will need to translate the trigger edges into function calls if you want the invocations to occur asynchronously, to be prioritized among other simultaneous events, or to occur more than once in a fixed time instant.

- To create a union of events from multiple sources. See “Creating a Union of Multiple Events” on page 2-34.
- To prioritize the reaction to one event relative to other simultaneous events. See “Translating Events to Control the Processing Sequence” on page 2-37.
- To delay the reactions to events. See the **Function-call time delay** parameter on the Signal-Based Event to Function-Call Event blocks reference page.
- To conditionalize the reactions to events. See “Conditionalizing Events” on page 2-39.

The term *event translation* refers to the conversion of one event into another. The result of the translation is often a function call, but can be another type of event. The result of the translation can occur at the same time as, or a later time than, the original event.

Blocks for Manipulating Events

The table below lists blocks that are useful for manipulating events.

Event to Manipulate	Block
Entity advancement	Entity Departure Event to Function-Call Event
Signal-based event	Signal-Based Event to Function-Call Event
Function call	Signal-Based Event to Function-Call Event
	Mux

If you connect the Entity Departure Counter block’s **#d** output port to a block that detects sample time hits or rising value changes, then you can view the counter as a mechanism for converting an entity advancement event into a signal-based event. Corresponding to each entity departure from the block is an increase in the value of the **#d** signal.

Creating a Union of Multiple Events

To generate a function-call signal that represents the union (logical OR) of multiple events, use this procedure:

- 1** Generate a function call for each event that is not already a function call. Use blocks in the Event Generators or Event Translation library.
- 2** Use the Mux block to combine the function-call signals.

The multiplexed signal carries a function call when any of the individual function-call signals carries a function call. If two individual signals carry a function call at the same time instant, then the multiplexed signal carries two function calls at that time instant.

Examples are in “Example: Performing a Computation on Selected Entity Paths” on page 10-31 and below.

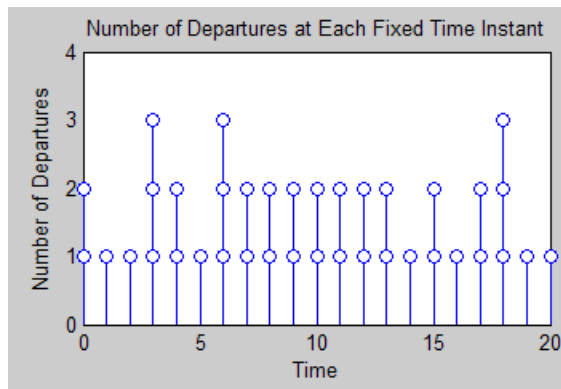
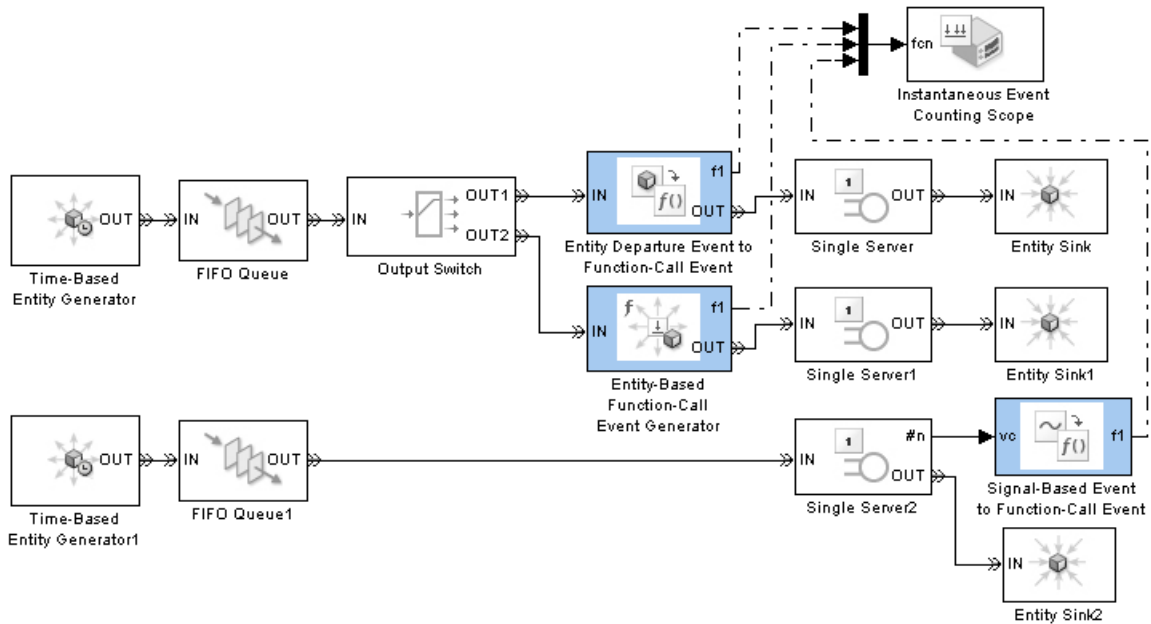
Example: Counting Events from Multiple Sources

The example below illustrates different approaches to event translation and event generation. This example varies the approach for illustrative purposes; in your own models, you might decide to use a single approach that you find most intuitive.

The goal of the example is to plot the number of arrivals at a bank of three servers at each value of time. Entities advance to the servers via one or two FIFO Queue blocks. To count arrivals and create the plot, the model translates each arrival at a server into a function call; the Mux block combines the three function-call signals to create an input to the Instantaneous Event Counting Scope block.

The three server paths use these methods for translating an entity arrival into a function call:

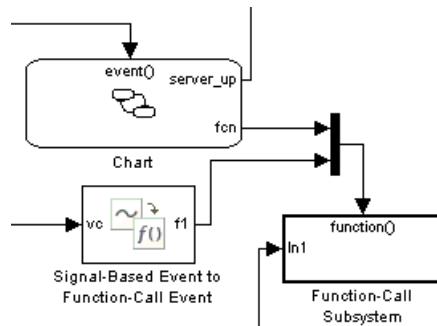
- One path uses the Entity Departure Event to Function-Call Event block, treating the problem as one of event translation.
- One path uses the Entity-Based Event Generator block, treating the problem as one of event generation. This is similar to the approach above.
- One path uses the Signal-Based Event to Function-Call Event block to translate an increase in the value of the server block's **#n** signal into a function call. This approach uses the fact that each arrival at the server block causes a simultaneous increase in the block's **#n** signal.



Example: Executing a Subsystem Based on Multiple Types of Events

You can configure a Discrete Event Subsystem block to detect signal-based events from one or more sources, and you can configure a Function-Call Subsystem block to detect function calls from one or more sources. Using an event translation block to convert a signal-based event into a function call, the fragment below effectively creates a subsystem that detects a function call from a Stateflow block and a signal-based event from another source. The subsystem is executed when either the Stateflow block generates a function call or the signal connected to the **vc** port of the Signal-Based Event to Function-Call Event block changes. If both events occur simultaneously, then the subsystem executes twice.

“Block execution” in this documentation is shorthand for “block methods execution.” Methods are functions that the Simulink engine uses to solve a model. Blocks are made up of multiple methods. For details, see “Block Methods” in the Simulink documentation.



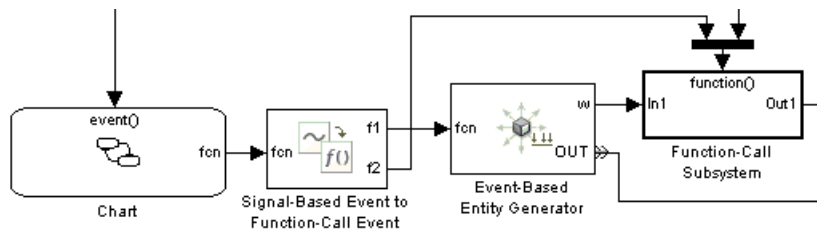
Another similar example is in “Example: Performing a Computation on Selected Entity Paths” on page 10-31.

Translating Events to Control the Processing Sequence

In some situations, event translation blocks can help you prescribe the processing sequence for simultaneous events. The examples below illustrate how to do this by taking advantage of the sequence in which an event translation block issues two function calls, and by converting an unprioritized function call into a function call having an event priority.

Example: Issuing Two Function Calls in Sequence

In the next model, entity generation and the execution of a function-call subsystem can occur simultaneously. At such times, it is important that the entity generation occur first, so that the entity generator updates the value of the **w** signal before the function-call subsystem uses **w** in its computation. This model ensures a correct processing sequence by using the same Signal-Based Event to Function-Call Event block to issue both function calls and by relying on the fact that the block always issues the **f1** function call before the **f2** function call.



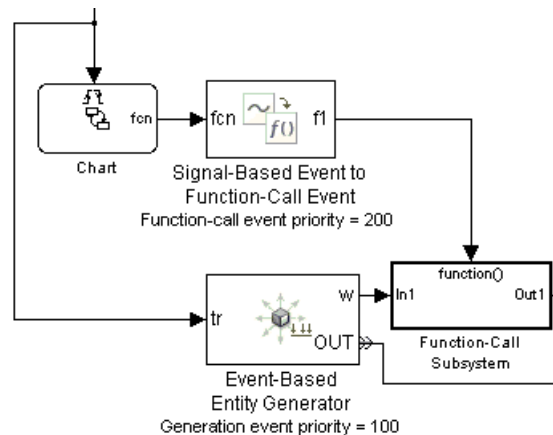
In this example, the Signal-Based Event to Function-Call Event block has this configuration:

- **Generate function call only upon** = Function call from port fcn
- **Generate optional f2 function call** selected

In this example, the Function-Call Split block in the Simulink libraries is not an alternative because it cannot connect to SimEvents blocks.

Example: Generating a Function Call with an Event Priority

The next model uses an event translation block to prioritize the execution of a function-call subsystem correctly on the event calendar, relative to a simultaneous event. In the model, a Stateflow block and an entity generator respond to edges of the same trigger signal. The Stateflow block calls an event translation block, which in turn calls a function-call subsystem. The subsystem performs a computation using the **w** output signal from the entity generator.



As in the earlier example, it is important that the entity generator update the value of the w signal before the function-call subsystem uses w in its computation. To ensure a correct processing sequence, the Signal-Based Event to Function-Call Event block replaces the original function call, which is not scheduled on the event calendar, with a new function call that is scheduled on the event calendar with a priority of 200. The Event-Based Entity Generator block schedules an entity-generation event on the event calendar with a priority of 100. As a result of the event translation and the relative event priorities, the entity generator generates the entity before the event translator issues the function call to the function-call subsystem whenever these events occur at the same value (or sufficiently close values) of the simulation clock.

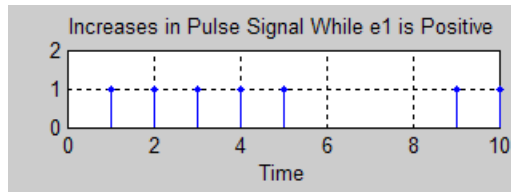
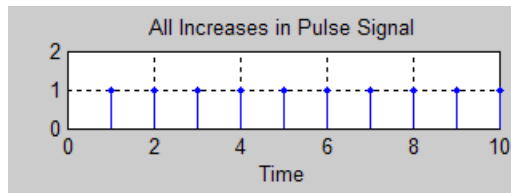
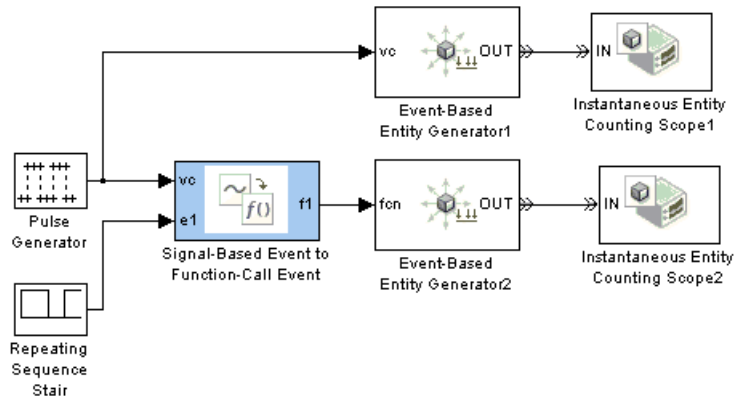
Conditionalizing Events

The Entity Departure Event to Function-Call Event and Signal-Based Event to Function-Call Event blocks provide a way to suppress the output function call based on a control signal. If the control signal is zero or negative when the block is about to issue the function call, then the block suppresses the function call. You can use this feature to

- Prevent simulation problems. The example in “Example: Detecting Changes in the Last-Updated Signal” on page 16-35 uses conditional function calls to prevent division-by-zero warnings.
- Model an inoperative state of a component of your system. See the next example.

Example: Modeling Periodic Shutdown of an Entity Generator

The example below uses Event-Based Entity Generator blocks to generate entities when a pulse signal changes its value. The top entity generator generates an entity upon each such event. The bottom entity generator responds to a function call issued by an event translation block that detects changes in the pulse signal's value. However, the event translation block issues a function call only upon value changes that occur while the **e1** input signal is positive. In this model, a nonpositive value of the **e1** signal corresponds to a failure or resting period of the entity generator.



Managing Simultaneous Events

- “Overview of Simultaneous Events” on page 3-2
- “Exploring Simultaneous Events” on page 3-4
- “Choosing an Approach for Simultaneous Events” on page 3-7
- “Assigning Event Priorities” on page 3-8
- “Example: Choices of Values for Event Priorities” on page 3-11
- “Example: Effects of Specifying Event Priorities” on page 3-26

Overview of Simultaneous Events

During a simulation, multiple events can occur at the same value of the simulation clock, whether or not due to causality. Also, the application treats events as simultaneous if their event times are sufficiently close, even if the event times are not identical. Events scheduled on the event calendar for times T and $T+\Delta t$ are considered simultaneous if $0 \leq \Delta t \leq 128 * \text{eps} * T$, where eps is the floating-point relative accuracy in MATLAB software and T is the simulation time.

This table indicates sources of relevant information that can help you understand and manage simultaneous events.

To Read About...	Refer to...	Description
Background	“Supported Events in SimEvents Models” on page 2-2	Overview of event types and the event calendar
Behavior	“Event Sequencing” on page 16-2	How the application determines which events to process first, when time and causality alone do not specify a unique sequence
Examples	“Example: Event Calendar Usage for a Queue-Server Model” on page 2-10	Illustrates basic functionality of the event calendar
	“Example: Choices of Values for Event Priorities” on page 3-11	Examines the role of event priority values
	“Example: Effects of Specifying Event Priorities” on page 3-26	Compares simulation behaviors when you specify and do not specify event priorities
Tips	“Choosing an Approach for Simultaneous Events” on page 3-7 and “Tips for Choosing Event Priority Values” on page 3-8	Tips to help you decide how to configure your model
Techniques	“Exploring Simultaneous Events” on page 3-4 and “Assigning Event Priorities” on page 3-8	Viewing behavior and working with explicit event priorities

When one of the simultaneous events is a signal update, information in “Choosing How to Resolve Simultaneous Signal Updates” on page 16-7 is also relevant.

Exploring Simultaneous Events

In this section...
“Using Nearby Breakpoints to Focus on a Particular Time” on page 3-5
“For Further Information” on page 3-5

One way that you can see details about which events occur simultaneously and the sequence in which the application processes them is by running the simulation with the SimEvents debugger. The debugger displays messages in the Command Window to indicate what is happening in the simulation, and lets you inspect states at any point where the debugger suspends the simulation. You might still need to infer some aspects of the simulation behavior that do not appear in the Command Window messages.

Tips for how you can use the debugger to explore simultaneous events, where the commands mentioned are valid at the `sedebug>>` prompt of the debugger, are:

- If you want to view the event calendar at any point in the simulation, enter `evcal`.
- If all the events you want to explore are on the event calendar and you are not interested in entity operations, enter `detail('en',0)`. The simulation log no longer issues messages about entity operations and the `step` function ignores entity operations.

The opposite command is `detail('en',1)`, which causes the simulation log to include messages about entity operations and makes it possible for `step` to suspend the simulation at an entity operation.

- If you want to see everything that happens at a particular time, use a pair of timed breakpoints, as in “Using Nearby Breakpoints to Focus on a Particular Time” on page 3-5.
- If you want to proceed in the simulation until it executes or cancels a particular event that is on the event calendar, find the event identifier (using `evcal` or the simulation log), use the event identifier in an `evbreak` command, and then enter `cont`.
- An event breakpoint is not the same as a timed breakpoint whose value equals the scheduled time of the event. The two breakpoints can cause

the simulation to stop at different points if the execution or cancelation of the event is not the first thing that happens at that value of time. For an example, see the `sedb.evbreak` reference page.

- The simulation log indicates the sequence of simultaneous events, but you might still have questions about why events are in that sequence. Referring to earlier messages in the simulation log might help answer your questions. If not, you might need to run the simulation again and inspect states at earlier points in the simulation. Debugging is often an iterative process.

Using Nearby Breakpoints to Focus on a Particular Time

- 1 Create a timed breakpoint at the time that you are interested in. For example, if you are interested in what happens at $T=3$, at the `sedbug>>` prompt, enter this command:

```
tbreak(3)
```

- 2 Enter `cont` to reach the breakpoint from step 1.

If the time that you specified in step 1 is an earlier approximation of the actual time at which something interesting happens, the simulation might stop at a time later than the time of the breakpoint. For example, suppose you guess $T=3$ from looking at a plot, but the actions of interest really occur at $T=3.0129$. In this case, having the simulation stop at $T=3.0129$ is desirable if nothing happens in the simulation at exactly $T=3$.

- 3 Create a timed breakpoint shortly after the current simulation time, by entering:

```
tbreak(simtime + 128*eps*simtime)
```

- 4 Enter `cont` to reach the next breakpoint. The portion of the simulation log between the last two `cont` commands contains the items of interest.

For Further Information

- “Overview of the SimEvents Debugger” on page 14-3 — More information about the debugger

- “Example: Choices of Values for Event Priorities” on page 3-11 — An example that explores simultaneous events and illustrates interpreting the debugger simulation log

Choosing an Approach for Simultaneous Events

When your simulation involves simultaneous events whose causality relationships do not determine a unique correct processing sequence, you might have a choice regarding their processing sequence. These tips can help you make appropriate choices:

- Several blocks offer a **Resolve simultaneous signal updates according to event priority** option. The default value, which depends on the block, is appropriate in most simulation contexts. Consider using the default value unless you have a specific reason to change it.
- If you need explicit control over the sequencing of specific kinds of simultaneous events, assign numerical event priorities for events that you want to defer until after other events are processed. For procedures and tips related to numerical event priorities, see “Assigning Event Priorities” on page 3-8.
- In some debugging situations, it is useful to see whether the simulation behavior changes when you either change the value of a block’s **Resolve simultaneous signal updates according to event priority** option or use an extreme value for an event priority. Experiments like this can help you determine which events might be sensitive to changes in the processing sequence. The debugger can also help you detect sensitivities.

For details on how the application treats simultaneous events, see “Processing Sequence for Simultaneous Events” on page 16-2 and “Resolution Sequence for Input Signals” on page 16-8.

Assigning Event Priorities

In this section...

“Procedure for Assigning Event Priorities” on page 3-8

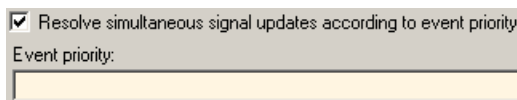
“Tips for Choosing Event Priority Values” on page 3-8

“Procedure for Specifying Equal-Priority Behavior” on page 3-9

Procedure for Assigning Event Priorities

To assign a numerical event priority to an event, use this procedure:

- 1 Find the block that produces the event you want to prioritize. For example, it might be an entity generator, a server, a gate, a counter, or a switch.
- 2 If the block’s dialog box has an option called **Resolve simultaneous signal updates according to event priority**, select this option. A parameter representing the event priority appears; in most blocks, the parameter’s name is **Event priority**.



- 3 Set the event priority parameter to a positive integer.

Note Some events have event priorities that are not numerical, such as SYS1 and SYS2. For more information about these priority values, see “System-Priority Events on the Event Calendar” on page 16-12 and “Processing Sequence for Simultaneous Events” on page 16-2.

Tips for Choosing Event Priority Values

Suppose you want to assign a numerical event priority value for Event X to defer its processing until after some simultaneous Event Y has been processed. A particular value for the event priority is not significant in

isolation; what matters is the relative handling of simultaneous events. Keep these tips in mind when choosing a value for the event priority:

- If Event Y does not have a numerical event priority, then any value for the event priority of Event X causes Event X to be processed later. (See “Processing Sequence for Simultaneous Events” on page 16-2 for details.)
- If Event Y has a numerical event priority, then choosing a larger value for the event priority of Event X causes Event X to be processed later. Simultaneous events having distinct numerical event priorities are processed in ascending order of the event priority values.
- Leaving gaps in the set of numerical values you choose lets you include additional events that require intermediate-value priorities. For example, if Event Y has priority 1 and Event X has priority 2, then you cannot force an Event Z to be processed after Event Y and before Event X. On the other hand, priority values of 100 and 200 would better accommodate future growth of your model.

For examples that show the effect of changing event priorities, see “Example: Choices of Values for Event Priorities” on page 3-11 and the Event Priorities demo.

Procedure for Specifying Equal-Priority Behavior

If simultaneous events on the event calendar share the same numerical value for their event priorities, then the application arbitrarily or randomly determines the processing sequence, depending on a modelwide configuration parameter. To set this parameter, use this procedure:

- 1** Select **Simulation > Configuration Parameters** from the model window. This opens the Configuration Parameters dialog box.
- 2** In the left pane, select **SimEvents**.
- 3** In the right pane, set **Execution order** to either **Randomized** or **Arbitrary**.
 - If you select **Arbitrary**, the application uses an internal algorithm to determine the processing sequence for events on the event calendar that have the same event priority and sufficiently close event times.

- If you select `Randomized`, the application randomly determines the processing sequence. All possible sequences have equal probability. The **Seed for event randomization** parameter is the initial seed of the random number generator; for a given seed, the generator's output is repeatable.

The processing sequence might be different from the sequence in which the events were scheduled on the event calendar.

Example: Choices of Values for Event Priorities

In this section...

“Overview of Example” on page 3-11

“Arbitrary Resolution of Signal Updates” on page 3-12

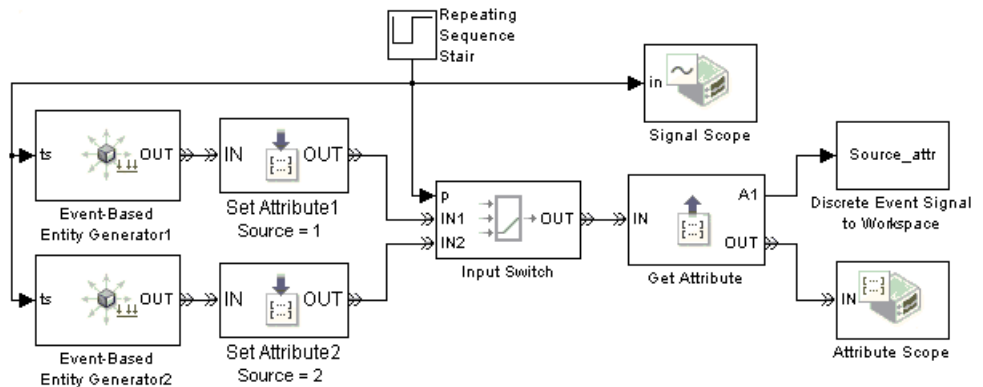
“Selecting a Port First” on page 3-12

“Generating Entities First” on page 3-19

“Randomly Selecting a Sequence” on page 3-24

Overview of Example

This example shows how you can vary the processing sequence for simultaneous events by varying their event priorities. The example creates race conditions at a switch and illustrates multiple ways to resolve the race conditions.



At $T=1, 2, 3, \dots$ the Repeating Sequence Stair block changes its value from 1 to 2 or from 2 to 1. The change causes the following events to occur, not necessarily in this sequence:

- The top entity generator generates an entity.
- The bottom entity generator generates an entity.

- The Input Switch block selects a different entity input port.

Both entity generators are configured so that if a generated entity cannot depart immediately, the generator holds the entity and temporarily suspends the generation of additional entities.

In the model, the two Set Attribute blocks assign a **Source** attribute to each entity. The attribute value is 1 or 2 depending on which entity generator generated the entity. The Attribute Scope block plots the **Source** attribute values to indicate the source of each entity that departs from the switch.

Arbitrary Resolution of Signal Updates

If the two entity generators and the switch all have the **Resolve simultaneous signal updates according to event priority** option turned off, then you cannot necessarily predict the sequence in which the blocks schedule their reactions to changes in the output signal from the Repeating Sequence Stair block.

The rest of this example assumes that the two entity generators and the switch all use the **Resolve simultaneous signal updates according to event priority** option, for greater control over the sequencing of simultaneous events.

Selecting a Port First

Suppose the two entity generators and the switch have the explicit event priorities shown.

Event Type	Event Priority
Generation event at top entity generator	300
Generation event at bottom entity generator	310
Port selection event at switch	200

The following describes what happens at $T=1, 2, 3$ using messages from the simulation log of the SimEvents debugger. To learn more about the debugger, see “Overview of the SimEvents Debugger” on page 14-3.

Behavior at T=1

- The output signal from the Repeating Sequence Stair block changes from 1 to 2 and blocks that connect to it detect the relevant update. The message from the Input Switch block indicates the previous and new values.

```

%=====
Detected Either Value Change                Time = 1.0000000000000000
: NewValue = 2
: PrevValue = 1
: Block = Input Switch
    
```

- The blocks that react to the update then schedule events on the event calendar.

```

%-----%
Events in the Event Calendar
  ID      EventTime      EventType      Priority Entity      Block
  ev4     1.0000000000000000  PortSelection  200   <none>   Input Switch
  ev5     1.0000000000000000  EntityGeneration  300   <none>   Event-Based Entity Generator1
  ev6     1.0000000000000000  EntityGeneration  310   <none>   Event-Based Entity Generator2
    
```

- The switch selects its **IN2** entity input port.

```

%=====
Executing PortSelection Event (ev4)          Time = 1.0000000000000000
: Entity = <none>                           Priority = 200
: Block = Input Switch
    
```

- The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.

```

%=====
Executing EntityGeneration Event (ev5)       Time = 1.0000000000000000
: Entity = <none>                           Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en1)
: Block = Event-Based Entity Generator1
    
```

- The bottom entity generator generates an entity. This entity advances from block to block until it reaches the Attribute Scope block, which destroys it.

```

%=====
Executing EntityGeneration Event (ev6)           Time = 1.0000000000000000
: Entity = <none>                               Priority = 310
: Block = Event-Based Entity Generator2
%.....%
Generating Entity (en2)
: Block = Event-Based Entity Generator2
%.....%
Entity Advancing (en2)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en2)
: From = Get Attribute
: To = Attribute Scope
%.....%
Destroying Entity (en2)
: Block = Attribute Scope

```

Behavior at T=2

- Blocks detect the next relevant update in the output signal from the Repeating Sequence Stair block, and react by scheduling events.

```

%-----%
Events in the Event Calendar

```

ID	EventTime	EventType	Priority	Entity	Block
ev8	2.0000000000000000	PortSelection	200	<none>	Input Switch
ev9	2.0000000000000000	EntityGeneration	300	<none>	Event-Based Entity Generator1
ev10	2.0000000000000000	EntityGeneration	310	<none>	Event-Based Entity Generator2

- The switch selects its **IN1** entity input port. This causes the top entity generator to output the entity it generated 1 second ago. This entity advances from block to block until it reaches the Attribute Scope block.

```

%=====
Executing PortSelection Event (ev8)           Time = 2.0000000000000000
: Entity = <none>                               Priority = 200

```

```

: Block = Input Switch
%.....%
Entity Advancing (en1)
: From = Event-Based Entity Generator1
: To = Set Attribute1
%.....%
[...other messages...]
%.....%
Entity Advancing (en1)
: From = Get Attribute
: To = Attribute Scope
%.....%
Destroying Entity (en1)
: Block = Attribute Scope

```

- The top entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block. A total of two entities from the top entity generator reach the scope at this time instant.

```

%=====
Executing EntityGeneration Event (ev9)           Time = 2.0000000000000000
: Entity = <none>                               Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en3)
: Block = Event-Based Entity Generator1
%.....%
Entity Advancing (en3)
: From = Event-Based Entity Generator1
: To = Set Attribute1
%.....%
[...other messages...]
%.....%
Entity Advancing (en3)
: From = Get Attribute
: To = Attribute Scope
%.....%
Destroying Entity (en3)
: Block = Attribute Scope

```

- The bottom entity generator generates an entity, which cannot depart because the switch's **IN2** entity input port is unavailable.

```

%=====
Executing EntityGeneration Event (ev10)           Time = 2.000000000000000
: Entity = <none>                                Priority = 310
: Block = Event-Based Entity Generator2
sedebug>>step over
%.....%
Generating Entity (en4)
: Block = Event-Based Entity Generator2

```

Behavior at T=3

- Blocks detect the next relevant update in the output signal from the Repeating Sequence Stair block, and react by scheduling events.

```

%-----%
Events in the Event Calendar

```

ID	EventTime	EventType	Priority	Entity	Block
ev13	3.000000000000000	PortSelection	200	<none>	Input Switch
ev14	3.000000000000000	EntityGeneration	300	<none>	Event-Based Entity Generator1
ev15	3.000000000000000	EntityGeneration	310	<none>	Event-Based Entity Generator2

- The switch selects its **IN2** entity input port. This causes the bottom entity generator to output the entity it generated 1 second ago. This entity advances from block to block until it reaches the Attribute Scope block.

```

%=====
Executing PortSelection Event (ev13)           Time = 3.000000000000000
: Entity = <none>                                Priority = 200
: Block = Input Switch
%.....%
Entity Advancing (en4)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en4)

```



```

: From = Get Attribute
: To   = Attribute Scope
%.....%
Destroying Entity (en4)
: Block = Attribute Scope

```

- The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.

```

%=====
Executing EntityGeneration Event (ev14)           Time = 3.0000000000000000
: Entity = <none>                                Priority = 300
: Block  = Event-Based Entity Generator1
%.....%
Generating Entity (en5)
: Block  = Event-Based Entity Generator1

```

- The bottom entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block. A total of two entities from the bottom entity generator reach the scope at this time instant.

```

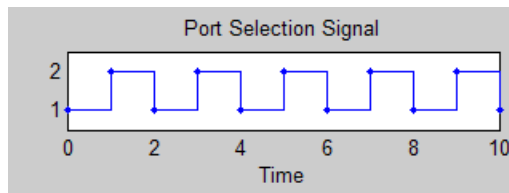
%=====
Executing EntityGeneration Event (ev15)           Time = 3.0000000000000000
: Entity = <none>                                Priority = 310
: Block  = Event-Based Entity Generator2
%.....%
Generating Entity (en6)
: Block  = Event-Based Entity Generator2
%.....%
Entity Advancing (en6)
: From = Event-Based Entity Generator2
: To   = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en6)
: From = Get Attribute
: To   = Attribute Scope
%.....%
Destroying Entity (en6)

```

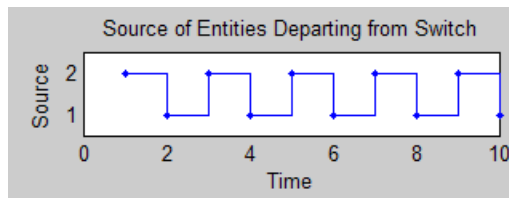
: Block = Attribute Scope

Evidence from Plots and Signals

The plot of entities' Source attribute values shows an alternating pattern of dots, as does the plot of the port selection signal **p**. The list of times and values of the entities' Source attribute, as recorded in the `Source_attr` variable in the MATLAB workspace, shows that two entities from the same entity generator reach the scope at $T=2, 3, 4$, etc.



Port Selection Signal



Switch Departures When Port Selection Is Processed First

```
[Source_attr.time, Source_attr.signals.values]
```

```
ans =
```

```
1     2  
2     1  
2     1  
3     2  
3     2  
4     1  
4     1  
5     2  
5     2
```

```

6      1
6      1
7      2
7      2
8      1
8      1
9      2
9      2
10     1
10     1
    
```

Generating Entities First

Suppose the two entity generators and the switch have the explicit event priorities shown below.

Event Type	Event Priority
Generation event at top entity generator	300
Generation event at bottom entity generator	310
Port selection event at switch	4000

At the beginning of the simulation, the port selection signal, **p**, is 1.

Behavior at T=1

- The output signal from the Repeating Sequence Stair block changes from 1 to 2 and blocks that connect to it detect the relevant update. The message from the Input Switch block indicates the previous and new values.

```

%=====
Detected Either Value Change                               Time = 1.0000000000000000
: NewValue = 2
: PrevValue = 1
: Block    = Input Switch
    
```

- The blocks that react to the update then schedule events on the event calendar.

```

%-----%
    
```

Events in the Event Calendar

ID	EventTime	EventType	Priority	Entity	Block
ev5	1.0000000000000000	EntityGeneration	300	<none>	Event-Based Entity Generator1
ev6	1.0000000000000000	EntityGeneration	310	<none>	Event-Based Entity Generator2
ev4	1.0000000000000000	PortSelection	4000	<none>	Input Switch

- The top entity generator generates an entity. This entity advances from block to block until it reaches the Attribute Scope block, which destroys it.

```

%=====
Executing EntityGeneration Event (ev5)           Time = 1.0000000000000000
: Entity = <none>                               Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en1)
: Block = Event-Based Entity Generator1
%.....%
Entity Advancing (en1)
: From = Event-Based Entity Generator1
: To = Set Attribute1
%.....%
[...other messages...]
%.....%
Entity Advancing (en1)
: From = Get Attribute
: To = Attribute Scope

```

- The bottom entity generator generates an entity, which cannot depart because the switch's **IN2** entity input port is unavailable.

```

%=====
Executing EntityGeneration Event (ev6)           Time = 1.0000000000000000
: Entity = <none>                               Priority = 310
: Block = Event-Based Entity Generator2
%.....%
Generating Entity (en2)
: Block = Event-Based Entity Generator2

```

- The switch selects its **IN2** entity input port. This causes the bottom entity generator to output the entity it just generated. This entity advances from block to block until it reaches the Attribute Scope block.

```

%=====
Executing PortSelection Event (ev4)                Time = 1.0000000000000000
: Entity = <none>                                Priority = 4000
: Block = Input Switch
%.....%
Entity Advancing (en2)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en2)
: From = Get Attribute
: To = Attribute Scope
    
```

Behavior at T=2

- Blocks detect the next relevant update in the output signal from the Repeating Sequence Stair block, and react by scheduling events.

```

%-----%
Events in the Event Calendar
  ID      EventTime      EventType      Priority Entity      Block
  ev10    2.0000000000000000 EntityGeneration 300    <none>    Event-Based Entity Generator1
  ev11    2.0000000000000000 EntityGeneration 310    <none>    Event-Based Entity Generator2
  ev9     2.0000000000000000 PortSelection   4000   <none>    Input Switch
    
```

- The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.

```

%=====
Executing EntityGeneration Event (ev10)           Time = 2.0000000000000000
: Entity = <none>                                Priority = 300
: Block = Event-Based Entity Generator1
%.....%
Generating Entity (en3)
    
```

```
: Block = Event-Based Entity Generator1
```

- The bottom entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block.

```
%=====
Executing EntityGeneration Event (ev11)           Time = 2.0000000000000000
: Entity = <none>                                Priority = 310
: Block = Event-Based Entity Generator2
%.....%
Generating Entity (en4)
: Block = Event-Based Entity Generator2
%.....%
Entity Advancing (en4)
: From = Event-Based Entity Generator2
: To = Set Attribute2
%.....%
[...other messages...]
%.....%
Entity Advancing (en4)
: From = Get Attribute
: To = Attribute Scope
```

- The switch selects its **IN1** entity input port. This causes the top entity generator to output the entity it just generated. This entity advances from block to block until it reaches the Attribute Scope block.

```
%=====
Executing PortSelection Event (ev9)               Time = 2.0000000000000000
: Entity = <none>                                Priority = 4000
: Block = Input Switch
sedebug>>step over
%.....%
Entity Advancing (en3)
: From = Event-Based Entity Generator1
: To = Set Attribute1
%.....%
[...other messages...]
%.....%
Entity Advancing (en3)
```

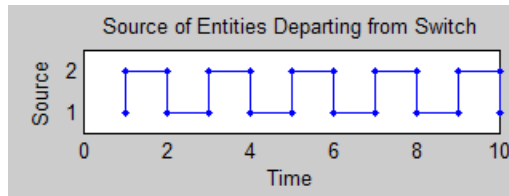
```

: From = Get Attribute
: To   = Attribute Scope

```

Plots and Signals

The plot of entities' Source attribute values shows that two entities from different entity generators depart from the switch every second.



Switch Departures When Entity Generations Are Processed First

```
[Source_attr.time, Source_attr.signals.values]
```

```
ans =
```

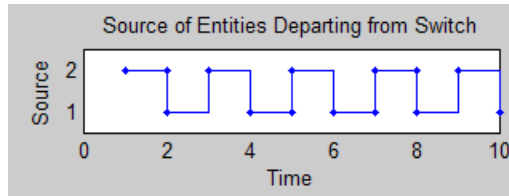
1	1
1	2
2	2
2	1
3	1
3	2
4	2
4	1
5	1
5	2
6	2
6	1
7	1
7	2
8	2
8	1
9	1
9	2
10	2

10 1

Randomly Selecting a Sequence

Suppose the two entity generators and the switch have equal event priorities. By default, the application uses an arbitrary processing sequence for the entity-generation events and the port-selection events, which might or might not be appropriate in an application. To avoid bias by randomly determining the processing sequence for these events, set **Execution order** to **Randomized** in the model's Configuration Parameters dialog box.

Sample attribute values and the corresponding plot are below, but your results might vary depending on the specific random numbers.



Switch Departures When Processing Sequence is Random

```
[Source_attr.time, Source_attr.signals.values]
```

```
ans =
```

```

1      2
2      2
2      1
3      2
4      1
4      1
5      1
5      2
5      2
6      1
7      1
7      2
8      2
8      1
```


9	2
10	1
10	1

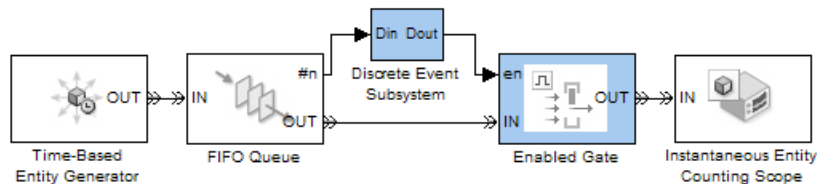
Example: Effects of Specifying Event Priorities

In this section...
“Overview of the Example” on page 3-26
“Default Behavior” on page 3-27
“Deferring Gate Events” on page 3-28
“Deferring Subsystem Execution to the Event Calendar” on page 3-30

Overview of the Example

This example illustrates how selecting or clearing the **Resolve simultaneous signal updates according to event priority** option—which influences whether or how an event is scheduled on the event calendar—can significantly affect simulation behavior. In particular, the example illustrates how deferring the reaction to a signal update can change how a gate lets entities out of a queue.

In this model, the Discrete Event Subsystem block returns 1 when the queue length is greater than or equal to 5, and returns 0 otherwise. When the subsystem returns 1, the gate opens to let one or more entities depart from the queue.



The number of entities departing from the queue at a given time depends on the **Resolve simultaneous signal updates according to event priority** parameter settings in the subsystem and the Enabled Gate block, as explained in the next section.

Default Behavior

By default, the Discrete Event Inport block inside the subsystem and the Enabled Gate block at the top level both have the **Resolve simultaneous signal updates according to event priority** option not selected. Neither block has any events with numerical priority values. The simulation behaves as follows:

Simulation Behavior

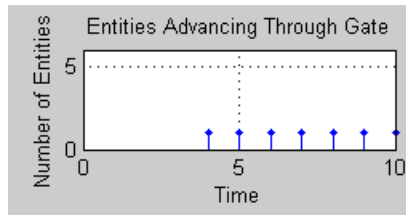
- 1** The queue accumulates entities until it updates the queue length signal, **#n**, to 5.
- 2** The subsystem executes immediately because the execution is not scheduled on the event calendar. The subsystem reports, and the gate detects, that the queue length is at the threshold.
- 3** The gate schedules an event to open. The event has priority SYS1.

Note Some events have event priorities that are not numerical, such as SYS1 and SYS2. For more information about these priority values, see “System-Priority Events on the Event Calendar” on page 16-12 and “Processing Sequence for Simultaneous Events” on page 16-2.

- 4** The application executes the event, and the gate opens.
- 5** One entity departs from the queue.
- 6** The gate schedules an event to request another entity. The event has priority SYS2.
- 7** The queue length decreases.
- 8** As a consequence of the queue length’s decrease, the subsystem executes immediately because the execution is not scheduled on the event calendar. The subsystem finds that the queue length is beneath the threshold.
- 9** The gate schedules an event to close. The event has priority SYS1.

- 10 The application executes the gate event. (Note that the application processes events with priority SYS1 before processing events with priority SYS2.) The gate closes.
- 11 The application executes the entity request event, but it has no effect because the gate is already closed.
- 12 Time advances until the next entity generation, at which point the cycle repeats.

In summary, when the queue length reaches the threshold, the gate permits exactly one entity to advance and then closes. This is because the subsystem reevaluates the threshold condition upon detecting the change in #n, and the gate's closing event has higher priority than its entity request event. The plots of departures from the gates reflect this behavior.



The rest of this example modifies the model to permit the queue to empty completely. The strategies are either to defer the reevaluation of the threshold condition or to defer the gate's reaction to the reevaluated threshold condition.

Deferring Gate Events

To illustrate how specifying a numerical event priority for the gate can defer its closing until more entities have advanced, open the original model and modify it as follows:

Procedure

- 1 Open the Enabled Gate block's dialog box by double-clicking the block.
- 2 Select **Resolve simultaneous signal updates according to event priority**.

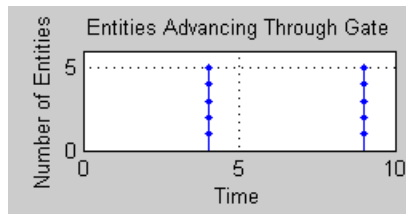
The change causes the gate to prioritize its events differently. The application processes events with priority SYS1 before processing events with numerical priority values. As a result, the simulation behaves as follows:

Simulation Behavior

- 1** The queue accumulates entities until it updates the queue length signal, #n, to 5.
- 2** The subsystem executes immediately because the execution is not scheduled on the event calendar. The subsystem finds that the queue length is at the threshold.
- 3** The gate schedules an event to open. The event has a numerical priority value.
- 4** The application executes the event, and the gate opens.
- 5** One entity departs from the queue.
- 6** The gate schedules an event to request another entity. The event has priority SYS2.
- 7** The queue length decreases.
- 8** As a consequence of the queue length's decrease, the subsystem executes immediately because the execution is not scheduled on the event calendar. The subsystem finds that the queue length is beneath the threshold.
- 9** The gate schedules an event to close. The event has a numerical priority value.
- 10** The application executes the entity request event.
- 11** Steps 5 through 10 repeat until the queue is empty. The gate remains open during this period. This repetition shows the difference in simulation behavior between SYS1 and a numerical value as the event priority for the gate event.
- 12** The application executes the gate event. The gate closes.

- 13 Time advances until the next entity generation, at which point the queue begins accumulating entities again.

In summary, when the queue length reaches the threshold, the gate permits the queue to become empty. This is because the gate does not react to the reevaluated threshold condition until after other simultaneous operations have been processed. The plot of departures from the gates reflect this behavior.



Deferring Subsystem Execution to the Event Calendar

To illustrate how scheduling a subsystem execution on the event calendar can defer the gate's closing until more entities have advanced, open the original model and modify it as follows:

Procedure

- 1 Open the subsystem by double-clicking the Discrete Event Subsystem block.
- 2 Open the inport block's dialog box by double-clicking the block labeled Din.
- 3 Select **Resolve simultaneous signal updates according to event priority**.

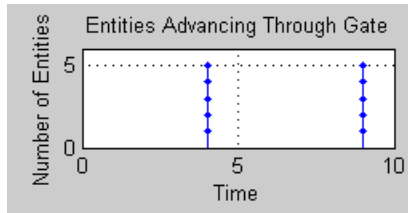
The change causes the subsystem to react to sample time hits in its input signal by scheduling an event to reevaluate the threshold condition, instead of reevaluating it immediately. As a result, the simulation behaves as follows:

Simulation Behavior

- 1 The queue accumulates entities until it updates the queue length signal, **#n**, to 5.

- 2** The subsystem schedules an execution event for the current time.
- 3** The event calendar causes the subsystem to execute. The subsystem finds that the queue length is at the threshold.
- 4** The gate schedules an event to open. The event has priority SYS1.
- 5** The application executes the event, and the gate opens.
- 6** One entity departs from the queue.
- 7** The gate schedules an event to request another entity. The event has priority SYS2.
- 8** The queue length decreases.
- 9** As a consequence of the queue length's decrease, the subsystem schedules an execution event for the current time. The event has a numerical priority value.
- 10** The application executes the entity request event. (Note that the application processes events with priority SYS2 before processing events with numerical priority values.)
- 11** Steps 6 through 10 repeat until the queue is empty. The gate remains open during this period. This repetition shows the difference in simulation behavior between scheduling the execution event and processing it immediately upon detecting the decrease in queue length.
- 12** The event calendar causes the subsystem to execute. The subsystem finds that the queue length is beneath the threshold.
- 13** The gate schedules an event to close.
- 14** The application executes the gate event. The gate closes.
- 15** The event calendar causes the subsystem to execute additional times, but the subsystem output is the same.
- 16** Time advances until the next entity generation, at which point the queue begins accumulating entities again.

In summary, when the queue length reaches the threshold, the gate permits the queue to become empty. This is because the subsystem does not reevaluate the threshold condition until after other simultaneous operations have been processed. The plot of departures from the gates reflect this behavior.



Working with Signals

- “Role of Event-Based Signals in SimEvents Models” on page 4-2
- “Generating Random Signals” on page 4-4
- “Using Data Sets to Create Event-Based Signals” on page 4-9
- “Manipulating Signals” on page 4-12
- “Sending Data to the MATLAB Workspace” on page 4-15
- “Working with Multivalued Signals” on page 4-19

Role of Event-Based Signals in SimEvents Models

In this section...
“Overview of Event-Based Signals” on page 4-2
“Comparison with Time-Based Signals” on page 4-2

Overview of Event-Based Signals

Discrete-event simulations often involve signals that change when events occur; for example, the number of entities in a server is a statistical output signal from a server block and the signal value changes when an entity arrives at or departs from the server. An event-based signal is a signal that can change in response to discrete events. Most output signals from SimEvents blocks are event-based signals.

Comparison with Time-Based Signals

Unlike time-based signals, event-based signals

- Do not have a true sample time. (These are *not* continuous signals, even though the sample time coloration feature makes the signal connection line black or gray, and a Probe block reports a sample time of zero.)
- Might be updated at time instants that do not correspond to time steps determined by time-based dynamics.
- Might undergo multiple updates in a single time instant.

For example, consider a signal representing the number of entities in a server. Computing this value at fixed intervals is wasteful if no entities arrive or depart for long periods. Computing the value at fixed intervals is inaccurate if entities arrive or depart in the middle of an interval, because the computation misses those events. Simultaneous events can make the signal multivalued; for example, if an entity completes its service and departs, which permits another entity to arrive at the same time instant, then the count at that time equals both 0 and 1 at that time instant. Furthermore, if an updated value of the count signal causes an event, then the processing of the signal update relative to other operations at that time instant can affect the processing sequence of simultaneous events and change the behavior of the simulation.

When you use output signals from SimEvents blocks to examine the detailed behavior of your system, you should understand when the blocks update the signals, including the possibility of multiple simultaneous updates. When you use event-based signals for controlling the dynamics of the simulation, understanding when blocks update the signals and when other blocks react to the updated values is even more important.

An event-based signal cannot be an element of a nonvirtual bus while retaining event-based timing. The reason is that a nonvirtual bus is an inherently time-based signal.

Note Blocks in the SimEvents libraries process signals whose data type is `double`. To convert between data types, use the Data Type Conversion block in the Simulink Signal Attributes library.

Generating Random Signals

In this section...

“Generating Random Event-Based Signals” on page 4-4

“Examples of Random Event-Based Signals” on page 4-5

“Generating Random Time-Based Signals” on page 4-6

Generating Random Event-Based Signals

The Event-Based Random Number block is designed to create event-based signals using a variety of distributions. The block generates a new random number from the distribution upon notifications from a port of a subsequent block. For example, when connected to the **t** input port of a Single Server block, the Event-Based Random Number block generates a new random number each time it receives notification that an entity has arrived at the server. The **t** input port of a Single Server block is an example of a notifying port; for a complete list, see “Notifying Ports” on page 16-21. You must connect the Event-Based Random Number block to exactly one notifying port, which then tells the block when to generate a new output value.

For details on the connectivity restrictions of the Event-Based Random Number block, see its reference page.

Generating Random Signals Based on Arbitrary Events

A flexible way to generate random event-based signals is to use the Signal Latch block to indicate explicitly which events cause the Event-Based Random Number block to generate a new random number. Use this procedure:

- 1 Insert an Event-Based Random Number block into your model and configure it to indicate the distribution and parameters you want to use.
- 2 Insert a Signal Latch block and set **Read from memory upon** to **Write to memory** event. The block no longer has an **rvc** signal input port.
- 3 Determine which events should result in the generation of a new random number, and set the Signal Latch block’s **Write to memory upon** accordingly.

- 4 Connect the signal whose events you identified in the previous step to the write-event port (**wts**, **wvc**, **wtr**, or **wfen**) of the Signal Latch block. Connect the Event-Based Random Number block to the **in** port of the Signal Latch block.

The **out** port of the Signal Latch block is the desired random event-based signal.

Examples of Random Event-Based Signals

Here are some examples using the Event-Based Random Number block:

- “Example: Using an Arbitrary Discrete Distribution as Intergeneration Time” in the SimEvents getting started documentation
- “Example: A Packet Switch” in the SimEvents getting started documentation
- “Example: Using Random Service Times in a Queuing System” in the SimEvents getting started documentation
- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-10
- “Example: M/M/5 Queuing System” on page 5-18
- “Example: Compound Switching Logic” on page 6-7

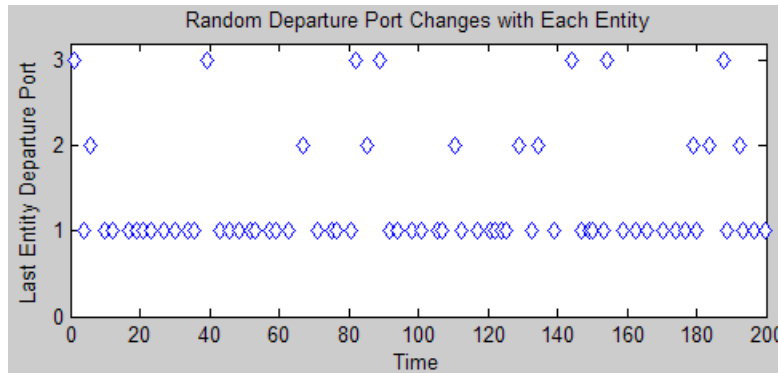
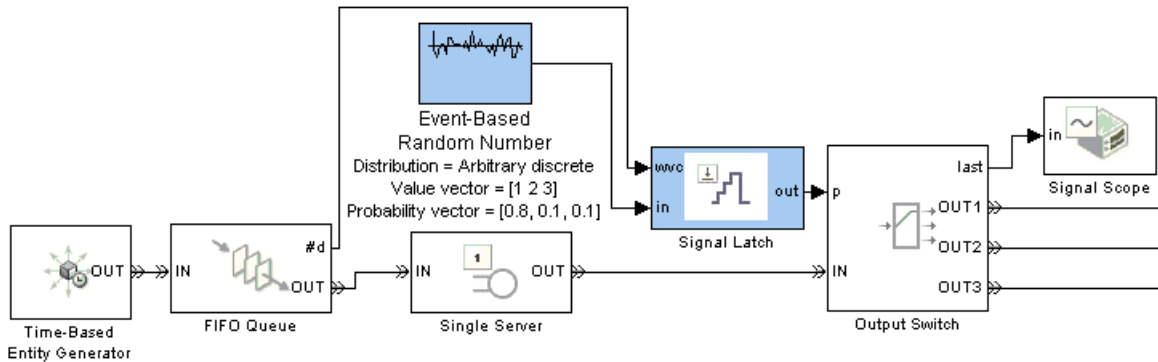
The model in “Example: Compound Switching Logic” on page 6-7 also illustrates how to use the Signal Latch block as described in “Generating Random Signals Based on Arbitrary Events” on page 4-4, to generate a random number upon each departure from an Input Switch block.

The models in “Example: Invalid Connection of Event-Based Random Number Generator” on page 14-86 illustrate how to follow the connection rules for the Event-Based Random Number block.

Example: Creating a Random Signal for Switching

The model below, similar to the one in “Example: Using Entity-Based Timing for Choosing a Port” on page 10-30, implements random output switching with a skewed distribution. The Signal Latch block causes the Event-Based Random Number block to generate a new random number upon each increase in the FIFO Queue block’s **#d** output signal, that is, each time an entity

advances from the queue to the server. The random number becomes the switching criterion for the Output Switch block that follows the server. The plot reflects the skewed probability defined in the Event-Based Random Number block, which strongly favors 1 instead of 2 or 3.



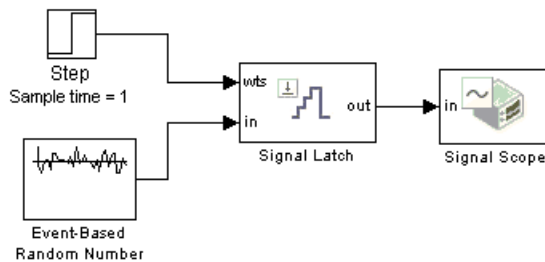
Generating Random Time-Based Signals

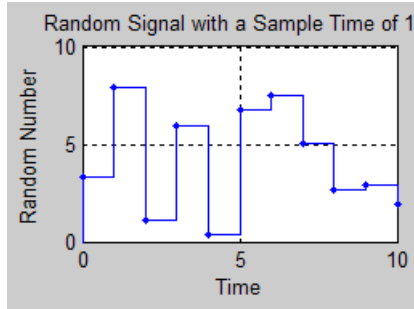
The Random Number and Uniform Random Number blocks in the Simulink Sources library create time-based random signals with Gaussian and uniform distributions, respectively. The Event-Based Random Number block supports other distributions, but is designed to create event-based signals. To generate time-based random signals using the Event-Based Random Number block, use this procedure:

- 1 Insert an Event-Based Random Number block into your model and configure it to indicate the distribution and parameters you want to use.
- 2 Insert and configure a Signal Latch block:
 - a Set **Write to memory upon** to Sample time hit from port **wts**.
 - b Set **Read from memory upon** to Write to memory event.

The block now has input ports **wts** and **in**, but not **wvc** or **rvc**.
- 3 Insert a Step block (or another time-based source block) and set **Sample time** to the desired sample time of the time-based signal you want to create.
- 4 Connect the Step block to the **wts** port of the Signal Latch block. Connect the Event-Based Random Number block to the **in** port of the Signal Latch block.

The **out** port of the Signal Latch block is a time-based signal whose sample time is the one specified in the Step block and whose values come from the Event-Based Random Number block. An example is below.





Using Data Sets to Create Event-Based Signals

In this section...
“Behavior of the Event-Based Sequence Block” on page 4-9
“Generating Sequences Based on Arbitrary Events” on page 4-10

Behavior of the Event-Based Sequence Block

Suppose you have a set of measured or expected service times for a server in the system you are modeling and you want to use that data in the simulation. You can use the Event-Based Sequence block to create a signal whose sequence of values comes from the data set and whose timing corresponds to relevant events, which in this case are the arrivals of entities at the server. You do not need to know in advance when entities will arrive at the server because the Event-Based Sequence block automatically infers from the server when to output the next value in the data set.

More generally, you can use the Event-Based Sequence block to incorporate your data into a simulation via event-based signals, where the block infers from a subsequent block when to output the next data value. You must connect the Event-Based Sequence block to exactly one notifying port, which tells the block when to generate a new output value. The **t** input port of a Single Server block is an example of a notifying port; for a list, see “Notifying Ports” on page 16-21.

The Event-Based Sequence reference page provides details on the connectivity restrictions of this block.

For examples using this block, see these sections:

- “Specifying Generation Times for Entities” on page 1-5
- “Example: Counting Simultaneous Departures from a Server” on page 1-21
- “Example: Setting Attributes” on page 1-10

Generating Sequences Based on Arbitrary Events

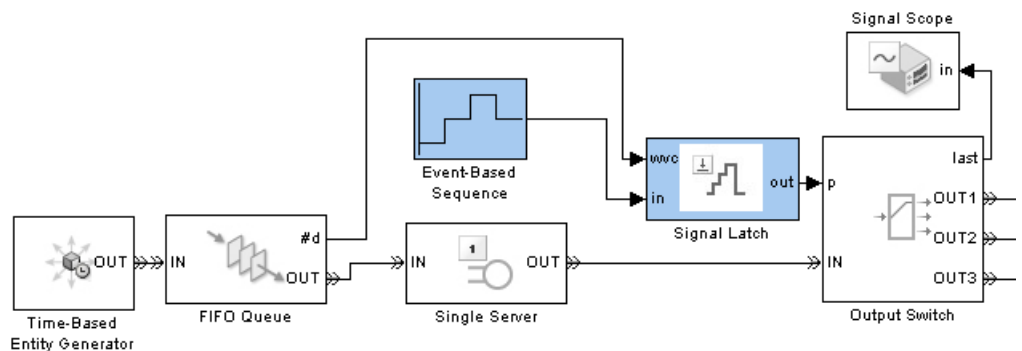
A flexible way to generate event-based sequences is to use the Signal Latch block to indicate explicitly which events cause the Event-Based Sequence block to generate a new output value. Use this procedure:

- 1 Insert an Event-Based Sequence block into your model and configure it to indicate the data you want to use.
- 2 Insert a Signal Latch block and set **Read from memory upon** to **Write to memory** event. The block no longer has an **rvc** signal input port.
- 3 Determine which events should result in the output of the next data value, and set the Signal Latch block's **Write to memory upon** accordingly.
- 4 Connect the signal whose events you identified in the previous step to the write-event port (**wts**, **wvc**, **wtr**, or **wfcn**) of the Signal Latch block. Connect the Event-Based Sequence block to the **in** port of the Signal Latch block.

The **out** port of the Signal Latch block is the desired event-based sequence.

Example

You can modify the model in “Example: Creating a Random Signal for Switching” on page 4-5 by replacing the Event-Based Random Number block with the Event-Based Sequence block.



This causes the model's Output Switch to select ports based on the data you provide. If you set the Event-Based Sequence block's **Vector of output values** parameter to `[1 2 3 2].'`, for example, then the switch selects ports 1, 2, 3, 2, 1, 2, 3, 2, 1,... as entities leave the queue during the simulation. If you change **Form output after final data value by** to `Holding final value`, then the switch selects ports 1, 2, 3, 2, 2, 2, 2,... instead.

Manipulating Signals

In this section...

“Specifying Initial Values of Event-Based Signals” on page 4-12

“Example: Resampling a Signal Based on Events” on page 4-13

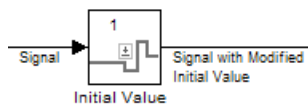
Specifying Initial Values of Event-Based Signals

Use the Initial Value block to modify the value that an event-based signal assumes between the start of the simulation and the first event affecting that signal. This technique is especially useful for output signals from Discrete Event Subsystem blocks, Stateflow blocks, and event-based signals that are part of feedback loops.

To define the initial value of an event-based signal, use this procedure:

- 1 Set the **Value until first sample time hit** parameter in the Initial Value block to your desired initial value.
- 2 Insert the Initial Value block on the line representing the signal whose initial value you want to modify.

The next schematic illustrates the meaning of the input and output signals of the Initial Value block.



The Initial Value block’s output signal uses your initial value until your original signal has its first sample time hit (that is, its first update). Afterward, the output signal and your original signal are identical.

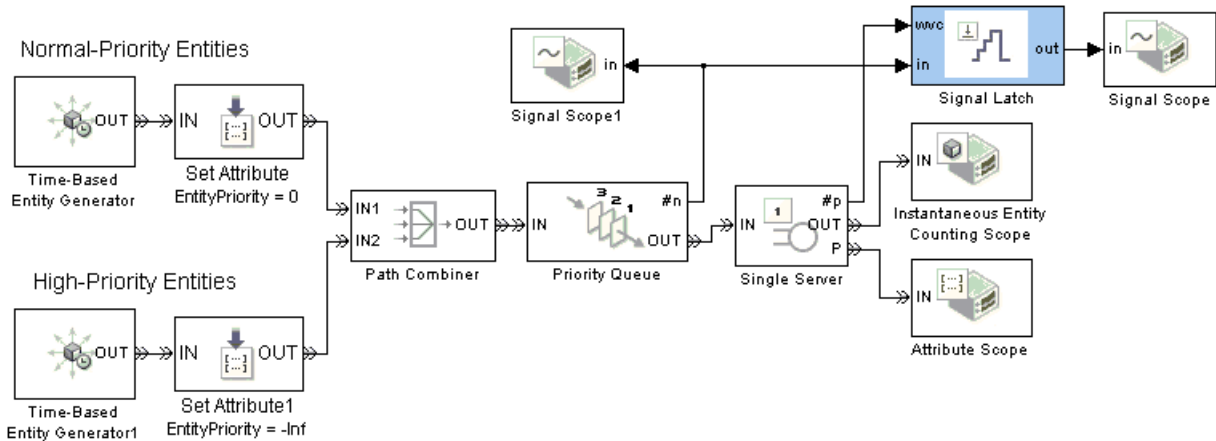
The following examples illustrate this technique:

- “Example: Controlling Joint Availability of Two Servers” on page 8-4 initializes an event-based signal for use in a feedback loop.

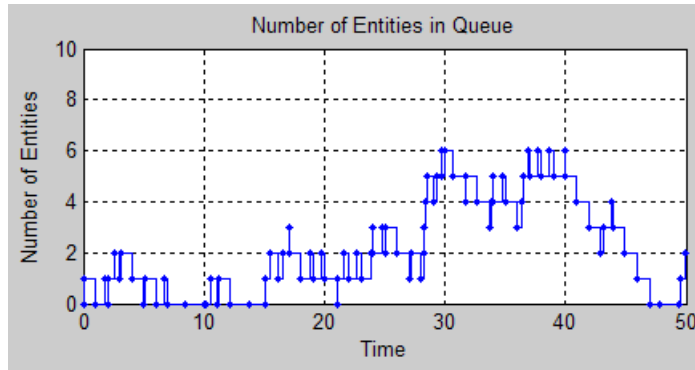
- “Example: Failure and Repair of a Server” on page 5-22 initializes an event-based signal that is the output of a Stateflow block.

Example: Resampling a Signal Based on Events

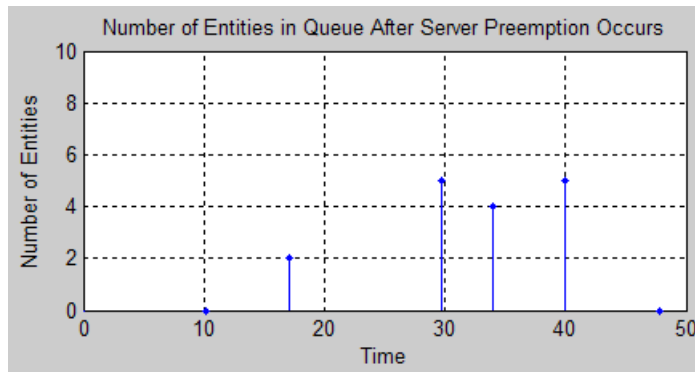
The example below contains a server that supports preemption of normal-priority entities by high-priority entities. This is similar to “Example: Preemption by High-Priority Entities” on page 5-12. Suppose that a preemption and the subsequent service of a high-priority entity represents a time interval during which the server is inoperable. The goal of this example is to find out how many entities are in the queue when the breakdown begins.



A plot of the Priority Queue block’s #n output signal indicates how many entities are in the queue at all times during the simulation.



The Signal Latch block resamples the $\#n$ signal, focusing only on the values that $\#n$ assumes when a high-priority queue preempts an entity already in the server. The Signal Latch block outputs a sample from the $\#n$ signal whenever the Single Server block's $\#p$ output signal increases, where $\#p$ is the number of entities that have been preempted from the server. Between pairs of successive preemption events, the Signal Latch block does not update its output signal, ignoring changes in $\#n$. A plot of the output from the Signal Latch block makes it easier to see how many entities are in the queue when the breakdown begins, compared to the plot of the entire $\#n$ signal.



Sending Data to the MATLAB Workspace

In this section...

“Behavior of the Discrete Event Signal to Workspace Block” on page 4-15

“Example: Sending Queue Length to the Workspace” on page 4-15

“Using the To Workspace Block with Event-Based Signals” on page 4-18

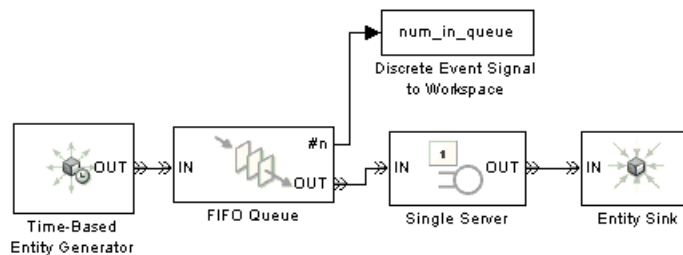
Behavior of the Discrete Event Signal to Workspace Block

The Discrete Event Signal to Workspace block writes event-based signals to the MATLAB workspace when the simulation stops or pauses. One way to pause a running simulation is to select **Simulation > Pause**. When inside a discrete event subsystem, the To Workspace block can also be useful for writing event-based signals to the MATLAB workspace.

Note To learn how to read data from the workspace during a discrete-event simulation, see “Using Data Sets to Create Event-Based Signals” on page 4-9.

Example: Sending Queue Length to the Workspace

The example below shows one way to write the times and values of an event-based signal to the MATLAB workspace. In this case, the signal is the **#n** output from a FIFO Queue block, which indicates how many entities the queue holds.



After you run this simulation, you can use the following code to create a two-column matrix containing the time values in the first column and the signal values in the second column.

```
times_values = [num_in_queue.time, num_in_queue.signals.values]
```

The output below reflects the Time-Based Entity Generator block's constant intergeneration time of 0.8 second and the Single Server block's constant service time of 1.1 second.


```
times_values =  
  
      0      0  
0.8000  1.0000  
1.1000      0  
1.6000  1.0000  
2.2000      0  
2.4000  1.0000  
3.2000  2.0000  
3.3000  1.0000  
4.0000  2.0000  
4.4000  1.0000  
4.8000  2.0000  
5.5000  1.0000  
5.6000  2.0000  
6.4000  3.0000  
6.6000  2.0000  
7.2000  3.0000  
7.7000  2.0000  
8.0000  3.0000  
8.8000  4.0000  
8.8000  3.0000  
9.6000  4.0000  
9.9000  3.0000
```

From the output, you can see that the number of entities in the queue increases at times that are a multiple of 0.8, and decreases at times that are a multiple of 1.1. At $T=8.8$, a departure from the server and an entity generation occur simultaneously; both events influence the number of entities in the queue. The output shows two values corresponding to $T=8.8$, enabling you to see the zero-duration value that the signal assumes at this time.

Using the To Workspace Block with Event-Based Signals

The To Workspace block in the Simulink Sinks library can be useful for working with event-based signals in special ways, such as

- Omitting repeated values of the signal and focusing on changes in the signal's value. For an example, see “Example: Sending Unrepeated Data to the MATLAB Workspace” on page 10-22.
- Recording values of multiple signals when any *one* of the signals has an update. To accomplish this, place multiple To Workspace blocks in a discrete event subsystem that has multiple input ports.

If you use the To Workspace block in the Simulink Sinks library to write event-based signals to the MATLAB workspace, you should

- 1** Set the block's **Save format** parameter to **Structure With Time**, which causes the block to record time values, not just signal values.
- 2** Place the To Workspace block in a discrete event subsystem to ensure that the workspace variable records data at appropriate times during the simulation.

For more details about discrete event subsystems, see “Role of Discrete Event Subsystems in SimEvents Models” on page 10-7.

Working with Multivalued Signals

In this section...
“Zero-Duration Values of Signals” on page 4-19
“Importance of Zero-Duration Values” on page 4-20
“Detecting Zero-Duration Values” on page 4-20

Zero-Duration Values of Signals

Some output signals from SimEvents blocks produce a new output value for each departure from the block. When multiple departures occur in a single time instant, the result is a multivalued signal. That is, at a given instant in time, the signal assumes multiple values in sequence. The sequence of values corresponds to the sequence of departures. Although the departures and values have a well-defined sequence, no time elapses between adjacent events.

Scenario: Server Departure and New Arrival

For example, consider the scenario in which an entity departs from a single server at time T and, consequently, permits another entity to arrive from a queue that precedes the server. The statistic representing the number of entities in the server is 1 just before time T because the first entity has not completed its service. The statistic is 1 just after time T because the second entity has begun its service. At time T , the statistic is 0 before it becomes 1 again. The value of 0 corresponds to the server's empty state after the first entity has departed and before the second entity has arrived. Like this empty state, the value of 0 does not persist for a positive duration.

Scenario: Queue Length

Another example of zero-duration values is in “Plotting the Queue-Length Signal”, which discusses a signal that indicates the length of a queue. At time 3, the queue length increases by 1 because a new entity arrives. Subsequently but still at time 3, the queue length decreases by 1 because an entity advances from the queue to the server. That is, the larger value at time 3 does not persist for a positive duration.

Importance of Zero-Duration Values

The values of signals, even values that do not persist for a positive duration, can help you understand or debug your simulations. In the example described in “Scenario: Server Departure and New Arrival” on page 4-19, the zero-duration value of 0 in the signal tells you that the server experienced a departure. If the signal assumed only the value 1 at time T (because 1 is the final value at time T), then the constant values before, at, and after time T would fail to indicate the departure. While you could use a departure count signal to detect departures specifically, the zero-duration value in the number-in-block signal provides you with more information in a single signal.

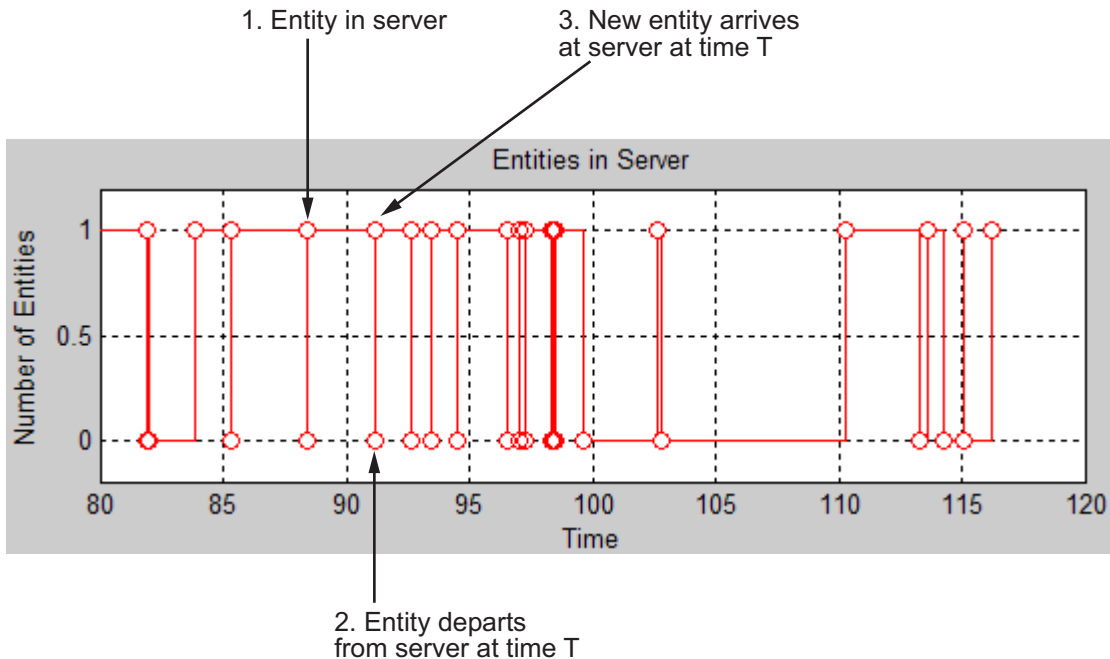
Detecting Zero-Duration Values

These topics describe ways to detect and examine zero-duration values:

- “Plotting Signals that Exhibit Zero-Duration Values” on page 4-20
- “Plotting the Number of Signal Changes Per Time Instant” on page 4-21
- “Viewing Zero-Duration Values in the MATLAB Workspace” on page 4-22

Plotting Signals that Exhibit Zero-Duration Values

One way to visualize event-based signals, including signal values that do not persist for a positive duration, is to use the Signal Scope or X-Y Signal Scope block. Either of these blocks can produce a plot that includes a marker for each signal value (or each signal-based event, in the case of the event counting scope). For example, the figure below uses a plot to illustrate the situation described in “Scenario: Server Departure and New Arrival” on page 4-19.



When multiple plotting markers occur along the same vertical line, it means that the signal assumes multiple values at a single time instant. The callouts in the figure describe the server states that correspond to a few key points of the plot.

Note Unlike the Signal Scope block and X-Y Signal Scope blocks, the Scope block in the Simulink Sinks library does not detect zero-duration values. For more information, see “Comparison with Time-Based Plotting Tools” on page 11-17.

Plotting the Number of Signal Changes Per Time Instant

To detect the presence of zero-duration values, but not the values themselves, use the Instantaneous Event Counting Scope block with the **Type of change in signal value** parameter set to **Either**. When the input signal assumes

multiple values at an instant of time, the plot shows a stem of height of two or greater.

For an example using this block, see “Example: Plotting Event Counts to Check for Simultaneity” on page 11-15.

Viewing Zero-Duration Values in the MATLAB Workspace

If an event-based signal assumes many values at one time instant and you cannot guess the sequence from a plot of the signal versus time, then you can get more information by examining the signal in the MATLAB workspace. By creating a variable that contains each time and signal value, you can recover the exact sequence in which the signal assumed each value during the simulation.

See “Sending Data to the MATLAB Workspace” on page 4-15 for instructions and an example.

Modeling Queues and Servers

The topics below supplement the discussion in “Basic Queues and Servers” in the SimEvents getting started documentation.

- “Using a LIFO Queuing Discipline” on page 5-2
- “Sorting by Priority” on page 5-5
- “Preempting an Entity in a Server” on page 5-11
- “Determining Whether a Queue Is Nonempty” on page 5-17
- “Modeling Multiple Servers” on page 5-18
- “Modeling the Failure of a Server” on page 5-20

Using a LIFO Queuing Discipline

In this section...
“Overview of LIFO Queues” on page 5-2
“Example: Waiting Time in LIFO Queue” on page 5-2

Overview of LIFO Queues

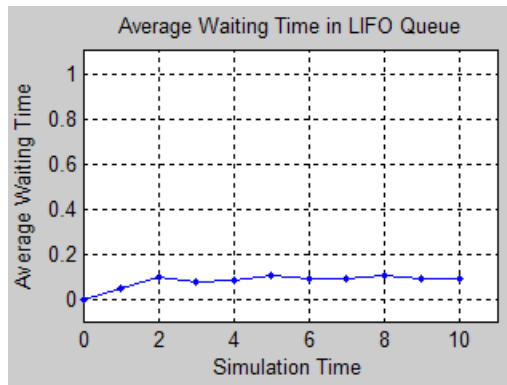
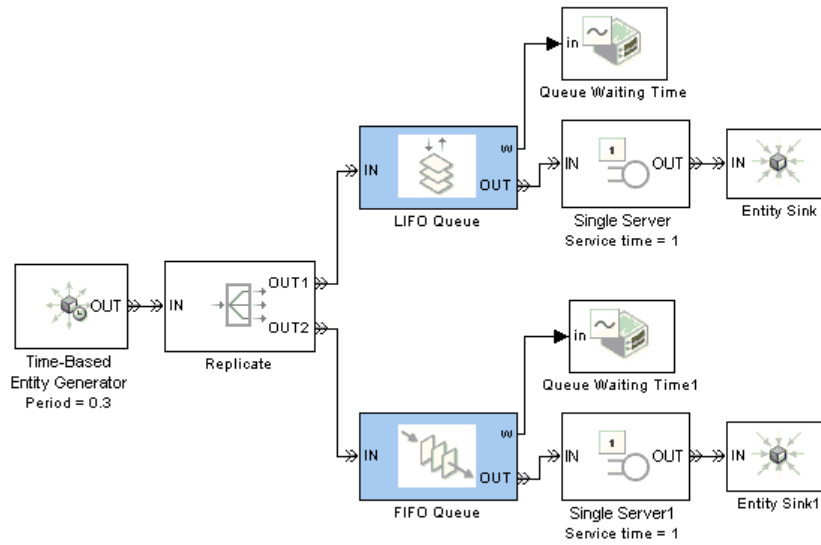
The LIFO Queue block supports the last-in, first-out (LIFO) queuing discipline. The entity that departs from the queue at a given time is the most recent arrival. You can interpret a LIFO queue as a stack.

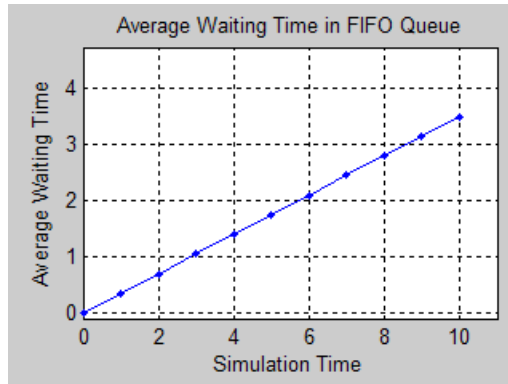
Some ways to see the difference between FIFO and LIFO queuing disciplines are to

- Attach data to entities to distinguish entities from each other. For more information about using entities to carry data, see “Setting Attributes of Entities” on page 1-7.
- View simulation statistics that you expect the queuing discipline to influence. One such statistic is the average waiting time in the queue; to compute the waiting time of each entity, the block must know which entity is departing at a given departure time.

Example: Waiting Time in LIFO Queue

As an example, compare the FIFO and LIFO disciplines in a D/D/1 queuing system with an intergeneration time of 0.3 and a service time of 1.





Sorting by Priority

In this section...
“Behavior of the Priority Queue Block” on page 5-5
“Example: FIFO and LIFO as Special Cases of a Priority Queue” on page 5-5
“Example: Serving Preferred Customers First” on page 5-8

Behavior of the Priority Queue Block

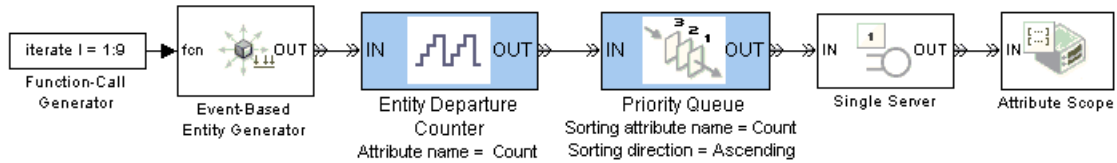
The Priority Queue block supports queuing in which entities' positions in the queue are based primarily on their attribute values. Arrival times are relevant only when attribute values are equal. You specify the attribute and the sorting direction using the **Sorting attribute name** and **Sorting direction** parameters in the block's dialog box. To assign values of the attribute for each entity, you can use the Set Attribute block as described in “Setting Attributes of Entities” on page 1-7.

Note While you can view the value of the sorting attribute as an entity priority, this value has nothing to do with event priorities or block priorities.

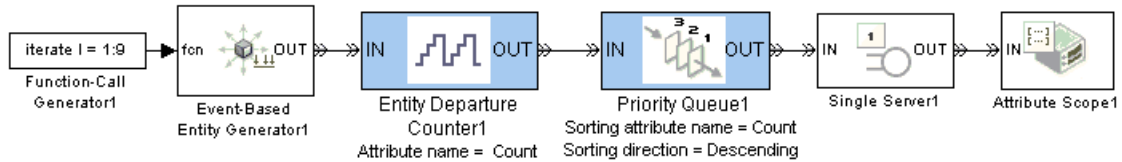
Example: FIFO and LIFO as Special Cases of a Priority Queue

Two familiar cases are shown in the example below, in which a priority queue acts like a FIFO or LIFO queue. At the start of the simulation, the FIFO and LIFO sections of the model each generate nine entities, the first of which advances immediately to a server. The remaining entities stay in the queues until the server becomes available. The sorting attribute is **Count**, whose values are the entities' arrival sequence at the queue block. In this example, the servers do not permit preemption; preemptive servers would behave differently.

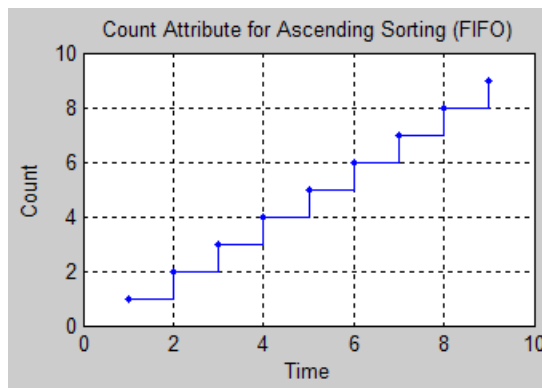
Priority Queue acts like FIFO Queue

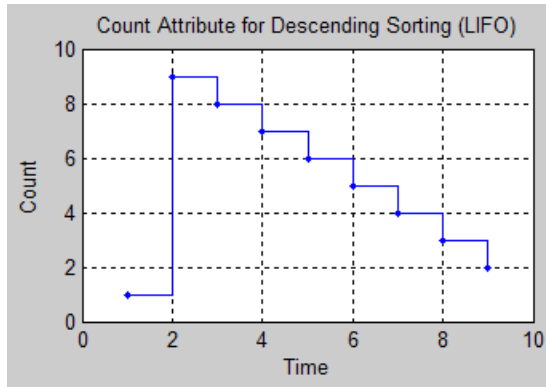


Priority Queue acts like LIFO Queue



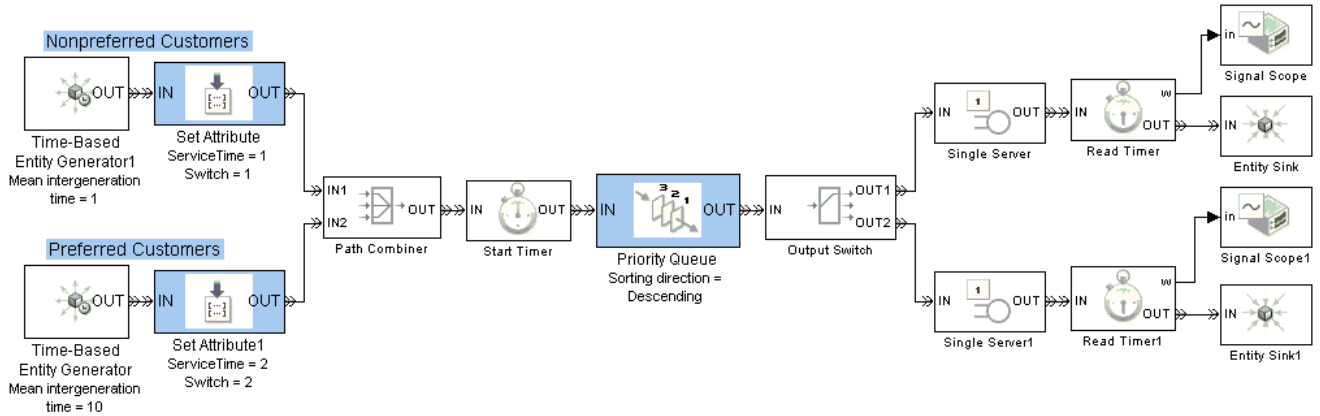
The FIFO plot reflects an increasing sequence of Count values. The LIFO plot reflects a descending sequence of Count values, except for the Count=1 entity that advances to the server before the queue has any other entities to sort.



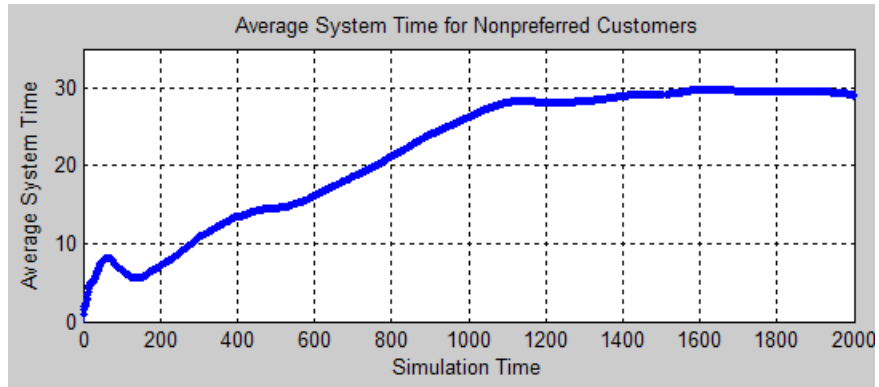


Example: Serving Preferred Customers First

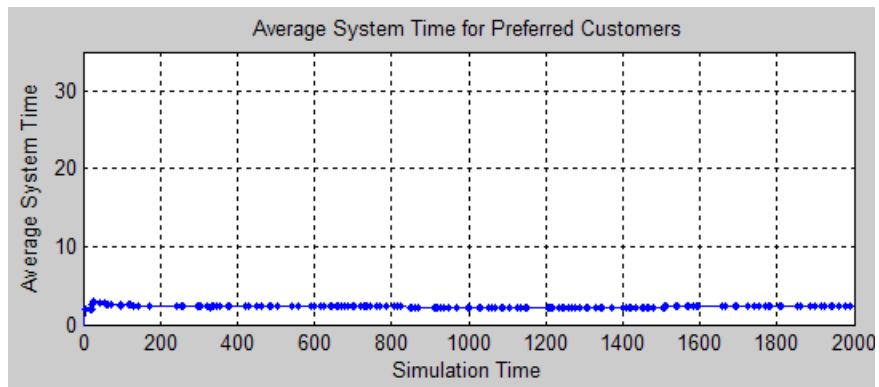
In the example below, two types of customers enter a queuing system. One type, considered to be preferred customers, are less common but require longer service. The priority queue places preferred customers ahead of nonpreferred customers. The model plots the average system time for the set of preferred customers and separately for the set of nonpreferred customers.



You can see from the plots that despite the shorter service time, the average system time for the nonpreferred customers is much longer than the average system time for the preferred customers.



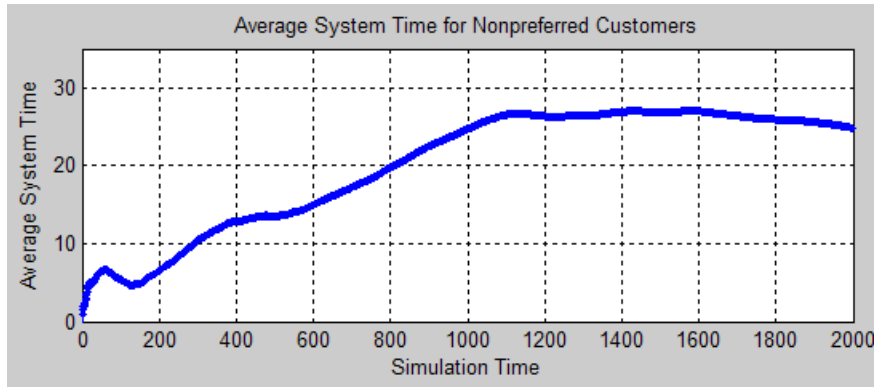
Average System Time for Nonpreferred Customers Sorted by Priority



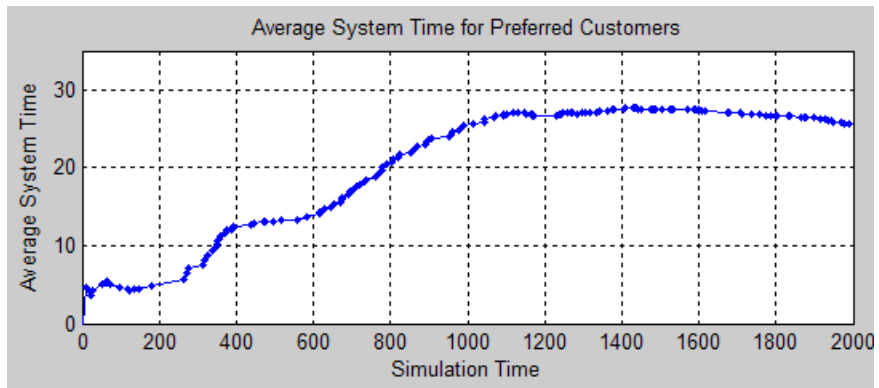
Average System Time for Preferred Customers Sorted by Priority

Comparison with Unsorted Behavior

If the queue used a FIFO discipline for all customers instead of a priority sorting, then the average system time would decrease slightly for the nonpreferred customers and increase markedly for the preferred customers.



Average System Time for Nonpreferred Customers Unsorted



Average System Time for Preferred Customers Unsorted

Preempting an Entity in a Server

In this section...

“Definition of Preemption” on page 5-11

“Criteria for Preemption” on page 5-11

“Residual Service Time” on page 5-12

“Queuing Disciplines for Preemptive Servers” on page 5-12

“Example: Preemption by High-Priority Entities” on page 5-12

Definition of Preemption

Preemption from a server is the replacement of an entity in the server by a different entity that satisfies certain criteria. The Single Server block supports preemption. The preempted entity immediately departs from the block through the **P** entity output port instead of through the usual **OUT** port.

Criteria for Preemption

Whether preemption occurs depends on attribute values of the entity in the server and of the entity attempting to arrive at the server. You specify the attribute using the **Sorting attribute name** parameter in the Single Server block’s dialog box. You use the **Sorting direction** parameter to indicate whether the preempting entity has a smaller (Ascending) or larger (Descending) value of the attribute, compared to the entity being replaced. (Both parameters are available after you select **Permit preemption based on attribute**.) To assign values of the sorting attribute for each entity, you can use the Set Attribute block, as described in “Setting Attributes of Entities” on page 1-7. Valid values for the sorting attribute are any real numbers, Inf, and -Inf.

If the attribute values are equal, no preemption occurs.

When preemption is supposed to occur, the **P** port must not be blocked. Consider connecting the **P** port to a queue or server with infinite capacity, to prevent a blockage during the simulation.

Note You can interpret the value of the sorting attribute as an entity priority. However, this value has nothing to do with event priorities or block priorities.

Residual Service Time

A preempted entity might or might not have completed its service time. The remaining service time the entity would have required if it had not been preempted is called the entity's *residual* service time. If you select **Write residual service time to attribute** in the Single Server block, then the block records the residual service time of each preempted entity in an attribute of that entity. If the entity completes its service time before preemption occurs, then the residual service time is zero.

For entities that depart from the block's **OUT** entity output port (that is, entities that are not preempted), the block records a residual service time only if the entity already has an attribute whose name matches the **Residual service time attribute name** parameter value. In this case, the block sets that attribute to zero when the entity departs from the **OUT** port.

Queuing Disciplines for Preemptive Servers

When you permit preemption in a Single Server block preceded by a queue, only the entity at the head of the queue can preempt an entity in the server.

The Priority Queue block is particularly appropriate for use with the preemption feature of the Single Server block. When an entity with sufficiently high priority arrives at the Priority Queue block, the entity goes to the head of the queue and immediately advances to the server.

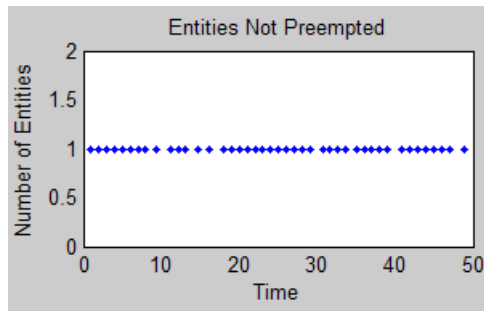
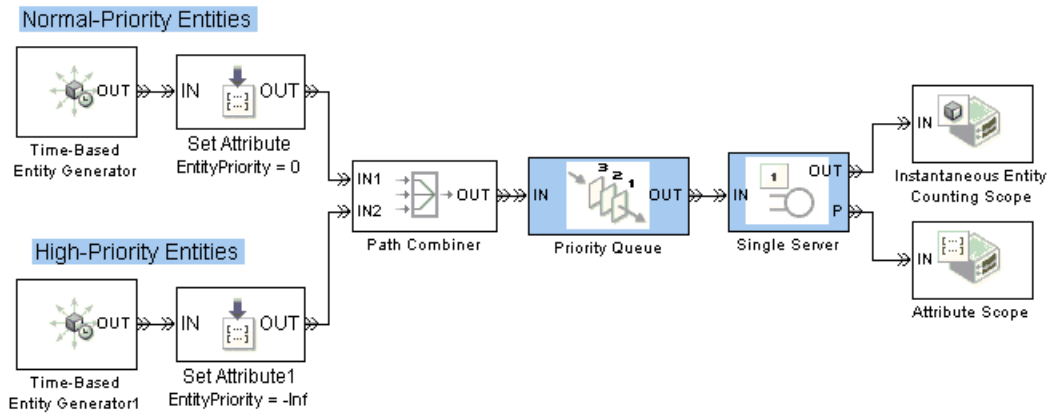
When using the Single Server and Priority Queue blocks together, you typically set the **Sorting attribute name** and **Sorting direction** parameters to the same values in both blocks.

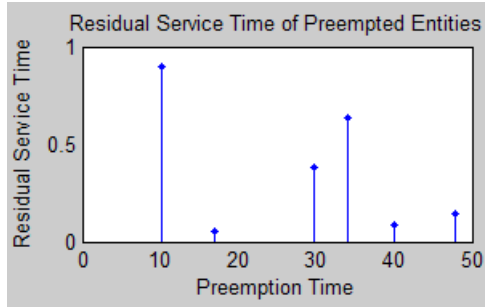
Example: Preemption by High-Priority Entities

The following example generates two classes of entities, most with an `EntityPriority` attribute value of 0 and some with an `EntityPriority` attribute value of `-Inf`. The sorting direction in the Priority Queue and Single

Server blocks is Ascending, so entities with sorting attribute values of $-\infty$ go to the head of the priority queue and immediately preempt any entity in the server except another entity whose sorting attribute value is $-\infty$.

One plot shows when nonpreemptive departures occur, while another plot indicates the residual service time whenever preemptive departures occur.





Appearance of Preemption-Related Operations in Debugger

To see how operations related to preemption appear in the SimEvents debugger, first zoom in on the plot of residual service time to find approximate times when preemptions occur. For example, the second preemption occurs shortly after $T=17$. Then, use the debugger:

- 1 Begin a debugger session for the example model. At the MATLAB command prompt, enter:

```
simeventsdocex('doc_preemptiveserver');
sedebug('doc_preemptiveserver')
```

- 2 Proceed in the simulation. At the `sedebug>>` prompt, enter:

```
tbreak 17
cont
```

The partial output indicates that an event is about to execute shortly after $T=17$:

```
Hit b1 : Breakpoint for first operation at or after time 17.000000

%=====
Executing EntityGeneration Event (ev33)           Time = 17.043502632805254
: Entity = <none>                                Priority = 300
: Block = Time-Based Entity Generator1
```

The event is the generation of the entity that preempts an entity in the server, but you cannot see that level of detail yet.

- 3** Proceed in the simulation to see what happens as a result of the event execution:

step over

The output shows that a new entity with identifier en19 advances to the head (position 1) of the priority queue and preempts the entity in the server.

```
%.....%
Generating Entity (en19)
: Block = Time-Based Entity Generator1
%.....%
Entity Advancing (en19)
: From = Time-Based Entity Generator1
: To   = Set Attribute1
%.....%
Setting Attribute on Entity (en19)
: EntityPriority = -Inf
: Block = Set Attribute1
%.....%
Entity Advancing (en19)
: From = Set Attribute1
: To   = Path Combiner
%.....%
Entity Advancing (en19)
: From = Path Combiner
: To   = Priority Queue
%.....%
Queuing Entity (en19)
: Priority Pos = 1 of 3
: Capacity = 25
: Block = Priority Queue
%.....%
Entity Advancing (en19)
: From = Priority Queue
: To   = Single Server
%.....%
Preempting Entity (en16)
: NewEntity = en19 (EntityPriority = -Inf)
: OldEntity = en16 (EntityPriority = 0)
: Block = Single Server
```

Further output reflects the effect on the preempted entity: its service completion event no longer applies, it carries the residual service time in an attribute, and it advances to the block that connects to the **P** port of the server.

```
%.....%
Canceling ServiceCompletion Event (ev49)
: EventTime = 17.094640781246184
: Priority   = 500
: Entity    = en16
: Block     = Single Server
%.....%
Setting Attribute on Entity (en16)
: ResidualServiceTime = 0.0511381484409306
: Block = Single Server
%.....%
Entity Advancing (en16)
: From = Single Server
: To   = Attribute Scope
%.....%
Destroying Entity (en16)
: Block = Attribute Scope
```

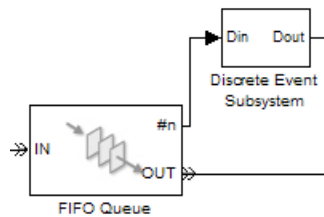
4 End the debugger session. At the `sedbug>>` prompt, enter:

```
sedb.quit
```

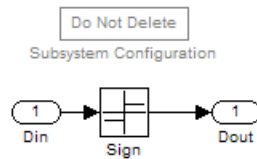
Determining Whether a Queue Is Nonempty

To determine whether a queue is storing any entities, use this technique:

- 1 Enable the **#n** output signal from the queue block. In the block dialog box, on the **Statistics** tab, set **Number of entities in queue** to On.
- 2 Connect the **#n** signal to a Discrete Event Subsystem block.



- 3 Open the subsystem by double-clicking the Discrete Event Subsystem block.
- 4 From the Math Operations library in the Simulink library set, insert a Sign block into the subsystem window. Connect the input port to the Din block and connect the output port to the Dout block.



As a result, the output from the subsystem has values of 0 and 1. A value of 1 indicates that the queue is storing one or more entities. A value of 0 indicates that the queue is not storing any entities.

Modeling Multiple Servers

In this section...

“Blocks that Model Multiple Servers” on page 5-18

“Example: M/M/5 Queuing System” on page 5-18

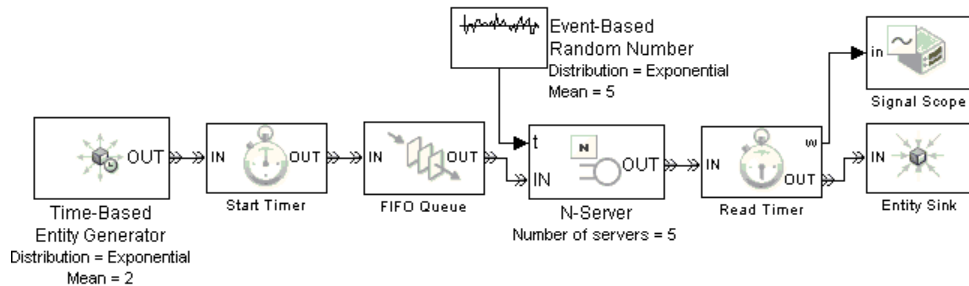
Blocks that Model Multiple Servers

You can use the N-Server and Infinite Server blocks to model a bank of identical servers operating in parallel. The N-Server block lets you specify the number of servers using a parameter, while the Infinite Server block models a bank of infinitely many servers.

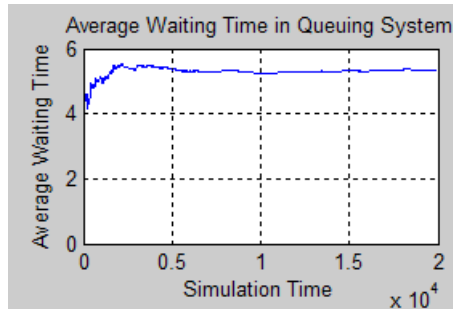
To model multiple servers that are not identical to each other, you must use multiple blocks. For example, to model a pair of servers whose service times do not share the same distribution, use a pair of Single Server blocks rather than a single N-Server block. The example in “Example: Selecting the First Available Server” in the SimEvents getting started documentation illustrates the use of multiple Single Server blocks with a switch.

Example: M/M/5 Queuing System

The example below shows a system with infinite storage capacity and five identical servers. In the notation, the M stands for Markovian; M/M/5 means that the system has exponentially distributed interarrival and service times, and five servers.



The plot below shows the waiting time in the queuing system.



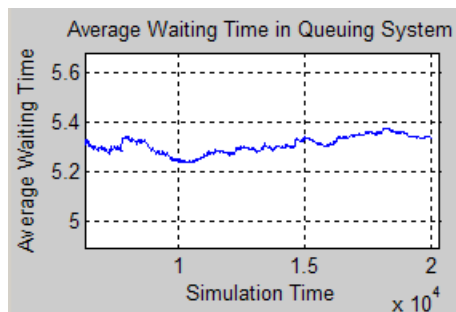
You can compare the empirical values shown in the plot with the theoretical value, $E[S]$, of the mean system time for an $M/M/m$ queuing system with an arrival rate of $\lambda=1/2$ and a service rate of $\mu=1/5$. Using expressions in [2], the computation is as follows.

$$\rho = \frac{\lambda}{m\mu} = \frac{(1/2)}{5(1/5)} = \frac{1}{2}$$

$$\pi_0 = \left[1 + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho} \right]^{-1} \approx 0.0801$$

$$E[S] = \frac{1}{\mu} + \frac{1}{\mu} \frac{(m\rho)^m}{m!} \frac{\pi_0}{m(1-\rho)^2} \approx 5.26$$

Zooming in the plot shows that the empirical value is close to 5.26.



Modeling the Failure of a Server

In this section...

“Server States” on page 5-20

“Using a Gate to Implement a Failure State” on page 5-20

“Using Stateflow Charts to Implement a Failure State” on page 5-21

Server States

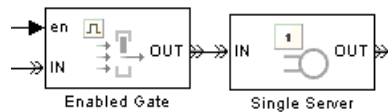
In some applications, it is useful to model situations in which a server fails. For example, a machine might break down and later be repaired, or a network connection might fail and later be restored. This section explores ways to model failure of a server, as well as server states.

The blocks in the Servers library do not have built-in states, so you can design states in any way that is appropriate for your application. Some examples of possible server states are in the table below.

Server as Communication Channel	Server as Machine	Server as Human Processor
Transmitting message	Processing part	Working
Connected but idle	Waiting for new part to arrive	Waiting for work
Unconnected	Off	Off duty
Holding message (pending availability of destination)	Holding part (pending availability of next operator)	Waiting for resource
Establishing connection	Warming up	Preparing to begin work

Using a Gate to Implement a Failure State

For any state that represents a server’s inability or refusal to accept entity arrivals even though the server is not necessarily full, a common implementation involves an Enabled Gate block preceding the server.



The gate prevents entity access to the server whenever the gate’s control signal at the **en** input port is zero or negative. The logic that creates the **en** signal determines whether or not the server is in a failure state. You can implement such logic using the techniques described in Chapter 7, “Using Logic” or using Stateflow charts to transition among a finite number of server states.

For an example in which an Enabled Gate block precedes a server, see “Example: Controlling Joint Availability of Two Servers” on page 8-4. The example is not specifically about a failure state, but the idea of controlling access to a server is similar. Also, you can interpret the Signal Latch block with the **st** output signal enabled as a two-state machine that changes state when read and write events occur.

Note A gate prevents new entities from arriving at the server but does not prevent the current entity from completing its service. If you want to eject the current entity from the server upon a failure occurrence, then you can use the preemption feature of the server to replace the current entity with a high-priority “placeholder” entity.

Using Stateflow Charts to Implement a Failure State

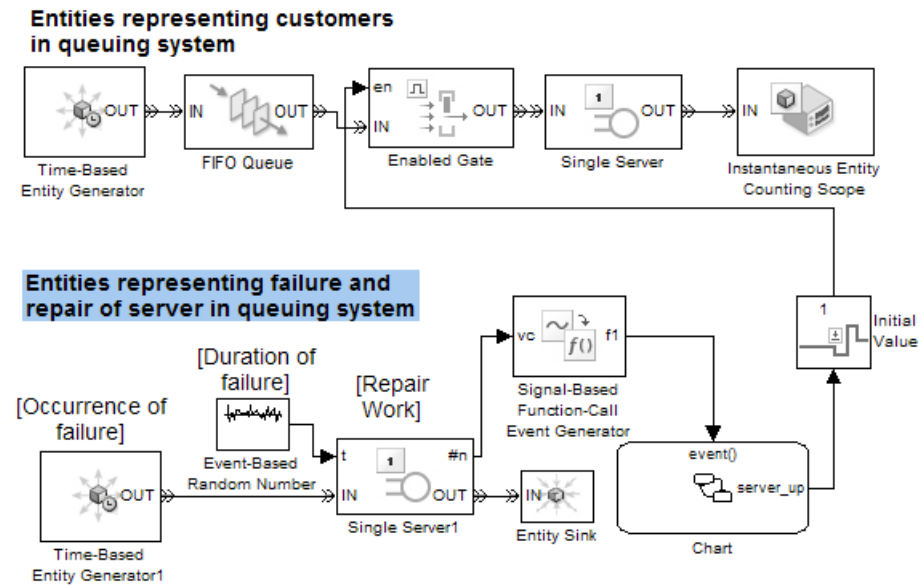
Stateflow software is suitable for implementing transitions among a finite number of server states. If you need to support more than just two states, then a Stateflow block might be more natural than a combination of Enabled Gate and logic blocks.

When modeling interactions between the state chart and discrete-event aspects of the model, note that a function call is the recommended way to make Stateflow blocks respond to asynchronous state changes. You can use blocks in the Event Generators and Event Translation libraries to produce a function call upon signal-based events or entity departures; the function call can invoke a Stateflow block. Conversely, a Stateflow block can output a

function call that can cause a gate to open, an entity counter to reset, or an entity generator to generate a new entity.

Example: Failure and Repair of a Server

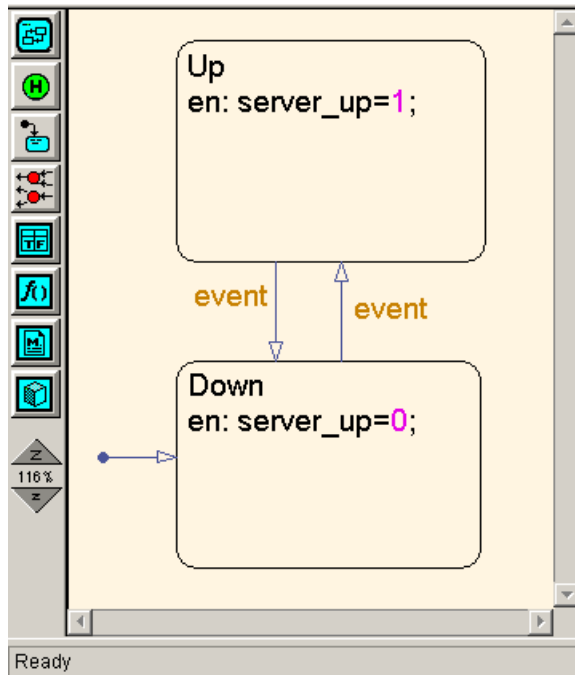
The example below uses a Stateflow block to describe a two-state machine. A server is either down (failed) or up (operable). The state of the server is an output signal from the Stateflow block and is used to create the enabling signal for an Enabled Gate block that precedes a server in a queuing system.



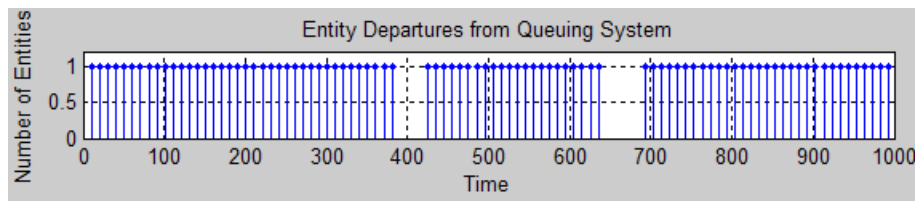
The lower portion of the model contains a parallel queuing system. The entities in the lower queuing system represent failures, not customers. Generation of a failure entity represents a failure occurrence in the upper queuing system. Service of a failure entity represents the time during which the server in the upper queuing system is down. Completion of service of a failure entity represents a return to operability of the upper queuing system.

When the lower queuing system generates an entity, changes in its server's #n signal invoke the Stateflow block that determines the state of the upper

queuing system. Increases in the **#n** signal cause the server to go down, while decreases cause the server to become operable again.



While this simulation runs, Stateflow alternately highlights the up and down states. The plot showing entity departures from the upper queuing system shows gaps, during which the server is down.

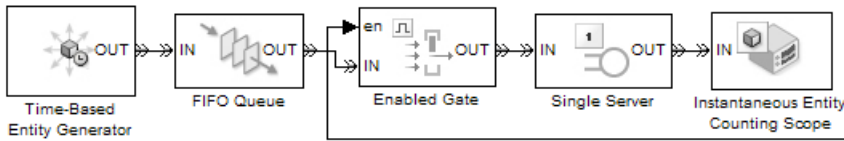


Although this two-state machine could be modeled more concisely with a Signal Latch block instead of a Stateflow block, the Stateflow chart scales more easily to include additional states or other complexity.

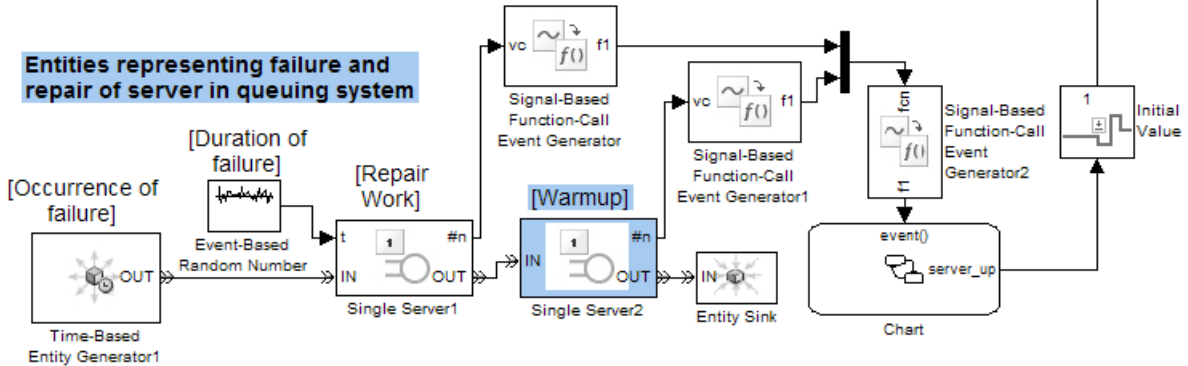
Example: Adding a Warmup Phase

The example below modifies the one in “Example: Failure and Repair of a Server” on page 5-22 by adding a warmup phase after the repair is complete. The Enabled Gate block in the upper queuing system does not open until the repair and the warmup phase are complete. In the lower queuing system, an additional Single Server block represents the duration of the warmup phase.

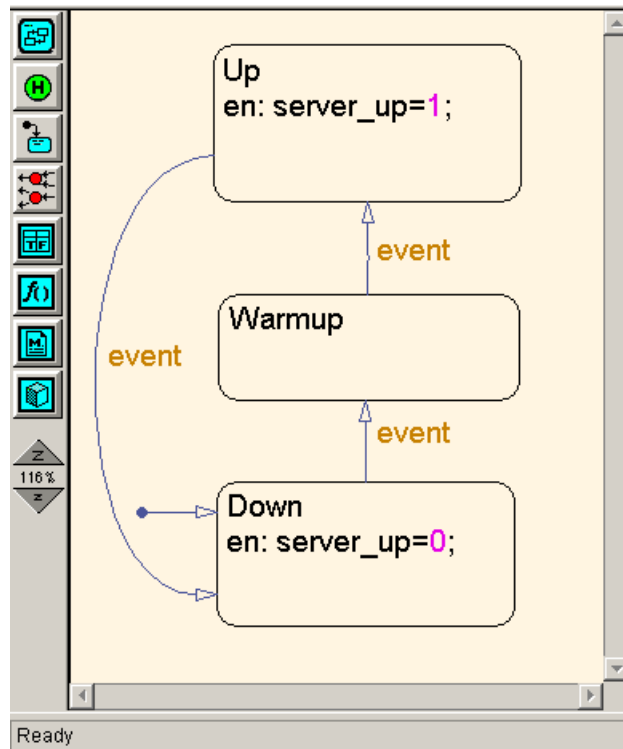
Entities representing customers in queuing system



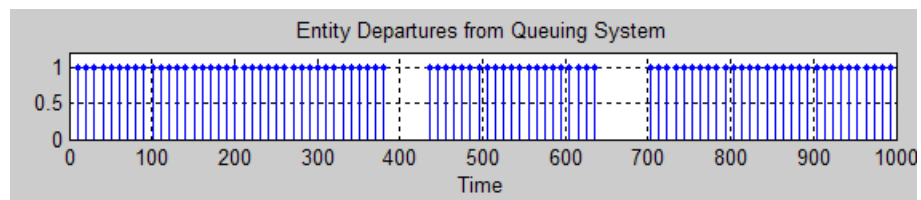
Entities representing failure and repair of server in queuing system



In the Stateflow block, the input function calls controls when the repair operation starts, when it ends, and when the warmup is complete. The result of the function-call event depends on the state of the chart when the event occurs. A rising edge of the Repair Work block’s #n signal starts the repair operation, a falling edge of the same signal ends the repair operation, and a falling edge of the Warmup block’s #n signal completes the warmup.



While this simulation runs, the Stateflow chart alternates among the three states. The plot showing entity departures from the upper queuing system shows gaps, during which the server is either under repair or warming up. By comparing the plot to the one in “Example: Failure and Repair of a Server” on page 5-22, you can see that the gaps in the server’s operation last slightly longer. This is because of the warmup phase.



Routing Techniques

The topics below supplement the discussion in “Designing Paths for Entities” in the SimEvents getting started documentation.

- “Output Switching Based on a Signal” on page 6-2
- “Example: Cascaded Switches with Skewed Distribution” on page 6-6
- “Example: Compound Switching Logic” on page 6-7

Output Switching Based on a Signal

In this section...

“Specifying an Initial Port Selection” on page 6-2

“Using the Storage Option to Prevent Latency Problems” on page 6-2

Specifying an Initial Port Selection

When the Output Switch block uses an input signal **p**, the block might attempt to use the **p** signal before its first sample time hit. If the initial value of the **p** signal is out of range (for example, zero) or is not your desired initial port selection for the switch, then you should specify the initial port selection in the Output Switch block’s dialog box. Use this procedure:

- 1 Select **Specify initial port selection**.
- 2 Set **Initial port selection** to the desired initial port selection. The value must be an integer between 1 and **Number of entity output ports**. The Output Switch block uses **Initial port selection** instead of the **p** signal’s value until the signal has its first sample time hit.

Tip A common scenario in which you should specify the initial port selection is when the **p** signal is an event-based signal in a feedback loop. The first entity is likely to arrive at the switch before the **p** signal has its first sample time hit. See “Example: Choosing the Shortest Queue” on page 7-3 for an example of this scenario.

Using the Storage Option to Prevent Latency Problems

When the Output Switch block uses an input signal **p**, the block must successfully coordinate its entity-handling operations with the operations of whichever block produces the **p** signal. For example, if **p** is an event-based signal that can change at the same time when an entity arrives, the simulation behavior depends on whether the block reacts to the signal update before or after the arrival.

Coordination that is inappropriate for the model can cause the block to use a value of **p** from a previous time. You can prevent a systemic latency problem by using the **Store entity before switching** option.

Effect of Enabling Storage

If you select **Store entity before switching** in the Output Switch block, then the block becomes capable of storing one entity at a time. Furthermore, the block decouples its arrival and departure processing to give other blocks along the entity's path an opportunity to complete their processing. Completing their processing is important if, for example, it affects the **p** signal of the Output Switch block.

If an entity arrives and the storage location is empty, then the block does the following:

- 1** Stores the arriving entity and schedules a storage completion event on the event calendar.
- 2** Yields control to blocks in the model that perform operations that are either not scheduled on the event calendar, or prioritized ahead of the storage completion event on the event calendar. For example, this might give other blocks a chance to update the signal that connects to the **p** port.
- 3** Executes the storage completion event.
- 4** Determines which entity output port is the selected port.
- 5** If the selected port is not blocked, the stored entity departs immediately.

If the selected port is blocked, the stored entity departs when one of these occurs:

- The selected port becomes unblocked.
- The selection changes to a port that is not blocked.
- The stored entity times out. For details on timeouts, see Chapter 9, "Forcing Departures Using Timeouts".

Storage for a Time Interval. A stored entity can stay in the block for a nonzero period of time if the selected port is blocked. The design of your model should account for the effect of this phenomenon on statistics or other simulation behaviors. For an example scenario, see the discussion of average wait in “Example Without Storage” on page 6-5.

Even if the stored entity departs at the same time that it arrives, step 2 is important for preventing latency.

Example Using Storage. The model in “Example: Choosing the Shortest Queue” on page 7-3 uses the **Store entity before switching** option in the Output Switch block. Suppose the queues have sufficient storage capacity so that the Output Switch block never stores an entity for a nonzero period of time. When an entity arrives at the Output Switch block, it does the following:

- 1 Stores the entity and schedules a storage completion event on the event calendar.
- 2 Yields control to other blocks so that the Time-Based Entity Generator and Discrete Event Subsystem blocks can update their output signals in turn.
- 3 Executes the storage completion event.
- 4 Possibly detects a change in the **p** signal as a result of the Discrete Event Subsystem block’s computation, and reacts accordingly by selecting the appropriate entity output port.
- 5 Outputs the entity using the up-to-date value of the **p** signal.

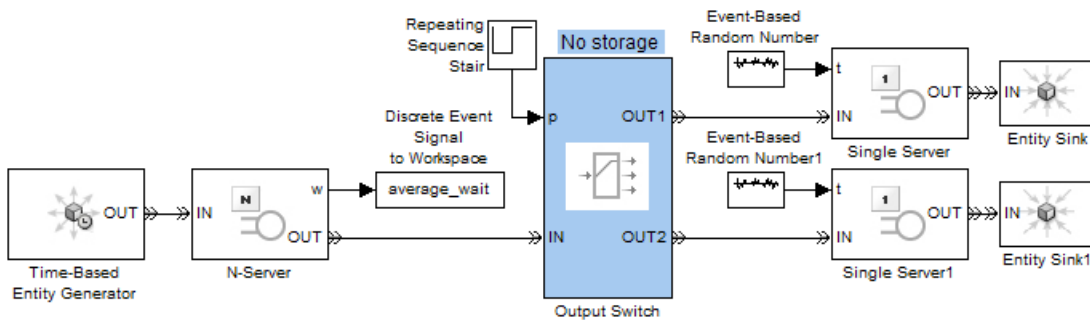
Effect of Disabling Storage

If you do not select **Store entity before switching** in the Output Switch block, then the block processes an arrival and departure as an atomic operation. The block assumes that the **p** signal is already up to date at the given time.

For common problems and troubleshooting tips, see “Unexpected Use of Old Value of Signal” on page 14-76 in the SimEvents user guide documentation.

Example Without Storage. The model below does not use the **Store entity before switching** option in the Output Switch block. Storage in the switch is unnecessary here because the application processes service completion events after the Repeating Sequence Stair block has already updated its output signal at the given time.

Tip It is not always easy to determine whether storage is unnecessary in a given model. If you are not sure, you should select **Store entity before switching**.



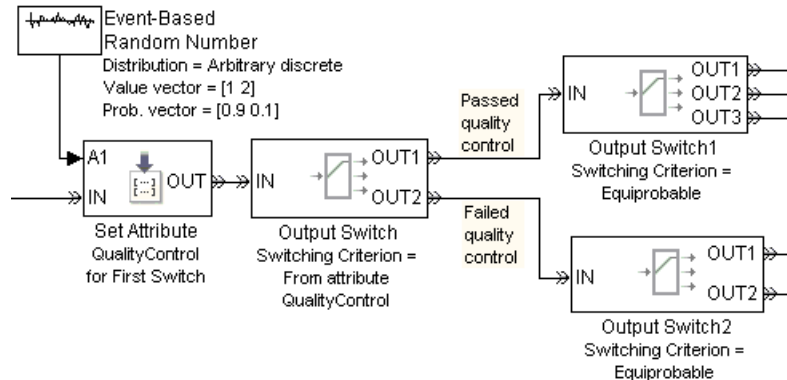
Furthermore, storage in the switch is probably undesirable in this model. Storing entities in the Output Switch block for a nonzero period of time would affect the computation of average wait, which is the N-Server block's **w** output signal. If the goal is to compute the average waiting time of entities that have not yet reached the Single Server blocks, then the model would need to account for entities stored in the switch for a nonzero period of time.

Example: Cascaded Switches with Skewed Distribution

Suppose entities represent manufactured items that undergo a quality control process followed by a packaging process. Items that pass the quality control test proceed to one of three packaging stations, while items that fail the quality control test proceed to one of two rework stations. You can model the decision making using these switches:

- An Output Switch block that routes items based on an attribute that stores the results of the quality control test
- An Output Switch block that routes passing-quality items to the packaging stations
- An Output Switch block that routes failing-quality items to the rework stations

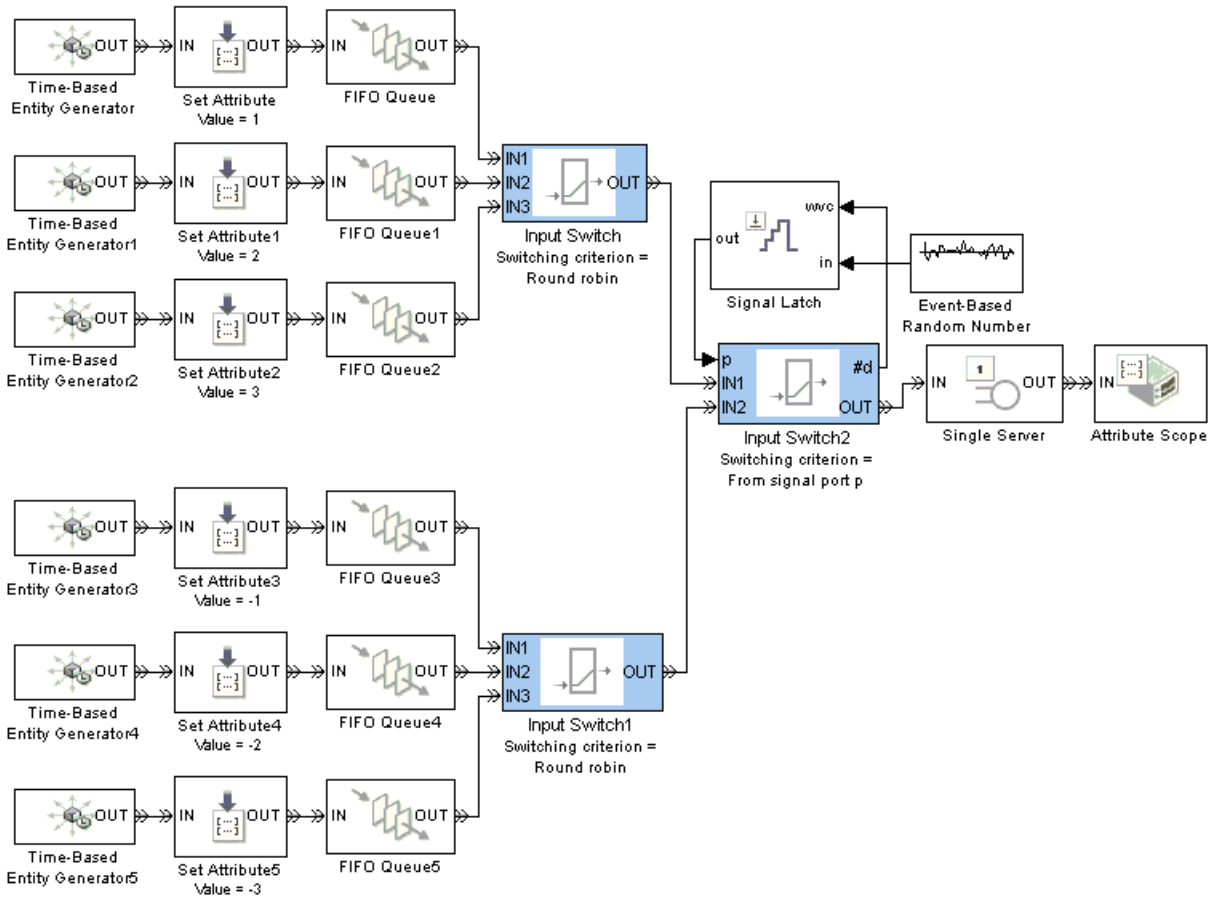
The figure below illustrates the switches and their switching criteria.



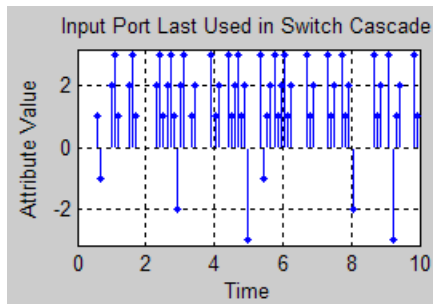
Example: Compound Switching Logic

Suppose a single server processes entities from two groups each consisting of three sources. The switching component between the entity sources and the server determines which entities proceed to the server whenever it is available. The switching component uses a distribution that is skewed toward entities from the first group. Within each group, the switching component uses a round-robin approach.

The example below shows how to implement this design using three Input Switch blocks. The first two Input Switch blocks have their **Switching criterion** parameter set to Round robin to represent the processing of entities within each group of entity sources. The last Input Switch block uses a random signal with a skewed probability distribution to choose between the two groups. The Signal Latch block causes the random number generator to draw a new random number after each departure from the last Input Switch block.



For tracking purposes, the model assigns an attribute to each entity based on its source. The attribute values are 1, 2, and 3 for entities in the first group and -1, -2, and -3 for entities in the second group. You can see from the plot below that negative values occur less frequently than positive values, reflecting the skewed probability distribution. You can also see that the positive values reflect a round-robin approach among servers in the top group, while negative values reflect a round-robin approach among servers in the bottom group.



Using Logic

- “Role of Logic in SimEvents Models” on page 7-2
- “Using Embedded MATLAB Function Blocks for Logic” on page 7-3
- “Using Logic Blocks” on page 7-10

Role of Logic in SimEvents Models

Logic can be an important component in a discrete-event simulation, for specifying

- Normal but potentially complex routing or gating behavior.

For example, you might want to model a multiple-queue system in which entities advance to the shortest queue. Such a model must also indicate what happens if the minimum length is not unique.

- Handling of overflows, blockages, and other special cases.

For example, a communication system might drop packets that overflow a queue, while a manufacturing assembly line might pause processing at one machine if it releases parts that overflow a second machine.

Using Embedded MATLAB Function Blocks for Logic

In this section...

“Overview of Use of Embedded MATLAB Function Blocks” on page 7-3

“Example: Choosing the Shortest Queue” on page 7-3

“Example: Varying Fluid Flow Rate Based on Batching Logic” on page 7-6

Overview of Use of Embedded MATLAB Function Blocks

If your logic algorithm is easier to express in code than in a block diagram, then you can use the Embedded MATLAB Function block to implement the logic. Details about how to use this block are in “Using the Embedded MATLAB Function Block” in the Simulink documentation.

Here are some tips specific to SimEvents models:

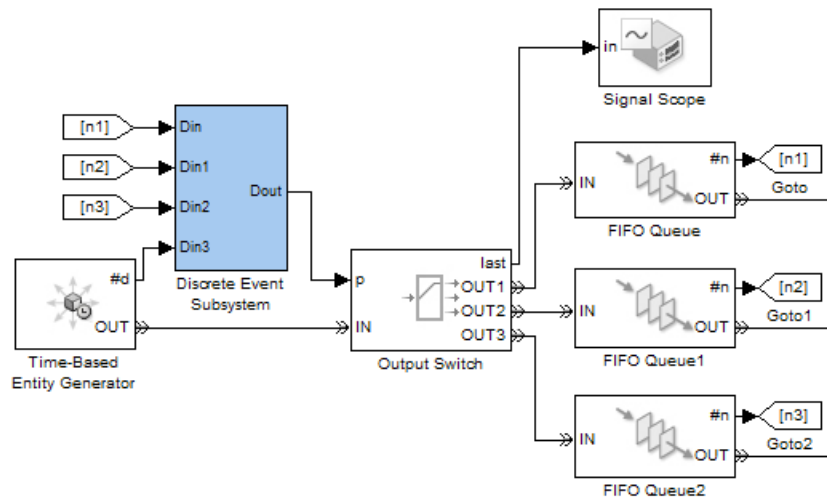
- If your logic algorithm requires data from an earlier call to the function, then you can use persistent variables to retain data between calls. For examples of this technique in SimEvents models, see
 - The Switching Logic subsystems in the Astable Multivibrator Circuit demo.
 - “Example: Computing a Time Average of a Signal” on page 12-11
- If you put an Embedded MATLAB Function block in a Discrete Event Subsystem block, use the Ports and Data Manager instead of Model Explorer to view or change properties such as the size or source of an argument. Model Explorer does not show the contents of Discrete Event Subsystem blocks.

Example: Choosing the Shortest Queue

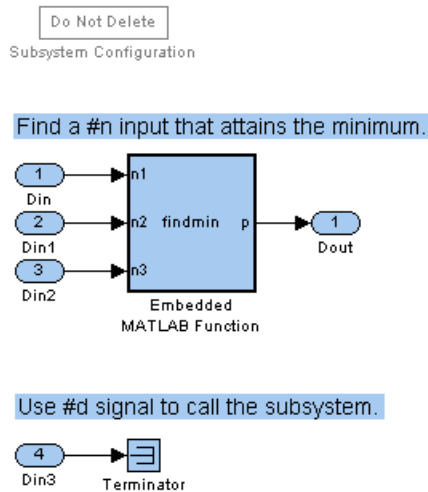
The model below directs entities to the shortest of three queues. It uses an Output Switch block to create the paths to the different queues. To implement the choice of the shortest queue, a discrete event subsystem queries each queue for its current length, determines which queue or queues achieve the minimum length, and provides that information to the Output Switch block. To ensure that the information is up to date when the Output Switch

block attempts to output the arriving entity, the block uses the **Store entity before switching** option; for details, see “Using the Storage Option to Prevent Latency Problems” on page 6-2.

For simplicity, the model omits any further processing of the entities after they leave their respective queues.



Although the block diagram shows signals at the $\#n$ signal output ports from the queue blocks and another signal at the p signal input port of the Output Switch block, the block diagram does not indicate how to compute p from the set of $\#n$ values. That computation is performed inside a discrete event subsystem that contains an Embedded MATLAB Function block.



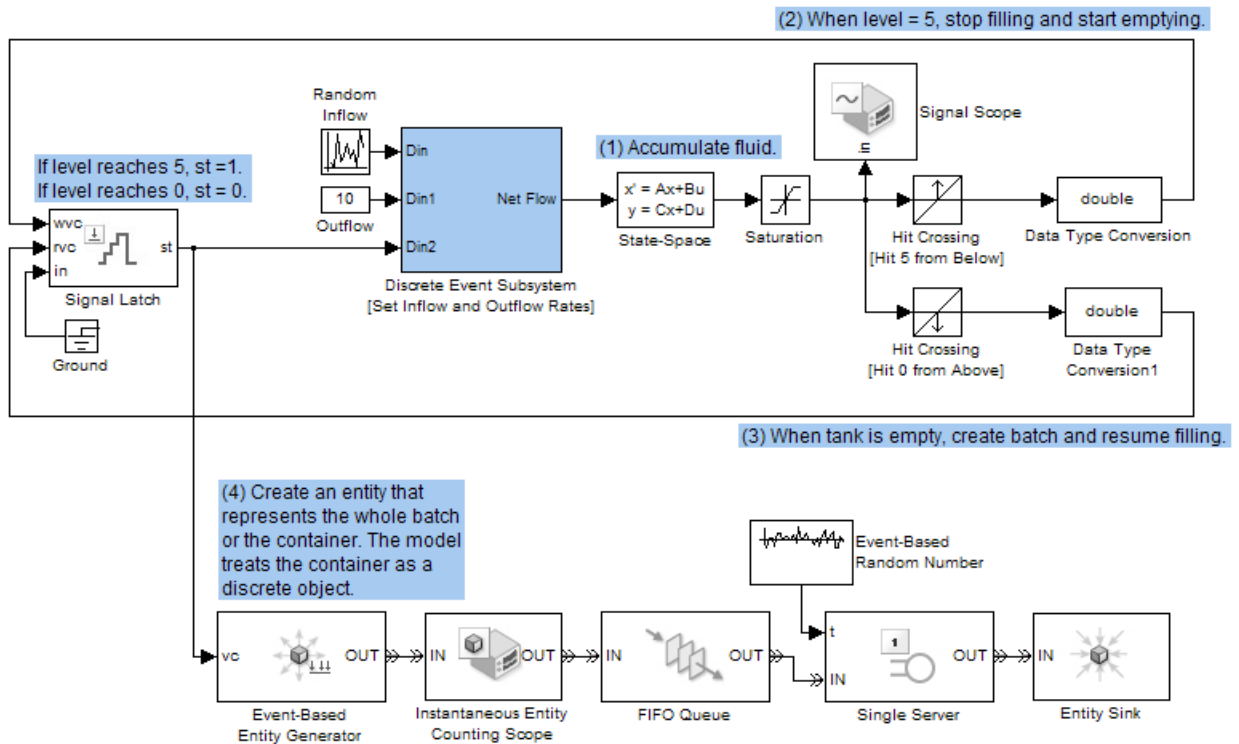
Subsystem Contents

If you double-click an Embedded MATLAB Function block in a model, an editor window shows the MATLAB function that specifies the block. In this example, the following MATLAB function computes the index of a queue having the shortest length, where the individual queue lengths are n_1 , n_2 , and n_3 . If more than one queue achieves the minimum, then the computation returns the smallest index among the queues that minimize the length.

```
function p = findmin(n1, n2, n3)
```

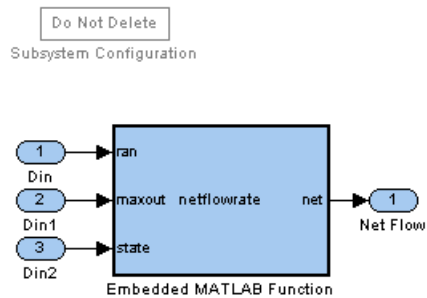
```
% p is the index of a queue having the shortest length.
[~, p] = min([n1 n2 n3]);
```

Note For visual simplicity, the model uses Goto and From blocks to connect the #n signals to the computation.



Top-Level Model

Within a discrete event subsystem, an Embedded MATLAB Function block uses a logical if-then structure to set the inflow and outflow rates. The MATLAB code also computes the net flow rate, which forms the block's output signal. The subsystem and code are below.



Subsystem Contents

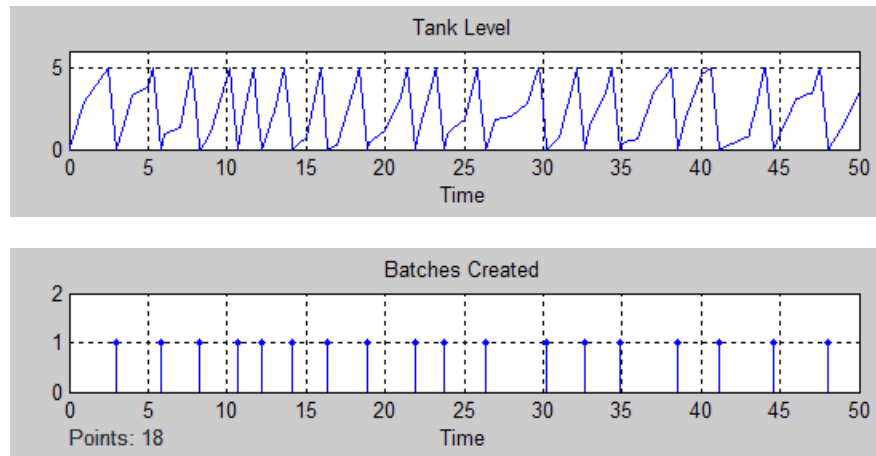
```
function net = netflowrate(ran,maxout,state)

% Compute the inflow and outflow rates.
if (state == 1)
    % Empty the tank.
    in = 0;
    out = maxout;
else
    % Fill the tank.
    in = ran;
    out = 0;
end

% Compute the net flow rate, which forms the output.
net = in - out;
```

While you could alternatively use a Switch block to compute the net flow rate, the choice of MATLAB code or a block-diagram representation might depend on which approach you find more intuitive.

The model plots the continuously changing level of fluid in the tank. A stem plot shows when each batch is created.



Using Logic Blocks

In this section...
“Overview of Use of Logic Blocks” on page 7-10
“Example: Using Servers in Shifts” on page 7-11
“Example: Choosing the Shortest Queue Using Logic Blocks” on page 7-14
“For Further Examples” on page 7-15

Overview of Use of Logic Blocks

The following blocks can be useful for modeling logic because they return a 0 or 1:

- Relational Operator
- Compare To Constant and Compare To Zero
- Interval Test and Interval Test Dynamic
- Detect Change, Detect Decrease, Detect Increase, etc.
- Signal Latch

Note Some blocks return a 0 or 1 of a Boolean or integer data type. Blocks in the SimEvents libraries process signals whose data type is `double`. To convert between data types, use the Data Type Conversion block in the Simulink Signal Attributes library.

For switching, you might need to compute an integer that indicates a port number. Here are some useful blocks for this situation:

- Switch
- Lookup Table
- Bias
- Rounding Function, if an earlier computation returns a noninteger

- Other blocks in the Math Operations library

Example: Using Servers in Shifts

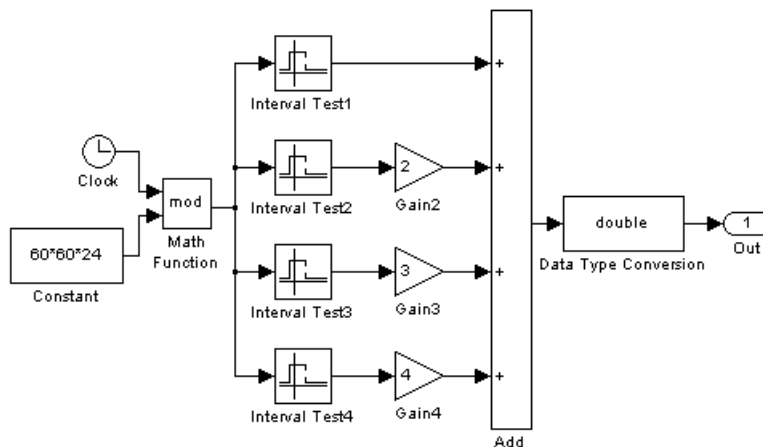
- “Computing the Server Index” on page 7-11
- “Top-Level Model” on page 7-12

Suppose you have four servers that operate in mutually exclusive shifts of unequal lengths, and you want to direct each entity to the server that is currently on duty. Suppose that the server on duty has the following index between 1 and 4:

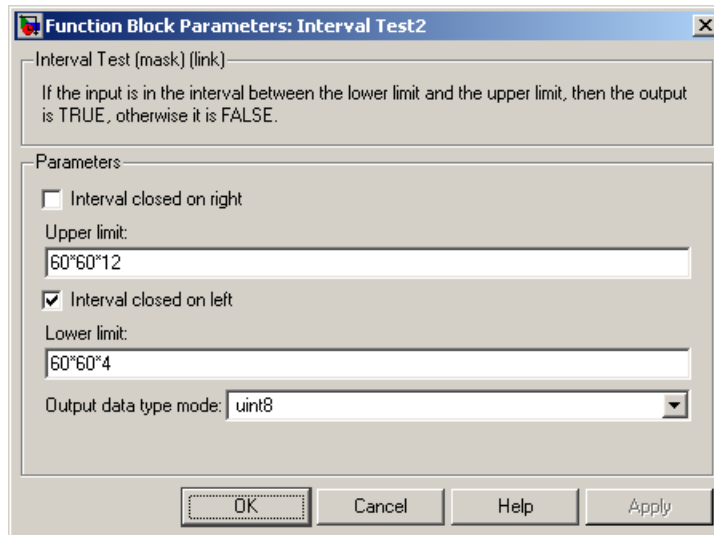
$$\text{Index} = \begin{cases} 1 & \text{between midnight and 4 A.M.} \\ 2 & \text{between 4 A.M. and noon} \\ 3 & \text{between noon and 8 P.M.} \\ 4 & \text{between 8 P.M. and midnight} \end{cases}$$

Computing the Server Index

You can compute the index of the server on duty using a subsystem like the one shown below, where the Interval Test blocks use **Lower limit** and **Upper limit** parameter values that represent the start and end of each shift in each 24-hour day.

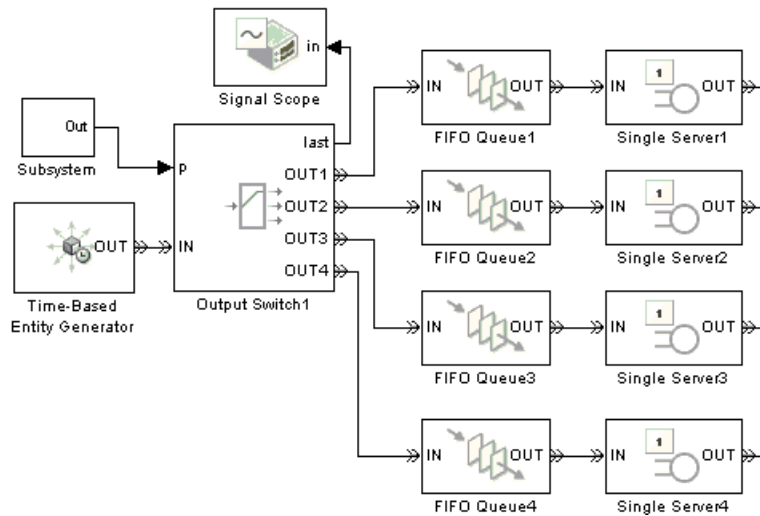


For example, the second shift is represented by the second Interval Test block, whose dialog box is shown below.

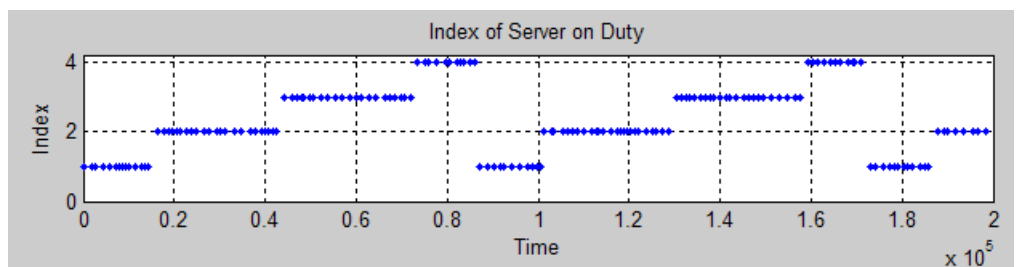


Top-Level Model

The figure below shows how you can integrate either kind of index computation, contained in a subsystem, into a larger model. It is similar to the example in “Example: Choosing the Shortest Queue Using Logic Blocks” on page 7-14 except that this example uses different switching logic that does not depend on feedback from the queues. The subsystem in this model is a virtual subsystem used for visual simplicity, not a discrete event subsystem.

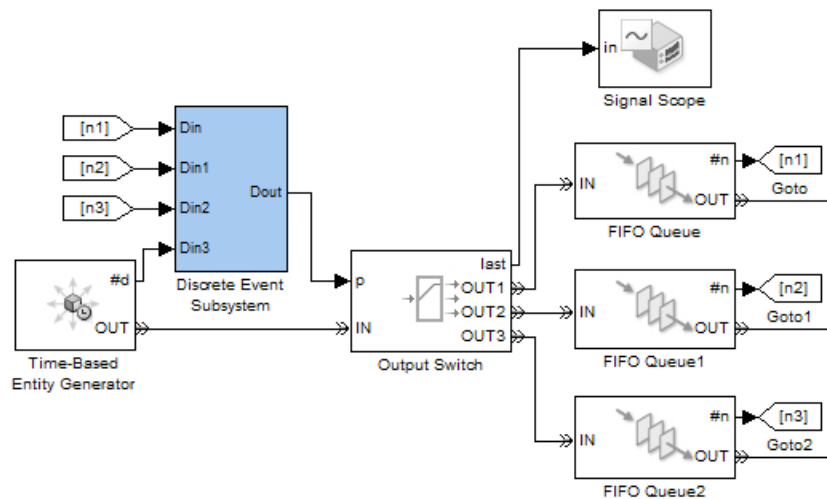


The sample plot below reflects the use of shifts. Each plotting marker corresponds to an entity departing from the switch block via one of the four entity output ports.



Example: Choosing the Shortest Queue Using Logic Blocks

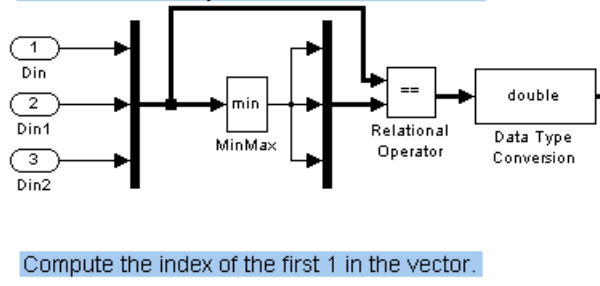
This example, a variation on the model in “Example: Choosing the Shortest Queue” on page 7-3, directs entities to the shortest of three queues. The discrete event subsystem computes the index of the shortest queue using logic blocks. If more than one queue achieves the minimum, then the computation returns the smallest index among the queues that minimize the length.



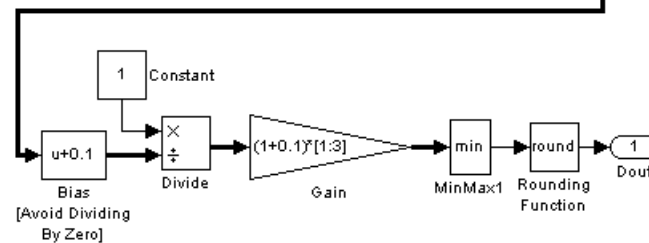
Top-Level Model

Do Not Delete
Subsystem Configuration

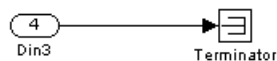
Determine which #n inputs attain the minimum and create a binary vector.



Compute the index of the first 1 in the vector.



Use #d signal to call the subsystem.



Subsystem Contents

For Further Examples

See these sections for additional examples of using logic blocks in SimEvents models:

- The logic diagrams depicted in “Stopping Upon Reaching a Particular State” on page 12-37

- The gate examples, especially “Example: Controlling Joint Availability of Two Servers” on page 8-4

Regulating Arrivals Using Gates

- “Role of Gates in SimEvents Models” on page 8-2
- “Keeping a Gate Open Over a Time Interval” on page 8-4
- “Opening a Gate Instantaneously” on page 8-6
- “Combining Gates” on page 8-9

Role of Gates in SimEvents Models

In this section...
“Overview of Gate Behavior” on page 8-2
“Types of Gate Blocks” on page 8-3

Overview of Gate Behavior

By design, certain blocks change their availability to arriving entities depending on the circumstances. For example,

- A queue or server accepts arriving entities as long as it is not already full to capacity.
- An input switch accepts an arriving entity through a single selected entity input port but forbids arrivals through other entity input ports.

Some applications require more control over whether and when entities advance from one block to the next. A gate provides flexible control via its changing status as either open or closed: by definition, an open gate permits entity arrivals as long as the entities would be able to advance immediately to the next block, while a closed gate forbids entity arrivals. You configure the gate so that it opens and closes under circumstances that are meaningful in your model.

For example, you might use a gate

- To create periods of unavailability of a server. For example, you might be simulating a manufacturing scenario over a monthlong period, where a server represents a machine that runs only 10 hours per day. An enabled gate can precede the server, to make the server’s availability contingent upon the time.

To learn about enabled gates, which can remain open for a time interval of nonzero length, see “Keeping a Gate Open Over a Time Interval” on page 8-4.

- To make departures from one queue contingent upon departures from a second queue. A release gate can follow the first queue. The gate’s control

signal determines when the gate opens, based on decreases in the number of entities in the second queue.

To learn about release gates, which open and then close in the same time instant, see “Opening a Gate Instantaneously” on page 8-6.

- With the **First port that is not blocked** mode of the Output Switch block. Suppose each entity output port of the switch block is followed by a gate block. An entity attempts to advance via the first gate; if it is closed, then the entity attempts to advance via the second gate, and so on.

This arrangement is explored in “Combining Gates” on page 8-9.

To learn about implementing logic that determines when a gate is open or closed, see Chapter 7, “Using Logic”.

Types of Gate Blocks

The Gates library offers these fundamentally different kinds of gate blocks:

- The **Enabled Gate** block, which uses a control signal to determine time intervals over which the gate is open or closed. For more information, see “Keeping a Gate Open Over a Time Interval” on page 8-4.
- The **Release Gate** block, which uses a control signal to determine a discrete set of times at which the gate is instantaneously open. The gate is closed at all other times during the simulation. For more information, see “Opening a Gate Instantaneously” on page 8-6.

Tip Many models follow a gate with a storage block, such as a queue or server.

Keeping a Gate Open Over a Time Interval

In this section...
“Behavior of Enabled Gate Block” on page 8-4
“Example: Controlling Joint Availability of Two Servers” on page 8-4

Behavior of Enabled Gate Block

The Enabled Gate block uses a control signal at the input port labeled **en** to determine when the gate is open or closed:

- When the **en** signal is positive, the gate is open and an entity can arrive as long as it would be able to advance immediately to the next block.
- When the **en** signal is zero or negative, the gate is closed and no entity can arrive.

Because the **en** signal can remain positive for a time interval of arbitrary length, an enabled gate can remain open for a time interval of arbitrary length. The length can be zero or a positive number.

Depending on your application, the **en** signal can arise from time-driven dynamics, state-driven dynamics, a SimEvents block’s statistical output signal, or a computation involving various types of signals.

Example: Controlling Joint Availability of Two Servers

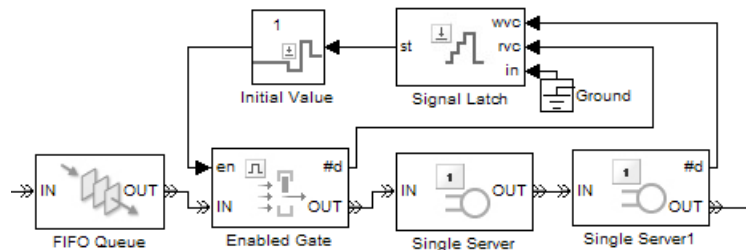
Suppose that each entity undergoes two processes, one at a time, and that the first process does not start if the second process is still in progress for the previous entity. Assume for this example that it is preferable to model the two processes using two Single Server blocks in series rather than one Single Server block whose service time is the sum of the two individual processing times; for example, you might find a two-block solution more intuitive or you might want to access the two Single Server blocks’ utilization output signals independently in another part of the model.

If you connect a queue, a server, and another server in series, then the first server can start serving a new entity while the second server is still serving the previous entity. This does not accomplish the stated goal. The model needs a gate to prevent the first server from accepting an entity too soon, that is, while the second server still holds the previous entity.

One way to implement this is to precede the first Single Server block with an Enabled Gate block that is configured so that the gate is closed when an entity is in either server. In particular, the gate

- Is open from the beginning of the simulation until the first entity's departure from the gate
- Closes whenever an entity advances from the gate to the first server, that is, when the gate block's **#d** output signal increases
- Reopens whenever that entity departs from the second server, that is, when the second server block's **#d** output signal increases

This arrangement is shown below.



The Signal Latch block's **st** output signal becomes 0 when the block's **rvc** input signal increases and becomes 1 when the **wvc** input signal increases. That is, the **st** signal becomes 0 when an entity departs from the gate and becomes 1 when an entity departs from the second server. In summary, the entity at the head of the queue advances to the first Single Server block if and only if both servers are empty.

Opening a Gate Instantaneously

In this section...

“Behavior of Release Gate Block” on page 8-6

“Example: Synchronizing Service Start Times with the Clock” on page 8-6

“Example: Opening a Gate Upon Entity Departures” on page 8-7

Behavior of Release Gate Block

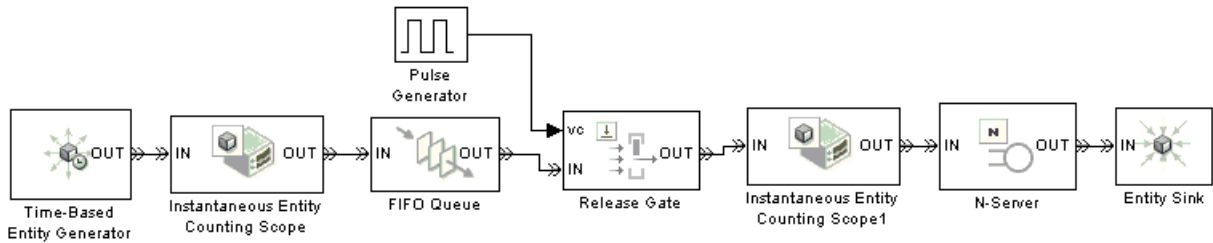
The Release Gate block opens instantaneously at a discrete set of times during the simulation and is closed at all other times. The gate opens when a signal-based event or a function call occurs. By definition, the gate’s opening permits one pending entity to arrive if able to advance immediately to the next block. No simulation time passes between the opening and subsequent closing of the gate; that is, the gate opens and then closes in the same time instant. If no entity is already pending when the gate opens, then the gate closes without processing any entities. It is possible for the gate to open multiple times in a fixed time instant, if multiple gate-opening events occur in that time instant.

An entity passing through a gate must already be pending before the gate-opening event occurs. Suppose a Release Gate block follows a Single Server block and a gate-opening event is scheduled simultaneously with a service completion event. If the gate-opening event is processed first, then the gate opens and closes before the entity completes its service, so the entity does not pass through the gate at that time instant. If the service completion is processed first, then the entity is already pending before the gate-opening event is processed, so the entity passes through the gate at that time instant. To learn more about the processing sequence for simultaneous events, see Chapter 3, “Managing Simultaneous Events”.

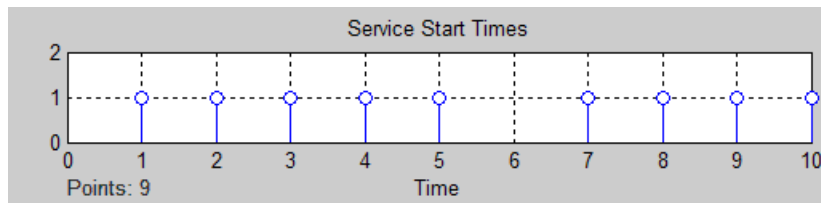
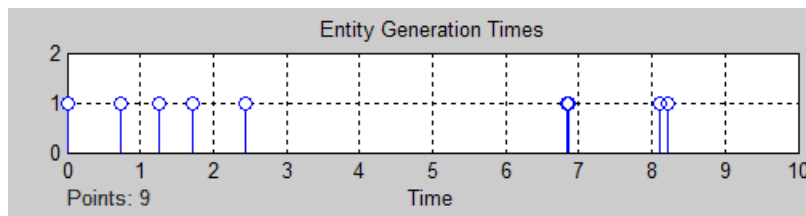
Example: Synchronizing Service Start Times with the Clock

In the example below, a Release Gate block with an input signal from a Pulse Generator block ensures that entities begin their service only at fixed time steps of 1 second, even though the entities arrive asynchronously. In this example, the Release Gate block has **Open gate upon** set to `Change in signal from port vc` and **Type of change in signal value** set to `Rising`,

while the Pulse Generator block has **Period** set to 1. (Alternatively, you could set **Open gate upon** to Trigger from port tr and **Trigger type** to Rising.)

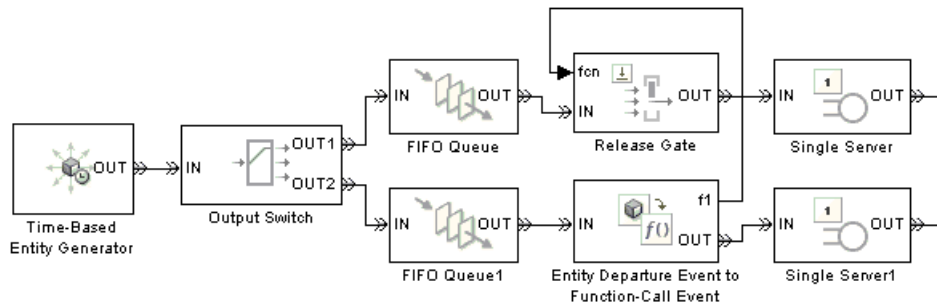


The plots below show that the entity generation times can be noninteger values, but the service beginning times are always integers.



Example: Opening a Gate Upon Entity Departures

In the model below, two queue-server pairs operate in parallel and an entity departs from the top queue only in response to a departure from the bottom queue. In particular, departures from the bottom queue block cause the Entity Departure Event to Function-Call Event block to issue a function call, which in turn causes the gate to open. The Release Gate block in this model has the **Open gate upon** parameter set to Function call from port fcn.



If the top queue in the model is empty when the bottom queue has a departure, then the gate opens but no entity arrives there.

When configuring a gate to open based on entity departures, be sure the logic matches your intentions. For example, when looking at the model shown above, you might assume that entities advance through the queue-server pairs during the simulation. However, if the Output Switch block is configured to select the first entity output port that is not blocked, and if the top queue has a large capacity relative to the number of entities generated during the simulation duration, then you might find that all entities advance to the top queue, not the bottom queue. As a result, no entities depart from the bottom queue and the gate never opens to permit entities to depart from the top queue. By contrast, if the Output Switch block is configured to select randomly between the two entity output ports, then it is likely that some entities reach the servers as expected.

Alternative Using Value Change Events

An alternative to opening the gate upon departures from the bottom queue is to open the gate upon changes in the value of the **#d** signal output from that queue block. The **#d** signal represents the number of entities that have departed from that block, so changes in the value are equivalent to entity departures. To implement this approach, set the Release Gate block's **Open gate upon** parameter to **Change in signal from port vc** and connect the **vc** port to the queue block's **#d** output signal.

Combining Gates

In this section...

“Effect of Combining Gates” on page 8-9

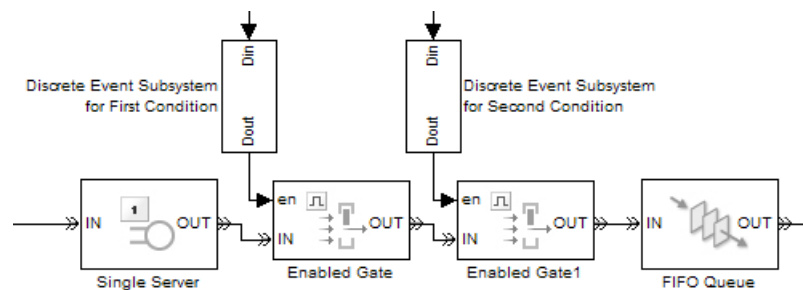
“Example: First Entity as a Special Case” on page 8-11

Effect of Combining Gates

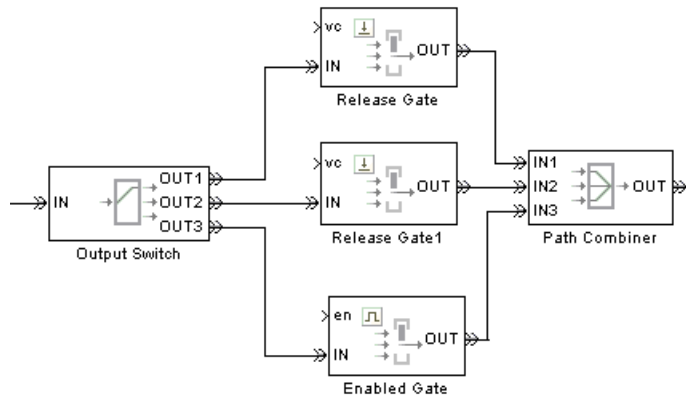
You can use multiple gate blocks in combination with each other:

- Using a Release Gate block and/or one or more Enabled Gate blocks in series is equivalent to a logical AND of their gate-opening criteria. For an entity to pass through the gates, they must all be open at the same time. The next figure shows a logical AND of two conditions.

Note Do not connect two Release Gate blocks in series. No entities would ever pass through such a series of gates because each gate closes before the other gate opens, even if the gate-opening events occur at the same value of the simulation clock.



- Using multiple gate blocks in parallel, you can implement a logical OR of their gate-opening criteria. Use the Output Switch and Path Combiner blocks as in the figure below and set the Output Switch block's **Switching criterion** parameter to First port that is not blocked.



Each entity attempts to arrive at the first gate; if it is closed, the entity attempts to arrive at the second gate, and so on. If all gates are closed, then the Output Switch block's entity input port is unavailable and the entity must stay in a preceding block (such as a queue or server preceding the switch).

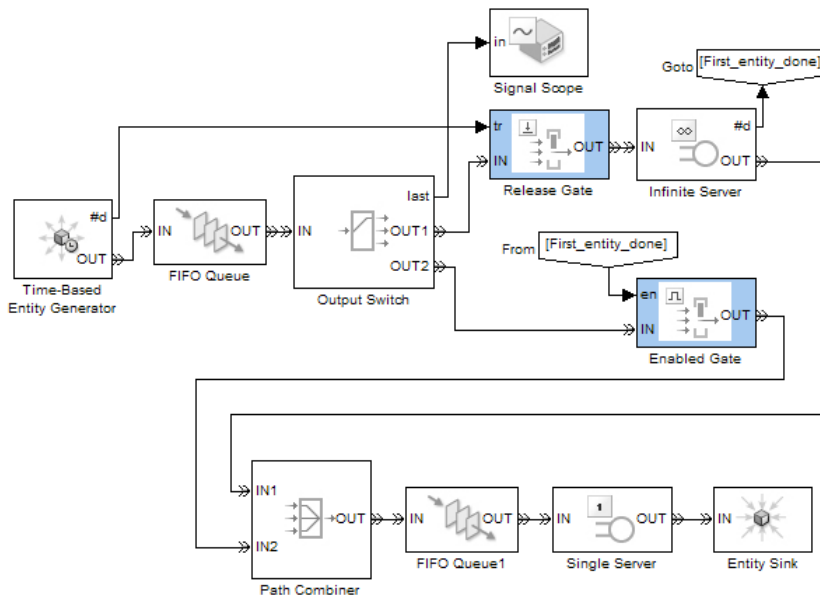
Note The figure above uses two Release Gate blocks and one Enabled Gate block, but you can use whatever combination is suitable for the logic of your application and whatever sequence you prefer. Also, the figure above omits the control signals (**vc** and **en**) for visual clarity but in your model these ports must be connected.

The Enabled Gate and Release Gate blocks open and close their gates in response to updates in their input signals. If you expect input signals for different gate blocks to experience simultaneous updates, then consider the sequence in which the application resolves the simultaneous updates. For example, if you connect an Enabled Gate block to a Release Gate block in series and the enabled gate closes at the same time that the release gate opens, then the sequence matters. If the gate-closing event is processed first, then a pending entity cannot pass through the gates at that time; if the gate-opening event is processed first, then a pending entity can pass through the gates before the gate-closing event is processed. To control the sequence, select the **Resolve simultaneous signal updates according to event priority** parameters in the gate blocks and specify appropriate **Event**

priority parameters. For details, see Chapter 3, “Managing Simultaneous Events”.

Example: First Entity as a Special Case

This example illustrates the use of a Release Gate block and an Enabled Gate block connected in parallel. The Release Gate block permits the arrival of the first entity of the simulation, which receives special treatment, while the Enabled Gate block permits entity arrivals during the rest of the simulation. In this example, a warmup period at the beginning of the simulation precedes normal processing.



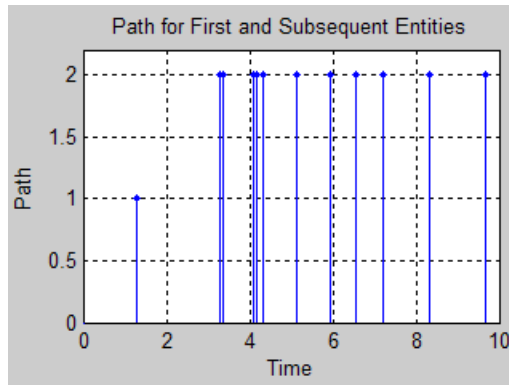
The Release Gate block is open precisely when the **#d** output signal from the Time-Based Entity Generator block rises from 0 to 1. That is, the gate is open for the first entity of the simulation and no other entities. The first entity arrives at an Infinite Server block, which represents the warmup period.

Subsequent entities find the Release Gate block’s entity input port unavailable, so they attempt to arrive at the Enabled Gate block. The Enabled Gate block is open during the entire simulation, except when the first entity

has not yet departed from the Infinite Server block. This logic is necessary to prevent the second entity from jumping ahead of the first entity before the warmup period is over.

The Path Combiner block merges the two entity paths, removing the distinction between them. Subsequent processing depends on your application; this model merely uses a queue-server pair as an example.

The plot below shows which path each entity takes during the simulation. The plot shows that the first entity advances from the first (Path=1) entity output port of the Output Switch block to the Release Gate block, while subsequent entities advance from the second (Path=2) entity output port of the Output Switch block to the Enabled Gate block.



Forcing Departures Using Timeouts

- “Role of Timeouts in SimEvents Models” on page 9-2
- “Basic Example Using Timeouts” on page 9-3
- “Basic Procedure for Using Timeouts” on page 9-4
- “Defining Entity Paths on Which Timeouts Apply” on page 9-7
- “Handling Entities That Time Out” on page 9-10
- “Example: Limiting the Time Until Service Completion” on page 9-14

Role of Timeouts in SimEvents Models

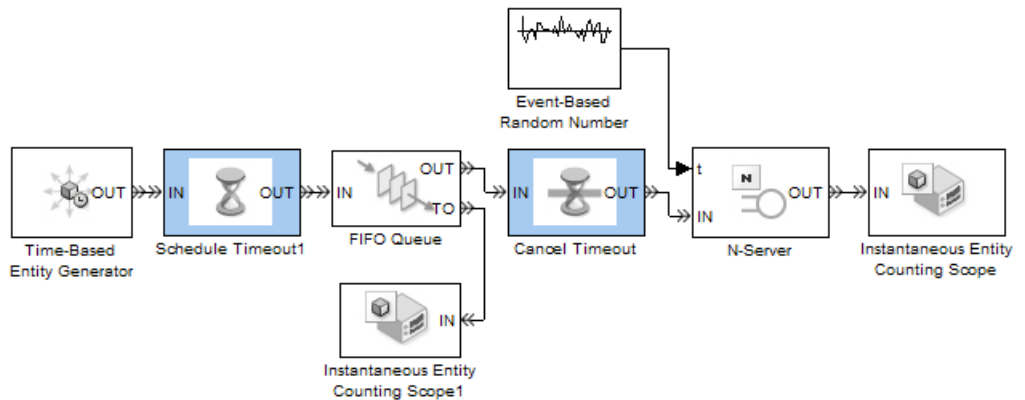
You can limit the amount of time an entity spends during the simulation on designated entity paths. Exceeding the limit causes a *timeout event* and the entity is said to have *timed out*. The duration of the time limit is called the *timeout interval*.

You might use timeout events to

- Model a protocol that explicitly calls for timeouts.
- Implement special routing or other handling of entities that exceed a time limit.
- Model entities that represent something perishable.
- Identify blocks in which entities wait too long.

Basic Example Using Timeouts

The model below limits the time that each entity can spend in a queue, but does not limit the time in the server. The queue immediately ejects any entity that exceeds the time limit. For example, if each entity represents customers trying to reach an operator in a telephone support call center, then the model describes customers hanging up the telephone if they wait too long to reach an operator. If customers reach an operator, they complete the call and do not hang up prematurely.



Each customer's arrival at the Schedule Timeout block establishes a time limit for that customer. Subsequent outcomes for that customer are as follows:

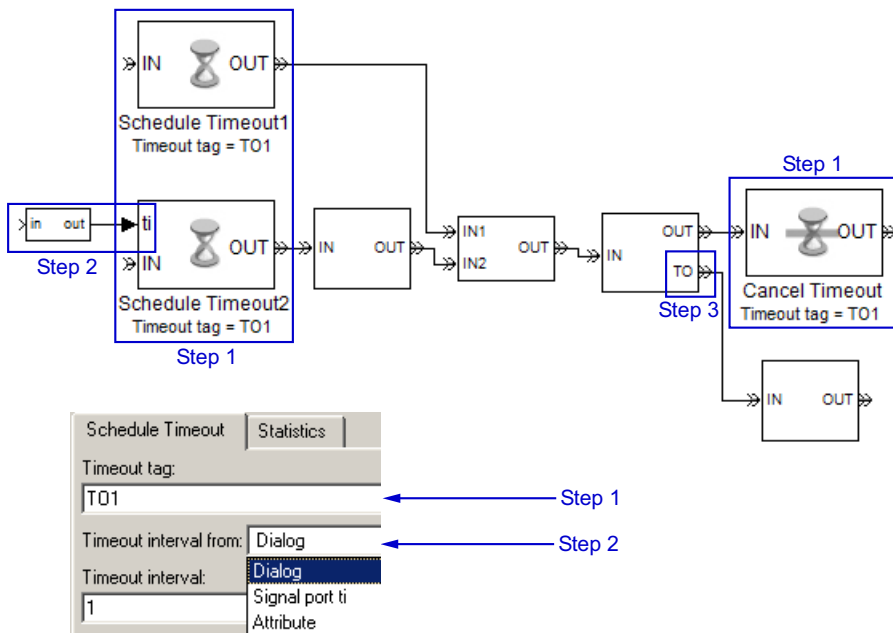
- **Entity Times Out** — If the customer is still in the queue when the clock reaches the time limit, the customer hangs up without reaching an operator. In generic terms, the entity times out, departs from the FIFO Queue block via the **TO** port, and does not reach the server.
- **Entity Advances to Server** — If the customer gets beyond the queue before the clock reaches the time limit, the customer decides not to hang up and begins talking with the operator. In generic terms, if the entity arrives at the Cancel Timeout block before the clock reaches the time limit, the entity loses its potential to time out because the block cancels a pending timeout event. The entity then advances to the server.

Basic Procedure for Using Timeouts

In this section...
“Schematic Illustrating Procedure” on page 9-4
“Step 1: Designate the Entity Path” on page 9-5
“Step 2: Specify the Timeout Interval” on page 9-5
“Step 3: Specify Destinations for Timed-Out Entities” on page 9-6

Schematic Illustrating Procedure

This section describes a typical procedure for incorporating timeout events into your model. The schematic below illustrates the procedure for a particular topology.



Step 1: Designate the Entity Path

Designate the entity path on which you want to limit entities' time. The path can be linear, with exactly one initial block and one final block, or the path can be nonlinear, possibly with multiple initial or final blocks. Insert Schedule Timeout and Cancel Timeout blocks as follows:

- Insert a Schedule Timeout block before each initial block in the path. The Schedule Timeout block schedules a timeout event on the event calendar whenever an entity arrives, that is, whenever an entity enters your designated path.
- Insert a Cancel Timeout block after each final block in the path, except final blocks that have no entity output port. The Cancel Timeout block removes a timeout event from the event calendar whenever an entity arrives, that is, whenever an entity leaves your designated path without having timed out. If a final block in the path has no entity output port, then the block automatically cancels the timeout event.
- Configure the Schedule Timeout and Cancel Timeout blocks with the same **Timeout tag** parameter. The timeout tag is a name that distinguishes a particular timeout event from other timeout events scheduled for different times for the same entity.

For sample topologies, see “Defining Entity Paths on Which Timeouts Apply” on page 9-7.

Step 2: Specify the Timeout Interval

Specify the timeout interval, that is, the maximum length of time that the entity can spend on the designated entity path, by configuring the Schedule Timeout block(s) you inserted:

- If the interval is the same for all entities that arrive at that block, you can use a parameter, attribute, or signal input. Indicate your choice using the the Schedule Timeout block's **Timeout interval from** parameter.
- If each entity stores its own timeout interval in an attribute, set the Schedule Timeout block's **Timeout interval from** parameter to **Attribute**.

This method is preferable to using the `Signal port ti` option with a Get Attribute block connected to the `ti` port; to learn why, see “Interleaving of Block Operations” on page 16-25.

- If the timeout interval can vary based on dynamics in the model, set the Schedule Timeout block’s **Timeout interval from** parameter to `Signal port ti`. Connect a signal representing the timeout interval to the `ti` port.

If the `ti` signal is an event-based signal, be sure that its updates occur before the entity arrives. For common problems and troubleshooting tips, see “Unexpected Use of Old Value of Signal” on page 14-76.

Step 3: Specify Destinations for Timed-Out Entities

Specify where an entity goes if it times out during the simulation:

- Enable the **TO** entity output port for some or all queues, servers, and Output Switch blocks along the entity’s path, by selecting **Enable TO port for timed-out entities** on the **Timeout** tab of the block’s dialog box. In the case of the Output Switch block, you can select that option only under certain configurations of the block; see its reference page for details.

If an entity times out while it is in a block possessing a **TO** port, the entity departs using that port.

- If an entity times out while it resides in a block that has no **TO** port, then the Schedule Timeout block’s **If entity has no destination when timeout occurs** parameter indicates whether the simulation halts with an error or discards the entity while issuing a warning.

Queues, servers, and the Output Switch block are the only blocks that can possess **TO** ports. For example, an entity cannot time out from gate or attribute blocks.

For examples of ways to handle timed-out entities, see “Handling Entities That Time Out” on page 9-10.

Defining Entity Paths on Which Timeouts Apply

In this section...

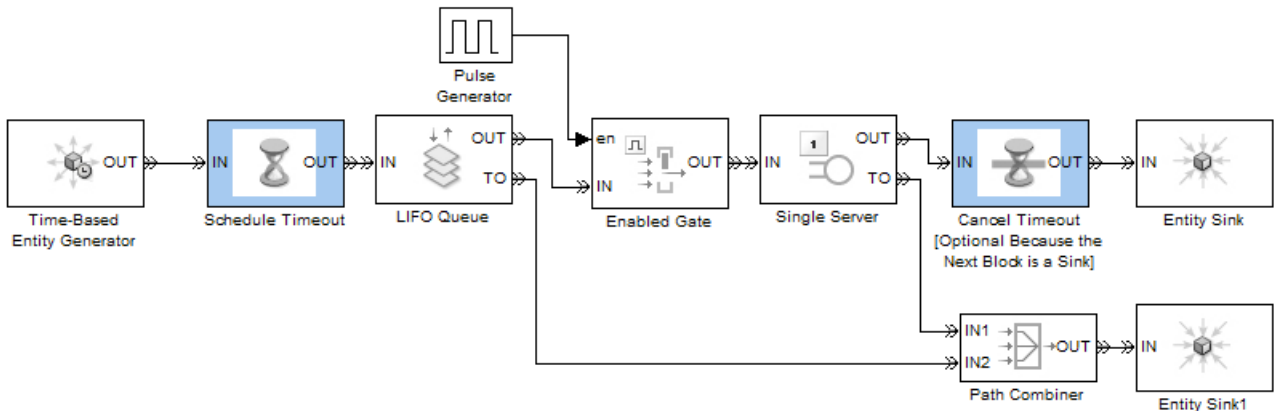
“Linear Path for Timeouts” on page 9-7

“Branched Path for Timeouts” on page 9-8

“Feedback Path for Timeouts” on page 9-8

Linear Path for Timeouts

The next figure illustrates how to position Schedule Timeout and Cancel Timeout blocks to limit the time on a linear entity path. The linear path has exactly one initial block and one final block. A Schedule Timeout block precedes the initial block (LIFO Queue) on the designated entity path, while a Cancel Timeout block follows the final block (Single Server) on the designated entity path.



In this example, the Cancel Timeout block is optional because it is connected to the Entity Sink block, which has no entity output ports. However, you might want to include the Cancel Timeout block in your own models for clarity or for its optional output signals.

Other examples of timeouts on linear entity paths include these:

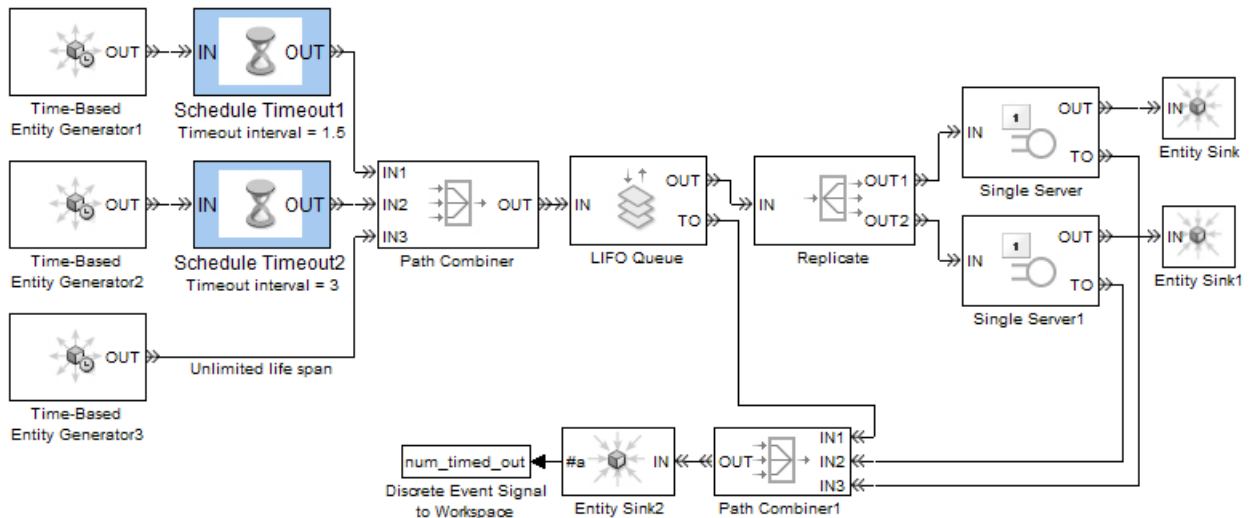
- “Basic Example Using Timeouts” on page 9-3

- “Example: Limiting the Time Until Service Completion” on page 9-14

Branched Path for Timeouts

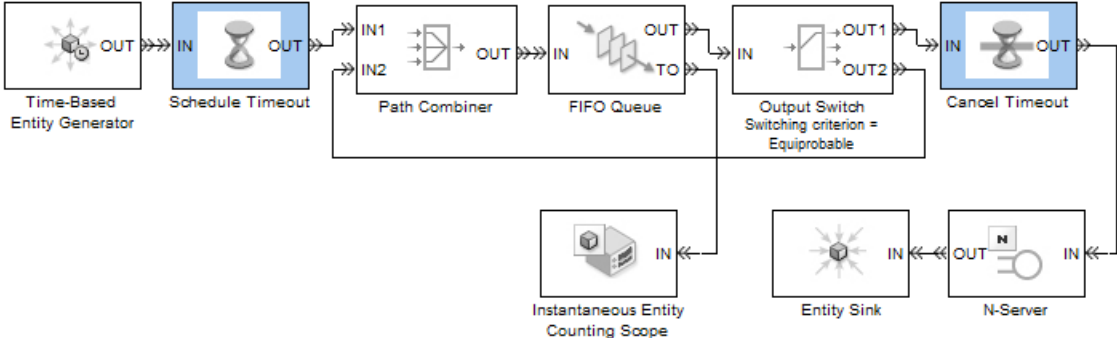
In the example below, entities from two sources have limited lifespans. Entities from a third source do not have limited lifespans.

Note When the Replicate block replicates an entity subject to a timeout, all departing entities share the same expiration time; that is, the timeout events corresponding to all departing entities share the same scheduled event time.



Feedback Path for Timeouts

In the example below, entities have limited total time in a queue, whether they travel directly from there to the server or loop back to the end of the queue one or more times.



Handling Entities That Time Out

In this section...

“Common Requirements for Handling Timed-Out Entities” on page 9-10

“Techniques for Handling Timed-Out Entities” on page 9-10

“Example: Dropped and Timed-Out Packets” on page 9-11

“Example: Rerouting Timed-Out Entities to Expedite Handling” on page 9-12

Common Requirements for Handling Timed-Out Entities

Your requirements for handling entities that time out might depend on your application or model. For example, you might want to

- Count timed-out entities to create metrics.
- Process timed-out entities specially.
- Discard timed-out entities without reacting to the timeout event in any other way.

Techniques for Handling Timed-Out Entities

To process or count timed-out entities, use one or more of the following optional ports of the individual queues, servers, and Output Switch blocks in the entities’ path. Parameters in the dialog boxes of the blocks let you enable the optional ports.

Port	Description	Parameter that Enables Port
Entity output port TO	Timed-out entities depart via this port, if present.	Enable TO port for timed-out entities on Timeout tab
Signal output port #to	Number of entities that have timed out from the block since the start of the simulation.	Number of entities timed out on Statistics tab

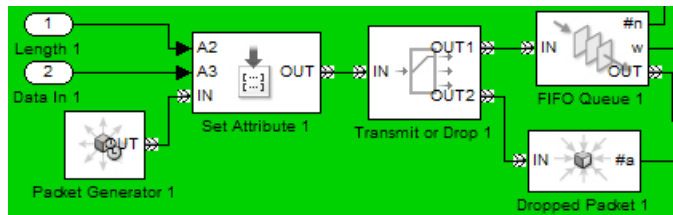
To combine paths of timed-out entities from multiple blocks, use a Path Combiner block.

To count entities that time out from various blocks, add the various **#to** signals using a discrete event subsystem. See the Time-Driven and Event-Driven Addition demo for an example of adding event-based signals.

Note If an entity times out while it is in a block that has no **TO** port, then the Schedule Timeout block's **If entity has no destination when timeout occurs** parameter indicates whether the simulation halts with an error or discards the entity while issuing a warning.

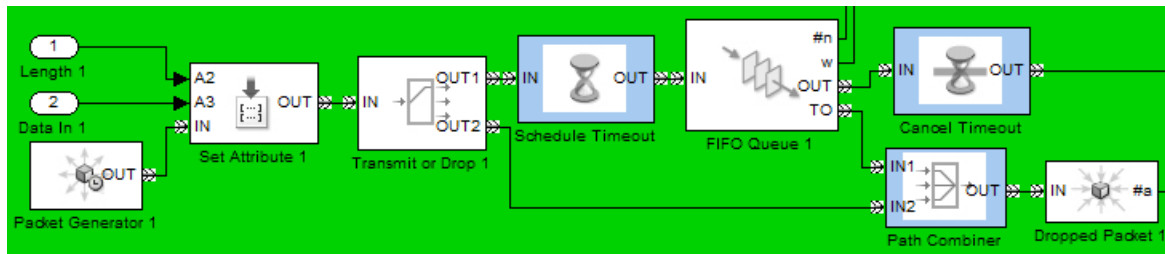
Example: Dropped and Timed-Out Packets

This example modifies a portion of the Shared Access Communications Media demo, by making it possible for a packet to time out while it is in a queue waiting for access to the communications media. Below is a portion of the original model, representing one transmitter. The transmitter drops any new packet that arrives while a packet is already waiting in the queue, and the simulation counts dropped packets.



Original Transmitter

Below, a modified version of the model places a Schedule Timeout block before the queue and a Cancel Timeout block after the queue. Furthermore, the newly enabled **TO** entity output port of the FIFO Queue block connects to a Path Combiner block. The Path Combiner block accepts packets that the transmitter drops immediately on arrival, as well as packets that the transmitter drops due to a timeout. The count of untransmitted packets now reflects both scenarios.

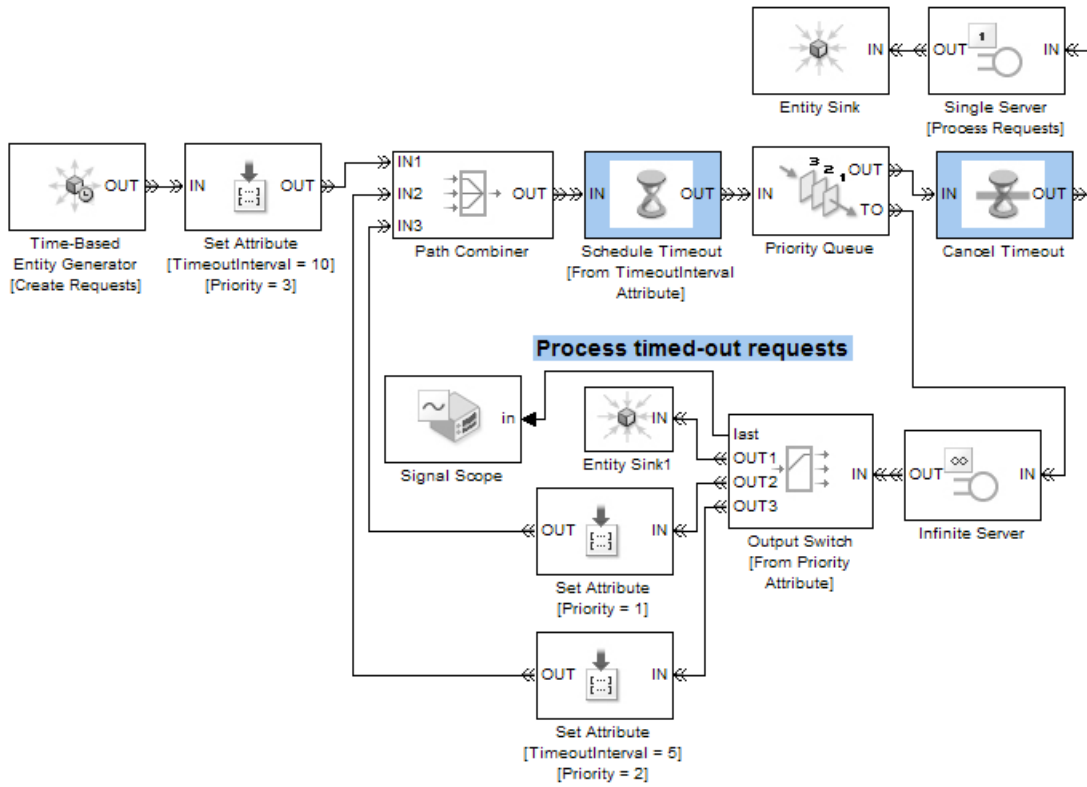


Transmitter Counting Dropped and Timed-Out Packets

Example: Rerouting Timed-Out Entities to Expedite Handling

In this example, timeouts and a priority queue combine to expedite the handling of requests that have waited for a long time in the queue. Requests initially have priority 3, which is the least important priority level in this model. If a request remains unprocessed for too long, it leaves the Priority Queue block via the **TO** entity output port. Subsequent processing is as follows:

- A priority-3 request becomes a priority-2 request, the timeout interval becomes shorter, and the request reenters the priority queue. The queue places this request ahead of all priority-3 requests already in the queue.
- A priority-2 request becomes a priority-1 request, the timeout interval remains unchanged, and the request reenters the priority queue. The queue places this request ahead of all priority-3 and priority-2 requests already in the queue.
- A priority-1 request, having timed out three times, is discarded.

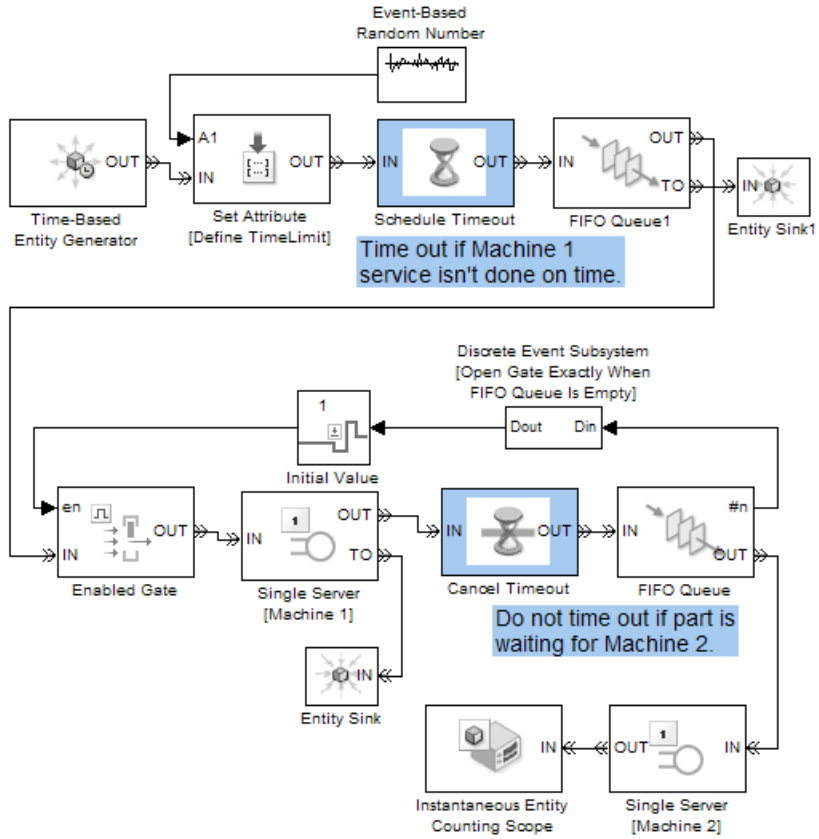


Example: Limiting the Time Until Service Completion

In this example, two machines operate in series to process parts. The example seeks to establish a time limit for the first machine's completion of active processing, not including any subsequent time that a part might need to wait for the second machine to be ready.

A Schedule Timeout block establishes the time limit before the part waits for the first machine. A Cancel Timeout block cancels the timeout event after the first machine's processing is complete. However, placing only a Cancel Timeout block between the two machines, modeled here as Single Server blocks, would not accomplish the goal because the part might time out while it is blocked in the first Single Server block.

The solution is to use a queue to provide a waiting area for the part while it waits for the second machine, and use a gate to prevent the first machine from working on a new part until the part has successfully advanced to the second machine. In the model below, parts always depart from the first Single Server block immediately after the service is complete; as a result, the time limit applies precisely to the service completion.



Controlling Timing with Subsystems

- “Timing Issues in SimEvents Models” on page 10-2
- “Role of Discrete Event Subsystems in SimEvents Models” on page 10-7
- “Blocks Inside Discrete Event Subsystems” on page 10-10
- “Working with Discrete Event Subsystem Blocks” on page 10-11
- “Examples Using Discrete Event Subsystem Blocks” on page 10-18
- “Creating Entity-Departure Subsystems” on page 10-27
- “Examples Using Entity-Departure Subsystems” on page 10-30
- “Using Function-Call Subsystems” on page 10-33

Timing Issues in SimEvents Models

In this section...

“Overview of Timing Issues” on page 10-2

“Timing for the End of the Simulation” on page 10-2

“Timing for a Statistical Computation” on page 10-3

“Timing for Choosing a Port Using a Sequence” on page 10-4

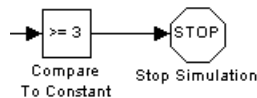
Overview of Timing Issues

Most SimEvents models contain one or more inherently time-based blocks from the Simulink libraries, in addition to inherently event-based blocks from the SimEvents libraries. When time-based and event-based behavior combine in one model, it is important to use correct modeling techniques to ensure correct timing in the simulation.

When you combine time-based and event-based blocks, consider whether you want the operation of a time-based block to depend *only* on time-oriented considerations or whether you want the time-based block to respond to events. The example scenarios that follow illustrate the difference. Also, the section “Role of Discrete Event Subsystems in SimEvents Models” on page 10-7 introduces the discrete event subsystem, an important modeling construct that you can use to make time-based blocks respond to events.

Timing for the End of the Simulation

Consider a queuing model in which you want to end the simulation precisely when the number of entities in the queue first equals or exceeds some threshold value. One way to model the constraint is to use the Compare To Constant block to check whether the queue length is greater than or equal to the threshold value and the Stop Simulation block to end the simulation whenever the comparison is true.



The queue length is an optional output from the FIFO Queue block from the signal output port labeled **#n**.



Time-Based Default Behavior

Connecting the **#n** signal directly to the signal input port of the Compare To Constant block looks correct topologically. However, that does *not* cause the simulation to stop at the exact moment when the **#n** signal reaches or exceeds the threshold for the first time. Instead, it causes the simulation to stop at the next time step for the time-based Compare To Constant and Stop Simulation blocks, which can be at a later time. The **#n** signal experiences a change in value based on an event, at a time that is unrelated to the time-based simulation clock, whereas most Simulink blocks default to time-based behavior.

Achieving Correct Event-Based Behavior

The correct way to cause the simulation to stop at the exact moment when the **#n** signal reaches or exceeds the threshold for the first time is to construct the model so that the Compare To Constant and Stop Simulation blocks respond to events, not the time-based simulation clock. Put these blocks inside a discrete event subsystem that is executed at exactly those times when the FIFO Queue block's **#n** signal increases from one integer to another.

For details on this solution, see “Example: Ending the Simulation Upon an Event” on page 10-21.

Timing for a Statistical Computation

Suppose that you want to compare the lengths of two queues. You can use the optional **#n** signal output port from each of two FIFO Queue blocks to produce the queue length signal, and the Relational Operator block to compare the values of the two signals.

Time-Based Comparison

The Relational Operator block in the Simulink library set is inherently time-based. In this example, you are comparing signals that experience asynchronous discontinuities. Comparing the signals in a time-based manner yields incorrect results because a discontinuity might occur between successive values of the time-based simulation clock.

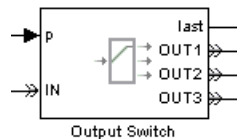
Achieving Correct Event-Based Behavior

To make the Relational Operator block respond to events, not the time-based simulation clock, you can put the block inside a discrete event subsystem that is executed at exactly those times when one of the FIFO Queue blocks' #n signals changes from one integer to another. This solution causes the simulation to perform the comparison at the correct times, which leads to a correct numerical result.

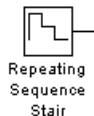
For details on this solution, see “Example: Comparing the Lengths of Two Queues” on page 10-18.

Timing for Choosing a Port Using a Sequence

Consider an Output Switch block that directs each arriving entity to one of three entity output ports, depending on the value of an input signal.



Suppose a Repeating Sequence Stair block generates the input signal by cycling through the values 3, 2, and 1 throughout the simulation.



So far, this description does not indicate *when* the Repeating Sequence Stair block changes its value. Here are some possibilities:

- It chooses a new value from the repeating sequence at regular *time* intervals, which might be appropriate if the switch is intended to select among three operators who work in mutually exclusive shifts in time.

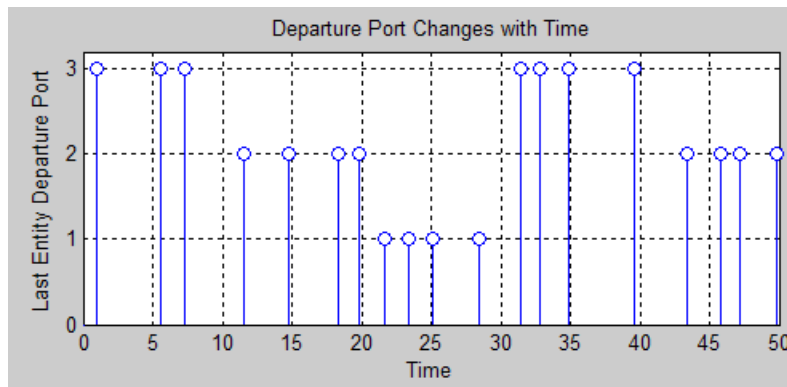
You can specify the time interval in the Repeating Sequence Stair block's **Sample time** parameter.

- It chooses a new value from the repeating sequence whenever an entity arrives at the Output Switch block, which might be appropriate if the switch is intended to distribute a load fairly among three operators who work in parallel.

To make the Repeating Sequence Stair block respond to entity advancement events, not the time-based simulation clock, you can put these blocks inside an appropriately configured function-call subsystem, as discussed in “Creating Entity-Departure Subsystems” on page 10-27.

For details on this approach, see “Example: Using Entity-Based Timing for Choosing a Port” on page 10-30.

These possibilities correspond to qualitatively different interpretations of the model as well as quantitatively different results. If the Output Switch block reports the index of its last selected entity output port (that is, the entity output port through which the most recent entity departure occurred), then a plot of this statistic against time looks quite different depending on the timing of the Repeating Sequence Stair block's operation. See the next two sample plots.



Departure Port Changes with Time

Role of Discrete Event Subsystems in SimEvents Models

In this section...

“Overview of Discrete Event Subsystems” on page 10-7

“Purpose of Discrete Event Subsystems” on page 10-8

“Processing Sequence for Events in Discrete Event Subsystems” on page 10-8

Overview of Discrete Event Subsystems

Given the questions raised in “Timing Issues in SimEvents Models” on page 10-2 about the response of time-based blocks to events, this section gives an overview of discrete event subsystems and describes how you can use them to ensure appropriate simulation timing. A discrete event subsystem:

- Contains time-based blocks. Examples include:
 - Constant with a sample time of inf
 - Sum or Relational Operator with a sample time of -1
 - Stop Simulation
- Cannot contain blocks from the SimEvents libraries, except the Discrete Event Inport, Discrete Event Outport, and Subsystem Configuration blocks.

Note If you want to put blocks that have entity ports into a subsystem as a way to group related blocks or to make a large model more hierarchical, select one or more blocks and use the **Edit > Create Subsystem** menu option. The result is a virtual subsystem, which does not affect the timing of the simulation but is merely a graphical construct.

- Has two basic forms: a Discrete Event Subsystem block and an appropriately configured Function-Call Subsystem block. A special case of the latter form is called an *entity-departure subsystem*.

- Is executed in response to signal-based events that you specify in the Discrete Event Inport blocks inside the Discrete Event Subsystem window, or in response to function calls in the case of a function-call subsystem.

“Block execution” in this documentation is shorthand for “block methods execution.” Methods are functions that the Simulink engine uses to solve a model. Blocks are made up of multiple methods. For details, see “Block Methods” in the Simulink documentation.

Purpose of Discrete Event Subsystems

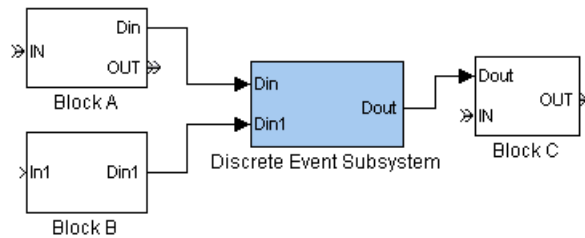
The purpose of a discrete event subsystem is to call the blocks in the subsystem at the exact time of each qualifying event and not at times suggested by the time-based simulation clock. This is an important change in the semantics of the model, not merely an optimization.

Processing Sequence for Events in Discrete Event Subsystems

When creating a discrete event subsystem, you might need to confirm or manipulate the processing sequence for two or more events, such as:

- Signal-based events that execute a Discrete Event Subsystem block
- Entity departures that execute an entity-departure subsystem
- Function calls that execute a function-call subsystem
- Updates in the values of signals that serve as inputs to any kind of discrete event subsystem

Consider the next schematic, which involves a discrete event subsystem. Suppose an entity departure from Block A, an entity arrival at Block C, and updates in all of the signals occur at a given value of the simulation clock.



Typically, the goal is to execute the subsystem:

- After the entity departure from Block A, which produces a signal that is an input to the subsystem.
- After both Block A and Block B update their output signals at that value of the simulation clock.

Be especially aware of this if you clear the **Execute subsystem upon signal-based events** option in a Discrete Event Inport block. Because the subsystem uses the most recent value of the signal, make sure that value is up to date, rather than a value from a previous call to the block that creates the signal.

See “Example: Detecting Changes from Empty to Nonempty” on page 10-24 for an example in which the last-updated signal executes the subsystem. See “Update Sequence for Output Signals” on page 16-34 to learn more about when blocks update their output signals.

- Before the entity processing at Block C, which uses an output signal from the subsystem. “Example: Normalizing a Statistic to Use for Routing” on page 10-19 shows how storing the entity in a switch before routing the entity to an output port can help produce the correct processing sequence. Another technique is to use an extra server block whose service time is zero; for details and examples, see “Interleaving of Block Operations” on page 16-25.

For details on processing sequences, see “Interleaving of Block Operations” on page 16-25, “Processing Sequence for Simultaneous Events” on page 16-2, and Chapter 3, “Managing Simultaneous Events”.

Blocks Inside Discrete Event Subsystems

The only blocks that are suitable for use in a discrete event subsystem are:

- Blocks having a **Sample time** parameter of -1, which indicates that the sample time is inherited from the driving block. An exception is the Discrete-Time Integrator block.
- Blocks that always inherit a sample time from the driving block, such as the Bias block or the Unary Minus block. An exception is the Merge block. To determine whether a block in one of the Simulink libraries inherits its sample time from the driving block, see the “Characteristics” table near the end of the block’s online reference page.
- Blocks whose outputs cannot change from their initial values during a simulation. For more information, see “Constant Sample Time” in the Simulink documentation.

Types of blocks that are *not* suitable for use in a discrete event subsystem include:

- Continuous-time blocks
- Discrete-time blocks with a **Sample time** parameter value that is positive and finite, and the Discrete-Time Integrator block with any sample time
- Blocks from the SimEvents libraries, except the Discrete Event Inport, Discrete Event Outport, and Subsystem Configuration blocks. In particular, a discrete event subsystem cannot contain blocks that possess entity ports or nested Discrete Event Subsystem blocks.
- Merge block

In some cases, you can work around these restrictions by entering a **Sample time** parameter value of -1 and/or by finding a discrete-time analogue of a continuous-time block. For example, instead of using the continuous-time Clock block, use the discrete-time Digital Clock block with a **Sample time** parameter value of -1.

Working with Discrete Event Subsystem Blocks

In this section...

“Setting Up Signal-Based Discrete Event Subsystems” on page 10-11

“Signal-Based Events That Control Discrete Event Subsystems” on page 10-14

For discrete event subsystems that respond to entity departures rather than signal-based events, see “Creating Entity-Departure Subsystems” on page 10-27.

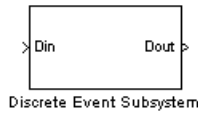
Setting Up Signal-Based Discrete Event Subsystems

Building on the conceptual information in “Role of Discrete Event Subsystems in SimEvents Models” on page 10-7, this section provides some practical information to help you incorporate Discrete Event Subsystem blocks into your SimEvents models.

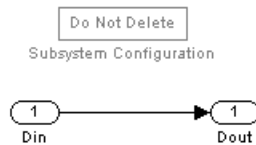
To create discrete event subsystems that respond to signal-based events, follow the next procedure using blocks in the SimEvents Ports and Subsystems library.

Note You cannot create a signal-based discrete event subsystem by selecting blocks and using **Edit > Create Subsystem** or by converting a time-based subsystem into a discrete event subsystem. If your model already contains the blocks you want to put into a discrete event subsystem, you can copy them into the subsystem window of a Discrete Event Subsystem block while following the next procedure.

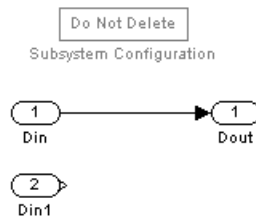
- 1 Drag the Discrete Event Subsystem block from the SimEvents Ports and Subsystems library into your model. Initially, it shows one signal input port **Din** and one signal output port **Dout**. These are signal ports, not entity ports, because the subsystem is designed to process signals rather than entities. Furthermore, the signal input ports carry only signals of data type `double`.



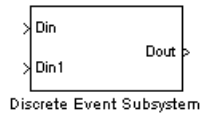
- 2 In the model window, double-click the Discrete Event Subsystem block to open the subsystem it represents. Initially, the subsystem contains an inport block connected to an output block. These are Discrete Event Inport and Discrete Event Outport blocks, which are not the same as the Inport and Outport blocks in the Simulink Ports & Subsystems library. The subsystem also contains a Subsystem Configuration block, which you should not delete.



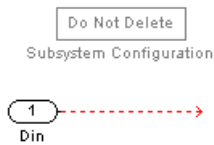
- 3 A discrete event subsystem must have at least one input that determines when the subsystem executes. To change the number of inputs or outputs to the subsystem, change the number of inport and outport blocks in the subsystem window:
 - If your subsystem requires an additional input or output, copy and paste an inport block or output block in the subsystem window. (Copying and pasting is different from duplicating the inport block, which is not supported.) The next figure shows a pasted inport block.



As a result, the subsystem block at the upper level of your model shows the additional port as appropriate. The next figure shows an additional input port on the subsystem block.



- If your subsystem needs no outputs, select and delete the output block in the subsystem window. The next figure shows the absence of output port blocks.

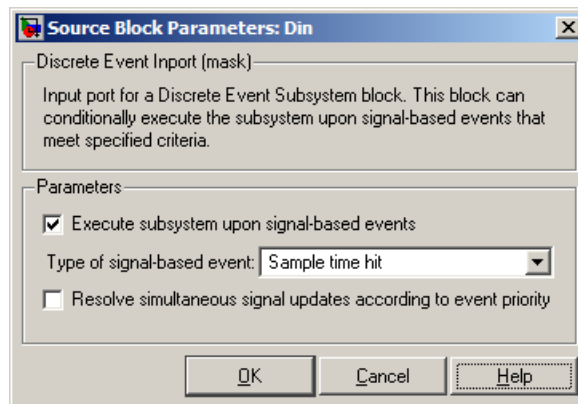


As a result, the subsystem block at the upper level of your model omits the output port. The next figure shows the absence of output ports on the subsystem block.



- 4 Drag other blocks into the subsystem window as appropriate to build the subsystem. This step is similar to the process of building the top level of your model, except that only certain types of blocks are suitable for use inside the subsystem. See “Blocks Inside Discrete Event Subsystems” on page 10-10 for details.

- 5 Configure each of the Discrete Event Inport blocks to indicate when the subsystem should be executed. Each inport block independently determines criteria for executing the subsystem:
 - To execute the subsystem when the signal corresponding to that inport block exhibits a qualifying signal-based event, select **Execute subsystem upon signal-based events** and use additional parameters in the dialog box to describe the signal-based event. If the signal is complex, triggers and value changes cannot serve as qualifying signal-based events.
 - To have the subsystem use the most recent value of the signal corresponding to that inport block without responding to signal-based events in that signal, clear the **Execute subsystem upon signal-based events** option.



Signal-Based Events That Control Discrete Event Subsystems

Blocks in a Discrete Event Subsystem operate in response to signal-based events. Using the dialog box of one or more of the Discrete Event Inport blocks inside the subsystem, you configure the subsystem so that it executes in response to particular types of events. The next table lists the types of events and corresponding values of the **Type of signal-based event** parameter of the Discrete Event Inport block.

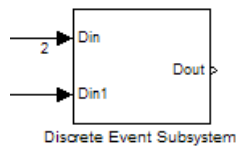
Event	Value of Type of signal-based event Parameter
An update (sample time hit) in a real or complex input signal, even if the updated value is the same as the previous value.	Sample time hit
A change in the value of a real input signal. If the signal is a nonscalar array, the subsystem executes once if any of the positions in the array has a change in value of the kind you specify (Rising, Falling, or Either). For example, a rising value change from [1 2 3] to [1 5 6] calls the subsystem once, not twice.	Change in signal
A rising edge or falling edge in a real input signal known as a trigger signal. If the signal is a nonscalar array, the subsystem executes once if any position in the array has an edge of the kind you specify (Rising, Falling, or Either).	Trigger

Note To call a subsystem upon an entity departure or upon a function call, see “Creating Entity-Departure Subsystems” on page 10-27 or “Using Function-Call Subsystems” on page 10-33, respectively.

When the Subsystem Executes Multiple Times

Each Discrete Event Inport block can potentially execute the subsystem at most once per sample time hit, regardless of whether the signal is scalar or nonscalar. Distinct Discrete Event Inport blocks can execute the subsystem independently of each other.

For example, in the next schematic, suppose **Din** is a vector of length 2 that the application updates three times at $T=1$, and **Din1** is a scalar that the application updates four times at $T=1$. **Din** causes the subsystem to execute three times, and **Din1** independently causes the subsystem to execute four times. As a result, the subsystem executes a total of seven times at $T=1$.



Comparison of Event Types for Discrete Event Subsystems

Here are some points to keep in mind when deciding which type of signal-based event should call your discrete event subsystem:

- If you want one of the inport blocks to provide an input signal without affecting the times at which the subsystem executes, clear the **Execute subsystem upon signal-based event** check box on that inport block. Depending on your model, preventing unwanted subsystem executions might be an optimization or an important modeling decision that prevents an incorrect computation. See these examples of input signals that do not cause subsystems to execute:
 - “Example: Detecting Changes from Empty to Nonempty” on page 10-24
 - “Example: Logging Data About the First Entity on a Path” on page 10-25
 - “Example: Detecting Changes in the Last-Updated Signal” on page 16-35

However, always select **Execute subsystem upon signal-based event** for at least one inport block of the subsystem or else the subsystem will never execute.

- Value changes are similar to sample time hits, except that a sample time hit might cause a signal to be updated with the same value. If you expect that calling the subsystem for repeated values of an input signal would produce incorrect numerical results or would be wasteful, execute the subsystem upon changes in the signal value rather than upon every sample time hit.
- The Discrete Event Subsystem block is similar to the ordinary Triggered Subsystem block in the Simulink Ports & Subsystems library. However, the Discrete Event Subsystem block can detect zero-duration values in the input signal, which are signal values that do not persist for a positive duration. (See “Working with Multivalued Signals” on page 4-19 for details on zero-duration values.) Unlike the Triggered Subsystem block, the Discrete Event Subsystem block can detect and respond to a trigger edge

formed by a zero-duration value, as well as multiple edges in a single instant of time.

- Sample time hits can occur for real or complex signals, while triggers and value changes (increasing or decreasing) are not supported for complex signals.

For more information about signal-based events, see “Types of Supported Events” on page 2-2.

Examples Using Discrete Event Subsystem Blocks

In this section...

“Example: Comparing the Lengths of Two Queues” on page 10-18

“Example: Normalizing a Statistic to Use for Routing” on page 10-19

“Example: Ending the Simulation Upon an Event” on page 10-21

“Example: Sending Unrepeated Data to the MATLAB Workspace” on page 10-22

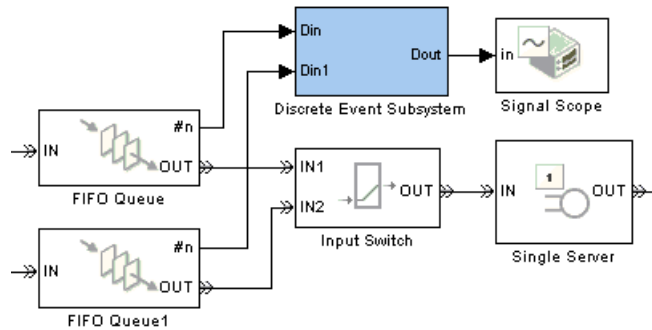
“Example: Focusing on Events, Not Values” on page 10-23

“Example: Detecting Changes from Empty to Nonempty” on page 10-24

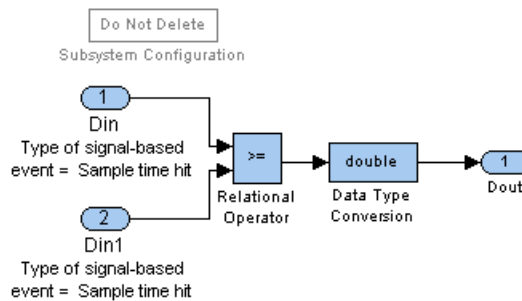
“Example: Logging Data About the First Entity on a Path” on page 10-25

Example: Comparing the Lengths of Two Queues

In a model containing two queues, a logical comparison of the lengths of the queues changes when either queue has an arrival or departure. A queue block's **#n** output signal is updated after each arrival if the queue is nonempty, and after each departure. By contrast, the Relational Operator block is a time-based block. The next model performs the comparison inside a discrete event subsystem whose Discrete Event Inport blocks both have **Type of signal-based event** set to `Sample time hit`. This way, the comparison occurs whenever either **#n** signal is updated. If both queues update their **#n** values at the same time on the simulation clock, the discrete event subsystem is called twice at that time.



Top-Level Model



Subsystem Contents

Example: Normalizing a Statistic to Use for Routing

Suppose you want to make a routing decision based on an output signal from a SimEvents block, but you must manipulate or normalize the statistic so that the routing block receives a value in the correct range. In the next model, the **util** output signal from the Single Server block assumes real values between 0 and 1, while the Output Switch block expects values of 1 or 2 in the attribute of each arriving entity. The discrete event subsystem adds 1 to the rounded utilization value.

The Discrete Event Inport block inside the subsystem has **Type of signal-based event** set to **Sample time hit** so that the computation occurs whenever the server block updates the value of the utilization signal. Also, the Output Switch block uses the **Store entity before switching** option.

Together, the configurations of the subsystem and switch ensure that the routing decision uses the most up-to-date value of the **util** signal.

At a time instant at which an entity departs from the Single Server block, the sequence of operations is as follows:

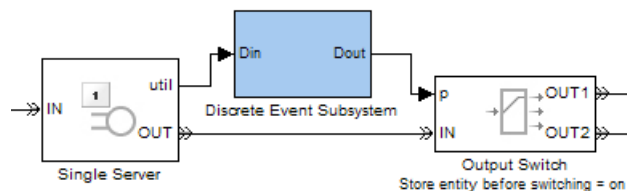
- 1 The entity advances from the Single Server block to the storage location in the Output Switch block.
- 2 The Single Server block updates its **util** output signal.
- 3 The subsystem reacts to the updated value of **util** by computing an updated value at the **p** input port of the Output Switch block.

The reaction is immediate because the **Resolve simultaneous signal updates according to event priority** parameter is not selected in the Discrete Event Inport block inside the subsystem. To learn more, see “Overview of Simultaneous Events” on page 3-2.

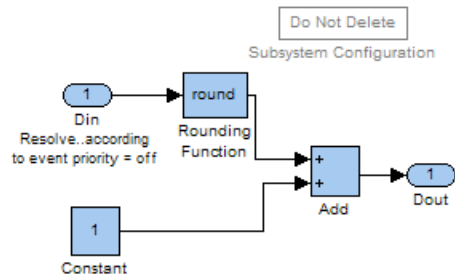
- 4 If the updated value of the **p** signal differs from its previous value, the Output Switch block selects a new entity output port.

Ensuring that the port selection is up-to-date before the entity departs from the switch is the reason for the **Store entity before switching** option. See “Using the Storage Option to Prevent Latency Problems” on page 6-2 for details.

- 5 The entity departs from the selected entity output port of the Output Switch block.



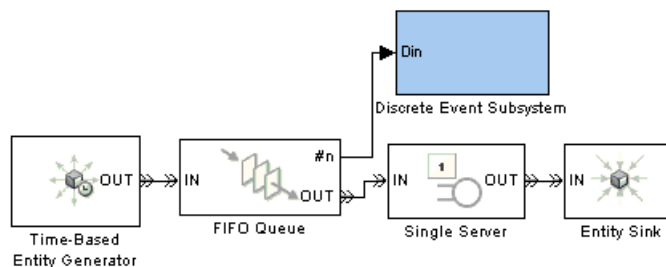
Top-Level Model



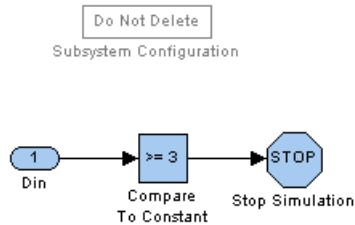
Subsystem Contents

Example: Ending the Simulation Upon an Event

This example ends the simulation as described in “Timing for the End of the Simulation” on page 10-2, precisely when the number of entities in a queue first equals or exceeds a threshold. In the next model, the Compare To Constant and Stop Simulation blocks are in a discrete event subsystem. The Discrete Event Inport block inside the subsystem has **Type of signal-based event** set to **Sample time hit** so that the subsystem executes at exactly those times when the FIFO Queue block updates the value of the queue length signal.



Top-Level Model

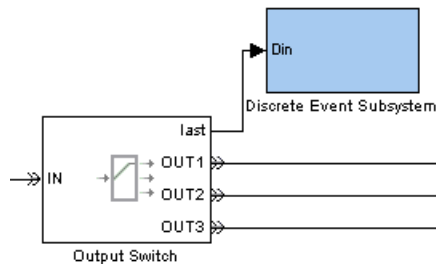


Subsystem Contents

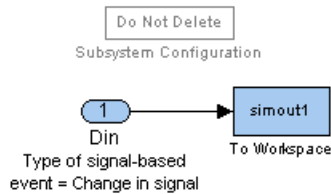
Example: Sending Unrepeated Data to the MATLAB Workspace

Suppose you want to log statistics to the MATLAB workspace, but you want to save simulation time and memory by capturing only values that are relevant to you. You might want to suppress repeated values, for example, or capture only values that represent an increase from the previous value.

In the next model, the discrete event subsystem contains a To Workspace block whose **Save format** parameter is set to **Structure With Time**. The Discrete Event Inport block inside the subsystem has **Type of signal-based event** set to **Change in signal** and **Type of change in signal value** set to **Either**, so that the MATLAB workspace variable tells you when the Output Switch block selects an entity output port that differs from the previously selected one. If, for example, the switch is configured to select the first port that is not blocked, then a change in the port selection indicates a change in the state of the simulation (that is, a previously blocked port has become unblocked, or a port becomes blocked that previously was not).



Top-Level Model



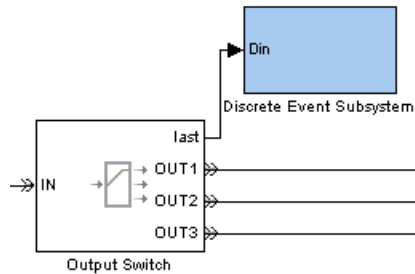
Subsystem Contents

Example: Focusing on Events, Not Values

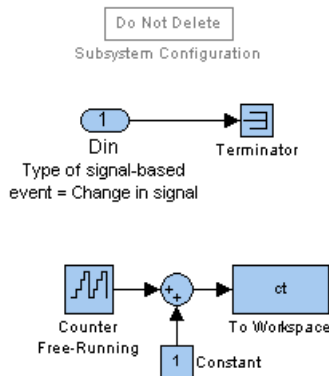
This example counts the number of times a signal changes its value, ignoring times at which the signal might be updated with the same value. The discrete event subsystem contains a Counter Free-Running block with an inherited sample time. Because the Counter Free-Running counts starting from 0, the subsystem also adds 1 to the counter output.

The Discrete Event Inport block inside the subsystem has **Type of signal-based event** set to **Change in signal** and **Type of change in signal value** set to **Either**, so that the subsystem is executed each time the input signal changes its value. In contrast to other subsystem examples, this subsystem does not use the signal's specific values for computations; the input signal is connected to a Terminator block inside the subsystem. The counter's value is what the subsystem sends to the MATLAB workspace.

In this example, avoiding extraneous calls to the subsystem is not merely a time-saver or memory-saver, but rather a strategy for producing the correct result.



Top-Level Model



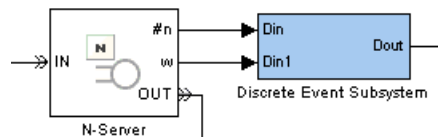
Subsystem Contents

Example: Detecting Changes from Empty to Nonempty

This example executes a subsystem only when an N-server changes from empty to nonempty, or vice versa, but not when the number of entities in the server remains constant or changes between two nonzero values. In the model, the N-Server block produces a **#n** signal that indicates the number of entities in the server. The server is empty if and only if the **#n** signal is 0. Connected to the **#n** signal is a Discrete Event Inport block inside the subsystem that has **Type of signal-based event** set to **Trigger** and **Trigger type** set to **Either**, so that the subsystem is executed each time the **#n** signal changes from 0 to 1 or from 1 to 0. Connected to the **w** signal is another Discrete Event Inport block inside the subsystem; this block has **Execute**

subsystem upon signal-based events cleared so that this signal does not cause additional calls to the subsystem; the subsystem merely uses the most recent value of the **w** signal whenever the **#n** signal exhibits a trigger edge.

Note Because the N-Server block updates the **w** signal before updating the **#n** signal, both signals are up to date when the trigger edge occurs.



If the server changes instantaneously from empty to nonempty and back to empty, the subsystem is called exactly twice in the same time instant, once for the rising edge and once for the subsequent falling edge. The Triggered Subsystem block might not detect the edges that the zero-duration value of 1 creates, and thus might not call the subsystem at that time instant. This is why the Discrete Event Subsystem block is more appropriate for this application.

Example: Logging Data About the First Entity on a Path

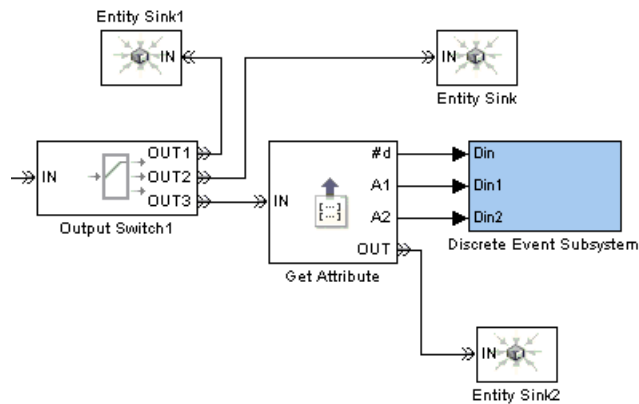
Suppose your model includes a particular entity path that entities rarely use, and you want to record certain attributes of the *first* entity that takes that path during the simulation. You can send the attribute values to the MATLAB workspace by using a To Workspace block inside a discrete event subsystem.

In the next model, the **#d** output signal from the Get Attribute block indicates how many entities have departed from the block. The other outputs from that block are the attribute values that you want to send to the workspace. Connected to the **#d** signal is a Discrete Event Inport block inside the subsystem that has **Type of signal-based event** set to **Trigger**, so that the subsystem is executed each time the **#d** signal changes from zero to one. Connected to the **A1** and **A2** signals are additional Discrete Event Inport blocks inside the subsystem. These blocks have **Execute subsystem upon signal-based events** cleared so that the attribute signals do not cause

additional calls to the subsystem; the subsystem merely uses the most recent value of the **A1** and **A2** signals whenever the **#d** signal exhibits a trigger edge.

The To Workspace block inside the subsystem does not actually create the variables in the workspace until the simulation ends, but the variable contents are correct because the timing of the subsystem corresponds to the time of the **#d** signal's first positive value.

Note Because the Get Attribute block updates the **A1** and **A2** signals before updating the **#d** signal, all signals are up to date when the trigger edge occurs.



Creating Entity-Departure Subsystems

In this section...

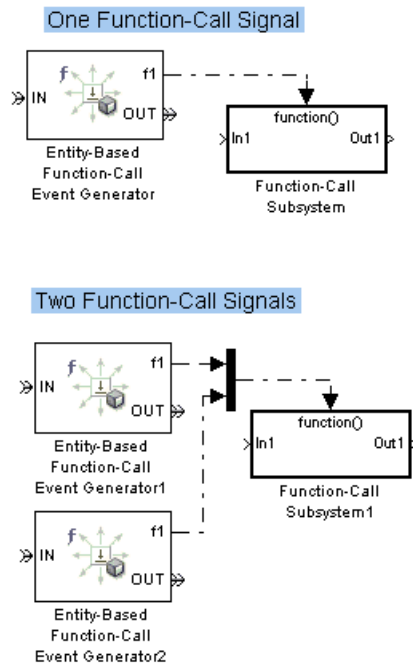
“Overview of Entity-Departure Subsystems” on page 10-27

“Accessing Blocks for Entity-Departure Subsystems” on page 10-28

“Setting Up Entity-Departure Subsystems” on page 10-29

Overview of Entity-Departure Subsystems

You can create a subsystem that executes only when an entity departs from a particular block in your model. The next figure shows a prototype, although most ports are not yet connected. The prototype uses the Entity-Based Function-Call Event Generator block to generate a function call when an entity departs. The function call executes the subsystem.



Prototype of Entity-Departure Subsystems

Accessing Blocks for Entity-Departure Subsystems

To create discrete event subsystems that respond to entity departures, use some or all of the blocks listed in this table.

Block	Library Location	Purpose
Entity-Based Function-Call Event Generator	Event Generators library in the SimEvents library set	Issues a function call corresponding to each entity departure.
Entity Departure Event to Function-Call Event	Event Translation library in the SimEvents library set	
Mux	Signal Routing library in the Simulink library set	Combines multiple function-call signals into a single function-call signal, if needed.
Function-Call Subsystem	Ports & Subsystems library in the Simulink library set	Contains blocks to execute upon each function call. You must configure the subsystem to propagate its execution context, as described in “Creating Entity-Departure Subsystems” on page 10-27.
Inport	Ports & Subsystems library in the Simulink library set	Links a subsystem to its parent system.
Outputport		

Setting Up Entity-Departure Subsystems

To create subsystems that respond to entity departures, follow this procedure:

- 1** Insert and configure the Function-Call Subsystem block as described in “Setting Up Function-Call Subsystems in SimEvents Models” on page 10-34.
- 2** Insert one or more of these blocks into your model. The first is easier to use but less flexible.
 - Entity-Based Function-Call Event Generator
 - Entity Departure Event to Function-Call Event

Note You can configure these blocks to issue a function call either before or after the entity departs. In most situations, the **After entity departure** option is more appropriate. The **Before entity departure** option can be problematic if a subsystem is executed too soon, but this option is an appropriate choice in some situations.

- 3** Connect the newly inserted blocks to indicate which entity departures should call the subsystem. If entity departures from multiple blocks should call the subsystem, combine multiple function-call signals using a Mux block.

Examples Using Entity-Departure Subsystems

In this section...

“Example: Using Entity-Based Timing for Choosing a Port” on page 10-30

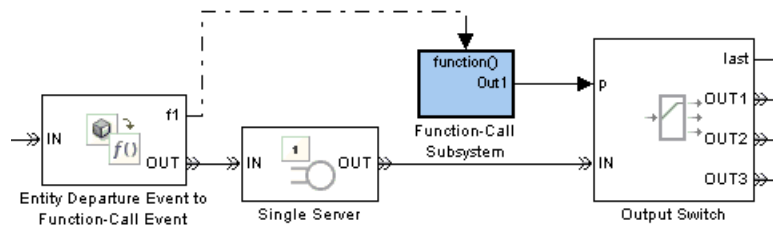
“Example: Performing a Computation on Selected Entity Paths” on page 10-31

Example: Using Entity-Based Timing for Choosing a Port

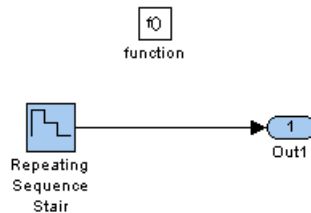
This example performs the entity-based routing described in “Timing for Choosing a Port Using a Sequence” on page 10-4. The example routes entities by establishing a sequence of paths and then choosing a number from that sequence for each entity that arrives at the routing block. This is the situation shown in the figure *Departure Port Changes with Each Entity* on page 10-6.

In the next model, the Function-Call Subsystem block contains a Repeating Sequence Stair block whose **Sample time** parameter is set to -1 (inherited). Any entity that arrives at the Output Switch block previously departed from the Entity Departure Event to Function-Call Event block. The function-call output from that block caused the subsystem to produce a number that indicates which entity output port the entity uses when it departs from the Output Switch block.

If you used the Repeating Sequence Stair block with an explicit sample time and not inside a subsystem, then the routing behavior would depend on the clock, as shown in the figure *Departure Port Changes with Time* on page 10-5, rather than on entity departures.



Top-Level Model

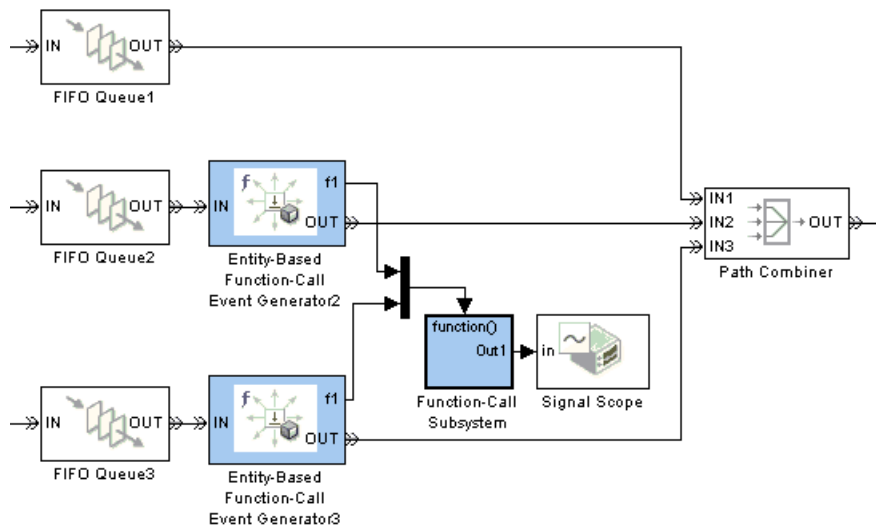


Subsystem Contents

The Entity Departure Event to Function-Call Event block, which issues function calls after entity departures, appears before the Single Server block instead of between the Single Server block and the Output Switch block. This placement ensures that when the function call executes the subsystem, the entity has not yet arrived at the switch but instead is stored in the server.

Example: Performing a Computation on Selected Entity Paths

The next model performs a computation whenever an entity arrives at the **IN2** or **IN3** entity input port of a Path Combiner block, but not when an entity arrives at the **IN1** port of the Path Combiner block. The computation occurs inside the Function-Call Subsystem block. When an entity departs from specific blocks that precede the Path Combiner block, the corresponding Entity-Based Function-Call Event Generator block issues a function call. A Mux block combines the two function-call signals, creating a function-call signal that calls the subsystem. If both event generators issue a function call at the same value of the simulation clock, the subsystem is called twice at that time.



Using Function-Call Subsystems

In this section...

“Using Function-Call Subsystems in Discrete-Event Simulations” on page 10-33

“Use Cases for Function-Call Subsystems” on page 10-33

“Setting Up Function-Call Subsystems in SimEvents Models” on page 10-34

Using Function-Call Subsystems in Discrete-Event Simulations

The most general kind of discrete event subsystem is an appropriately configured Function-Call Subsystem block, where the appropriate configuration requires selecting the **Propagate execution context across subsystem boundary** option as a subsystem property. You can execute such a subsystem at the exact time of an input function call, whether the function call comes from a Stateflow block, a block in the Event Generators library, a block in the Event Translation library, or the Function-Call Generator block.

Use Cases for Function-Call Subsystems

The Discrete Event Subsystem block and the entity-departure subsystems discussed in “Working with Discrete Event Subsystem Blocks” on page 10-11 and “Creating Entity-Departure Subsystems” on page 10-27, respectively, are special cases of the Function-Call Subsystem block configured as a discrete event subsystem. You might require the additional flexibility provided by the Function-Call Subsystem approach if you want to execute the subsystem upon:

- The logical OR of multiple event occurrences, where the events can come from any combination of a Stateflow block, another source of function calls, or a signal-based event. To do this, use the Mux block to combine multiple function-call signals into one function-call signal.

For an example, see “Example: Executing a Subsystem Based on Multiple Types of Events” on page 2-37.

- The logical AND of an event occurrence and some underlying condition. To do this, use blocks in the Event Translation library and select **Suppress**

function call f1 if enable signal e1 is not positive (or the similar option for the **f2** and **e2** ports).

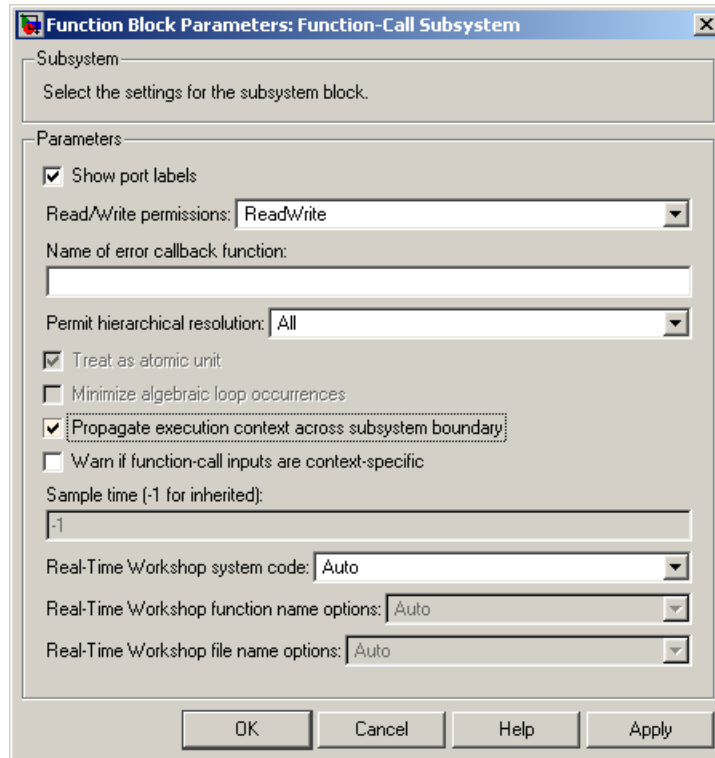
For an example, see “Example: Detecting Changes in the Last-Updated Signal” on page 16-35.

Setting Up Function-Call Subsystems in SimEvents Models

To use a Function-Call Subsystem block in a model that uses event-based blocks or event-based signals, follow the next procedure.

Note Selecting the **Propagate execution context across subsystem boundary** option is particularly important for ensuring that the subsystem executes at the correct times.

- 1** Drag the Function-Call Subsystem block from the Simulink Ports & Subsystems library into your model. Initially, it shows one signal input port **In1**, one output signal port **Out1**, and a control port **function()**. These are signal ports, not entity ports, because the subsystem is designed to process signals rather than entities.
- 2** Select the subsystem block and choose **Edit > Subsystem Parameters** from the model window’s menu bar.
- 3** In the Block Parameters dialog box, select **Propagate execution context across subsystem boundary** and click **OK**.



- 4 In the model window, double-click the Function-Call Subsystem block to open the subsystem it represents. Initially, the subsystem contains an Inport block connected to an Outport block. These blocks are from the Simulink Ports & Subsystems library, and are not the same as the Discrete Event Inport and Discrete Event Outport blocks in the SimEvents Ports and Subsystems library. The subsystem also contains a block labeled “function.”
- 5 To change the number of inputs or outputs to the subsystem, change the number of Inport and Outport blocks in the subsystem window. You can copy, paste, and delete these blocks.

Note Do not change the **Initial output** parameter in the Outport. If you need to define an initial output value for the signal, connect the Initial Value block to the output signal at the upper level of the model, outside the subsystem.

- 6 Drag other blocks into the subsystem window as appropriate to build the subsystem. This step is similar to the process of building the top level of hierarchy in your model, except that only certain types of blocks are suitable for use inside the subsystem. See “Blocks Inside Discrete Event Subsystems” on page 10-10 for details.

Plotting Data

- “Choosing and Configuring Plotting Blocks” on page 11-2
- “Working with Scope Plots” on page 11-8
- “Using Plots for Troubleshooting” on page 11-10
- “Example: Plotting Entity Departures to Verify Timing” on page 11-11
- “Example: Plotting Event Counts to Check for Simultaneity” on page 11-15
- “Comparison with Time-Based Plotting Tools” on page 11-17

Choosing and Configuring Plotting Blocks

In this section...
“Sources of Data for Plotting” on page 11-2
“Inserting and Connecting Scope Blocks” on page 11-3
“Connections Among Points in Plots” on page 11-4
“Varying Axis Limits Automatically” on page 11-5
“Caching Data in Scopes” on page 11-6
“Examples Using Scope Blocks” on page 11-6

Sources of Data for Plotting

The table below indicates the kinds of data you can plot using various combinations of blocks and parameter values. To view or set the parameters, open the dialog box using the Parameters toolbar button in the plot window.

Data	Block	Parameter
Scalar signal vs. time	Signal Scope	X value from = Event time
Scalar signal values without regard to time	Signal Scope	X value from = Index
Two scalar signals (X-Y plot)	X-Y Signal Scope	
Attribute vs. time	Attribute Scope	X value from = Event time
Attribute values without regard to time	Attribute Scope	X value from = Index
Two attributes of same entity (X-Y plot)	X-Y Attribute Scope	
Attribute vs. scalar signal	Get Attribute block to assign the attribute value to a signal; followed by X-Y Signal Scope	
Scalar signal vs. attribute		

Data	Block	Parameter
Number of entity arrivals per time instant	Instantaneous Entity Counting Scope	
Number of events per time instant	Instantaneous Event Counting Scope	

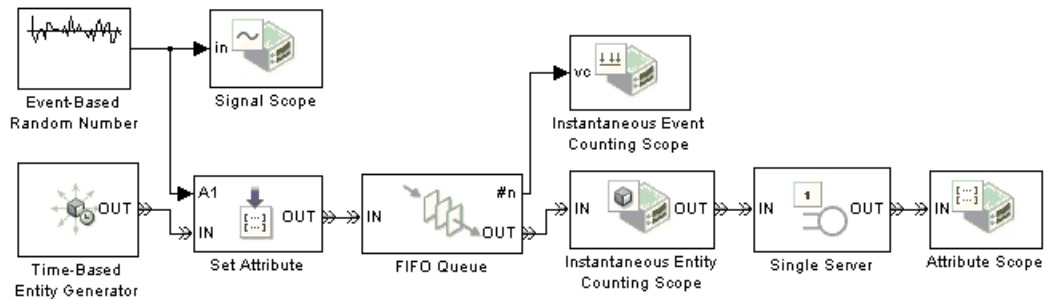
The Signal Scope and X-Y Signal Scope blocks are appropriate for data arising from discrete-event simulations because the plots can include zero-duration values. That is, the plots can include multiple values of the signal at a given time. By contrast, the Scope block in the Simulink Sources library plots at most one value of the signal at each time.

Inserting and Connecting Scope Blocks

The scope blocks reside in the SimEvents Sinks library. The table indicates the input ports on each scope block.

Block	Input Ports	Port Description
Signal Scope	One signal input port	Signal representing the data to plot
X-Y Signal Scope	Two signal input ports	Signals representing the data to plot
Attribute Scope	One entity input port	Entities containing the attribute value to plot
X-Y Attribute Scope	One entity input port	Entities containing the attribute values to plot
Instantaneous Entity Counting Scope	One entity input port	Entities whose arrivals the block counts
Instantaneous Event Counting Scope	One signal input port	Signal whose signal-based events or function calls the block counts

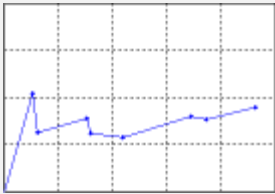
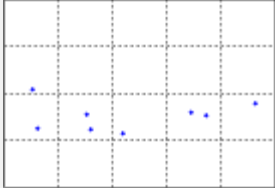
The following figure shows some typical arrangements of scope blocks in a model. Notice that the blocks that have entity input ports can have optional entity output ports, and that signal lines can branch whereas entity connection lines cannot.



Connections Among Points in Plots

You can configure certain scope blocks to determine whether and how it connects the points that it plots. The following table indicates the options. To view or change the parameter settings, open the dialog box using the Parameters toolbar button in the plot window.

Connection Characteristics	Setting	Sample Plot
Stairstep across, then up or down. Also known as a zero-order hold.	Plot type = Stair in the block dialog box	
Vertical line from horizontal axis to point. No connection with previous or next plotted point. Also known as a stem plot.	Plot type = Stem in the block dialog box	

Connection Characteristics	Setting	Sample Plot
Line segment from point to point. Also known as a first-order hold.	Plot type = Continuous in the block dialog box	
No connection with other points or with axis. Also known as a scatter plot.	Style > Line > None in the plot window	

Note If no initial condition, data value, or arriving entity indicates a value to plot at $T=0$, the plot shows no point at $T=0$. In this case, the plot does not connect the first plotted point to the $T=0$ edge of the plot.

Varying Axis Limits Automatically

Using parameters on the **Axes** tab of the dialog box of scope blocks, you set the initial limits for the axes of the plot. Also, these parameters let you choose how the block responds when a point does not fit within the current axis limits:

- **If X value is beyond limit**
- **If Y value is beyond limit**

Choices for the parameters are in the table.

Option	Description
Stretch axis limits	Maintain one limit while doubling the size of the displayed interval (without changing the size of the containing plot window)
Keep axis limits unchanged	Maintain both limits, which means that points outside the limits do not appear
Shift axis limits	Maintain the size of the displayed interval while changing both limits

Other operations can still affect axis limits, such as the autoscale, zoom, and pan features.

To store the current limits of both axes for the next simulation, select **Axes > Save axes limits** from the plot window menu.

Caching Data in Scopes

The **Data History** tab of the dialog box of scope blocks lets you balance data visibility with simulation efficiency. Parameters on the **Data History** tab determine how much data the blocks cache during the simulation. Caching data lets you view it later, even if the scope is not open during the simulation. Caching less or no data accelerates the simulation and uses less memory.

If you set the **Store data when scope is closed** parameter to **Limited**, uncached data points disappear when:

- The simulation ends
- You interact with the plot after pausing the simulation (using **Simulation > Pause**, for example)

Examples Using Scope Blocks

The following examples use scope blocks to create different kinds of plots:

Example	Description
“Plotting the Queue-Length Signal” and “Observations from Plots” in the SimEvents getting started documentation	Stairstep and continuous plots of statistical signals
“Example: Round-Robin Approach to Choosing Inputs” in the SimEvents getting started documentation	Stem plot of data from an attribute
“Example: Using Servers in Shifts” on page 7-11	Unconnected plot of a signal using dots
“Example: Setting Attributes” on page 1-10	Stairstep plots of data from attributes using Attribute Scope blocks as sinks
“Example: Synchronizing Service Start Times with the Clock” on page 8-6	Stem plots that count entities using Instantaneous Entity Counting Scope blocks with entity output ports
X-Y Signal Scope reference page	Continuous plot of two signals
X-Y Attribute Scope reference page	Unconnected plot of two attributes using x’s as plotting markers

Working with Scope Plots

In this section...
“Customizing Plots” on page 11-8
“Exporting Plots” on page 11-9

Customizing Plots

After a scope block opens its plot window, you can modify several aspects of the plot by using the menu and toolbar of the plot window:

- **Axes > Autoscale** resizes both axes to fit the range of the data plus some buffer space.
- The Zoom In and Zoom Out toolbar buttons change the axes as described in the MATLAB documentation about zooming in 2-D views.
- The Pan toolbar button moves your view of a plot.
- The **Style** menu lets you change the line type, marker type, and color of the plot. (You can also select **Style > Line > None** to create a plot of unconnected points.) Your changes become part of the block configuration and persist across sessions when you save the model.
- **Axes > Save axes limits** updates the following parameters on the **Axes** tab of the block dialog box to reflect the current limits of the axes:
 - **Initial X axis lower limit**
 - **Initial X axis upper limit**
 - **Initial Y axis lower limit**
 - **Initial Y axis upper limit**
- **Axes > Save position** updates the **Position** parameter on the **Figure** tab of the block dialog box to reflect the current position and size of the window.

Note Some menu options duplicate the behavior of a parameter in the block dialog box. In this case, selecting the menu option *replaces* the corresponding parameter value in the dialog. You can still edit the parameter values in the dialog manually. An example of a duplicate pair of menu option and dialog box parameter is **Show grid**.

Exporting Plots

The Save Figure toolbar button lets you export the current state of the plot to a file:

- Exporting to a FIG-file enables you to reload it in a different MATLAB software session. Reloading the file opens a new plot. The new plot is *not* associated with the original scope block. The new plot does not offer the same menu and toolbar options as in the original plot window.
- Exporting to a graphics file enables you to insert the graphic into a document.

Using Plots for Troubleshooting

Here are typical ways to use plotting blocks in the SimEvents Sinks library to troubleshoot problems.

Technique	Example
Check when an entity departs from the block. To do this, plot the #d output signal of the block.	“Example: Plotting Entity Departures to Verify Timing” on page 11-11
Check whether operations such as service completion or routing are occurring as you expect. To do this, plot statistical output signals such as pe or last , if applicable.	“Example: Using Servers in Shifts” on page 7-11 and “Timing for Choosing a Port Using a Sequence” on page 10-4
Check whether a block uses a control signal as you expect. To do this, plot input signals such as port selection, service time, or intergeneration time, and compare the values with observations of how the corresponding blocks use those signals.	“Example: Choices of Values for Event Priorities” on page 3-11
Check how long entities spend in a region of the model. To do this, plot the output of a Read Timer block.	“Example: M/M/5 Queuing System” on page 5-18
Check whether events you expect to be simultaneous are, in fact, simultaneous. To do this, use the Instantaneous Entity Counting Scope or Instantaneous Event Counting Scope block.	“Example: Counting Simultaneous Departures from a Server” on page 1-21 and “Example: Plotting Event Counts to Check for Simultaneity” on page 11-15

Example: Plotting Entity Departures to Verify Timing

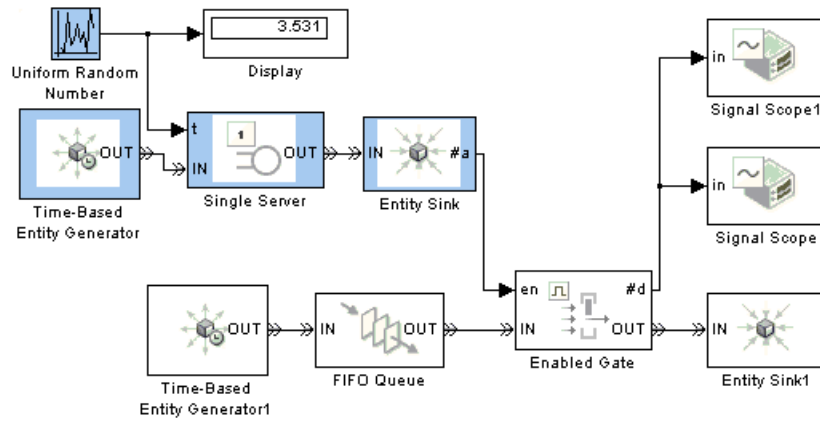
In this section...
“Overview of Example” on page 11-11
“Model Exhibiting Correct Timing” on page 11-11
“Model Exhibiting Latency” on page 11-13

Overview of Example

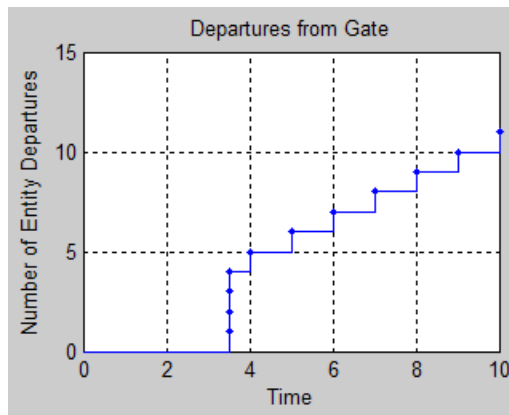
This example compares two methods of opening a gate at a random time and leaving the gate open for the rest of the simulation. The Signal Scope block lets you see when the gate opens, to check whether the timing is what you expected. One method exhibits latency in the opening of the gate.

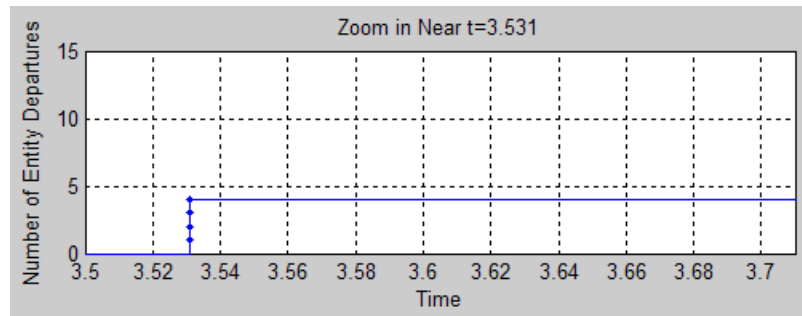
Model Exhibiting Correct Timing

The following model views the random opening of the gate as a discrete event, and models it via an entity departure from a server at a random time. The Time-Based Entity Generator block generates exactly one entity, at $T=0$. The Single Server block delays the entity for the amount of time indicated by the Uniform Random Number block, 3.531 s in this case. At $T=3.531$, the entity arrives at the Entity Sink block. This time is exactly when the #a signal of the sink block changes from 0 to 1, which in turn causes the gate to open.



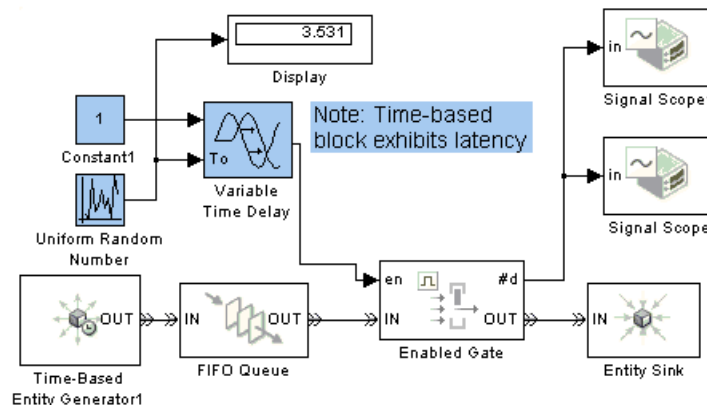
By using the zoom feature of the scope, you can compare the time at which entities depart from the Enabled Gate block with the random time shown on the Display block in the model.



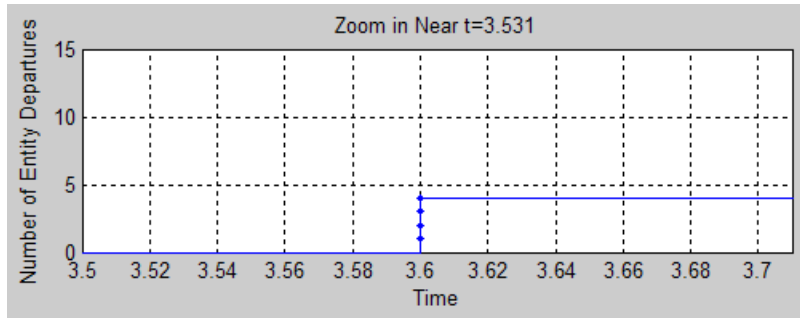
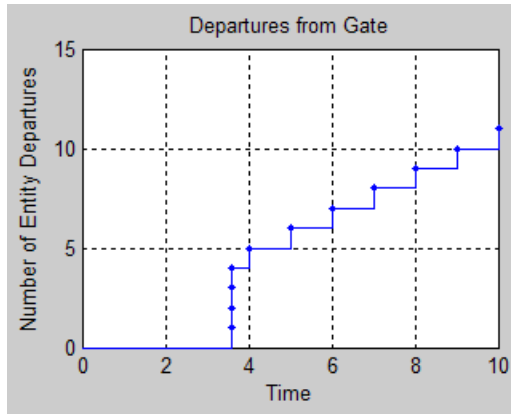


Model Exhibiting Latency

The model below uses the Variable Time Delay block to create a step signal that is intended to change from 0 to 1 at a random time. However, because the Variable Time Delay block is time-based, it updates its output signal at times dictated by the time-based simulation clock. The step signal does not actually change from 0 to 1 until the next sample time hit after the time indicated by the random number. This is the intentional documented behavior of this time-based block.



By using the zoom feature of the scope, you can see that entities depart from the Enabled Gate block later than the random time shown on the Display block in the model.

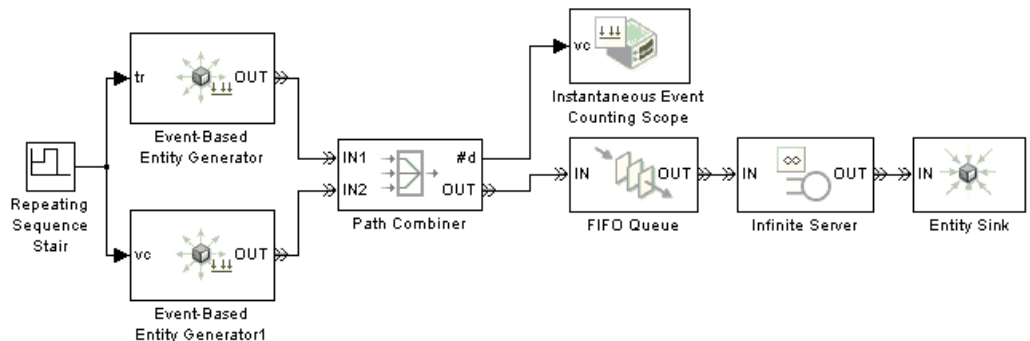


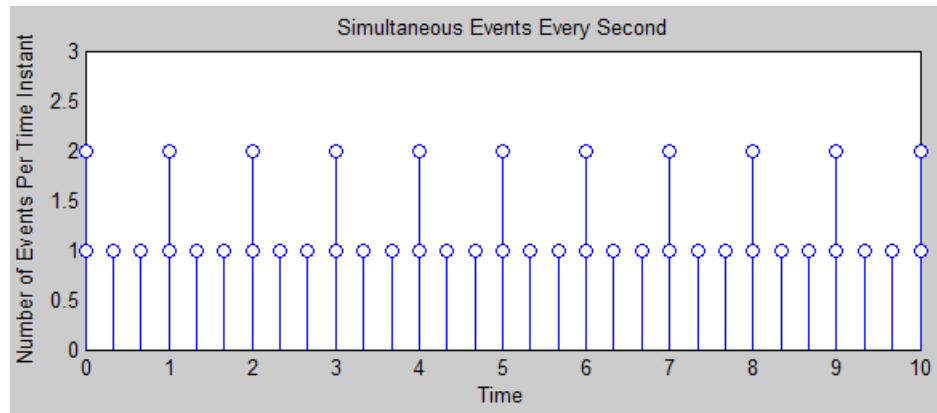
Example: Plotting Event Counts to Check for Simultaneity

The example below suggests how to use the Instantaneous Event Counting Scope block to determine whether events you want to be simultaneous are truly simultaneous.

Suppose you want two entity generators with periods of 1 and $1/3$ to create simultaneous entity departures every second, so that event priorities determine which entity arrives at the queue first. By counting events at each value of time and checking when the count is 2, you can confirm that two entity generation events are truly simultaneous.

The model below uses two Event-Based Entity Generator blocks receiving the same input signal. You can see from the plot that simultaneous events occur every second, as desired.



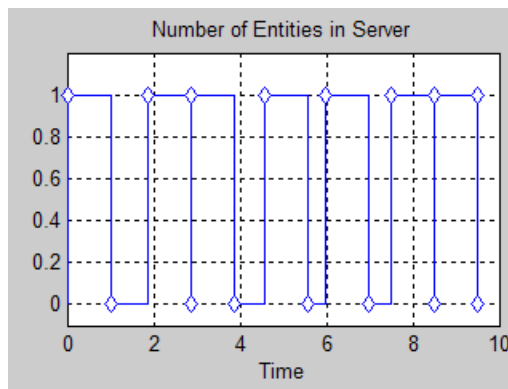


Although this example uses the Instantaneous Event Counting Scope to plot a **#d** signal, you can alternatively use the Instantaneous Entity Counting Scope to count entities departing from the Path Combiner block.

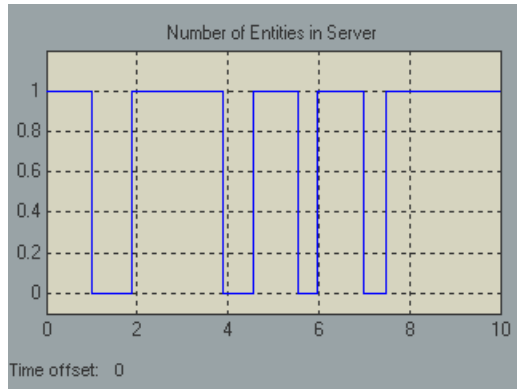
Comparison with Time-Based Plotting Tools

Simulink software offers plotting tools designed for signals in time-based simulations. Examples are the Scope block, XY Graph block, Signal & Scope Manager, and `simplot` function. In general, you should plot event-based signals using event-based tools such as blocks in the SimEvents Sinks library. Time-based plotting tools plot at most one value at each time, whereas blocks in the SimEvents Sinks library can include zero-duration values. Time-based plotting tools might also display the event-based signal with some latency.

Compare the two figures below, which depict the same data, when you consider which plotting tools are more appropriate in your event-based simulations.



Discrete-Event Plot of Number of Entities in a Server



Time-Based Plot of Number of Entities in a Server

Using Statistics

- “Role of Statistics in Discrete-Event Simulation” on page 12-2
- “Accessing Statistics from SimEvents Blocks” on page 12-5
- “Deriving Custom Statistics” on page 12-8
- “Using Timers” on page 12-20
- “Varying Simulation Results by Managing Seeds” on page 12-28
- “Regulating the Simulation Length” on page 12-35

Role of Statistics in Discrete-Event Simulation

In this section...
“Overview” on page 12-2
“Statistics for Data Analysis” on page 12-2
“Statistics for Run-Time Control” on page 12-3

Overview

Most SimEvents blocks can producing one or more statistical output signals. You can use these signals to gather data from the simulation or to influence the dynamics of the simulation. This section gives an overview of both purposes, in these topics:

The rest of this chapter illustrates some modeling and analysis techniques that you can use in SimEvents models. However, a detailed treatment of statistical analysis is well beyond the scope of this User’s Guide; see the works listed in “Selected Bibliography” for more information.

Statistics for Data Analysis

Often, the purpose of creating a discrete-event simulation is to improve understanding of the underlying system being modeled or to use simulation results to help make decisions about the underlying system. Numerical results gathered during simulation can be important tools.

For example, if you simulate the operation and maintenance of equipment on an assembly line, then you might use the computed production and defect rates to help decide whether to change your maintenance schedule. As another example, if you simulate a communication bus under varying bus loads, then you might use computed average delays in high- or low-priority messages to help determine whether a proposed architecture is viable.

Just as you decide how to design a simulation model that adequately describes the underlying system, you decide how to design the statistical measures that you will use to learn about the system. Some questions to consider are

- Which statistics are meaningful for your investigation or decision? For example, if you are trying to maximize efficiency, then what is an appropriate measure of efficiency in your system? As another example, does a mean give the best performance measure for your system, or is it also worthwhile to consider the proportion of samples in a given interval?
- How can you compute the desired statistics? For example, do you need to ignore any transient effects, does the choice of initial conditions matter, and what stopping criteria are appropriate for the simulation?
- To ensure sufficient confidence in the result, how many simulation runs do you need? One simulation run, no matter how long, is still a single sample and probably inadequate for valid statistical analysis.

To learn how to perform a series of simulation runs programmatically, see Chapter 15, “Running Discrete-Event Simulations Programmatically”.

For details concerning statistical analysis and variance reduction techniques, see the works [7], [4], [1], and [2] listed in “Selected Bibliography” in the SimEvents getting started documentation.

Statistics for Run-Time Control

Some systems rely on statistics to influence the dynamics. For example, a queuing system with discouraged arrivals has a feedback loop that adjusts the arrival rate throughout the simulation based on statistics reported by the queue and server, as illustrated in the [Queuing System with Discouraged Arrivals](#) demo.

When you create simulations that use statistical signals to control the dynamics, you must have access to the current values of the statistics at key times throughout the simulation, not just at the end of the simulation. Some questions to consider while designing your model are:

- Which statistics are meaningful, and how should they influence the dynamics of the system?
- How can you compute the desired statistics at the right times during the simulation? It is important to understand when SimEvents blocks update each of their statistical outputs and when other blocks can access the updated values. This topic is discussed in Chapter 4, “Working with Signals”.

- Do you need to account for initial conditions or extreme values in any special way? For example, if your control logic involves the number of entities in a queue, then be sure that the logic is sound even when the queue is empty or full.
- Will small perturbations result in large changes in the system's behavior? When using statistics to control the model, you might want to monitor those statistics or other statistics to check whether the system is undesirably sensitive to perturbations.

Accessing Statistics from SimEvents Blocks

In this section...

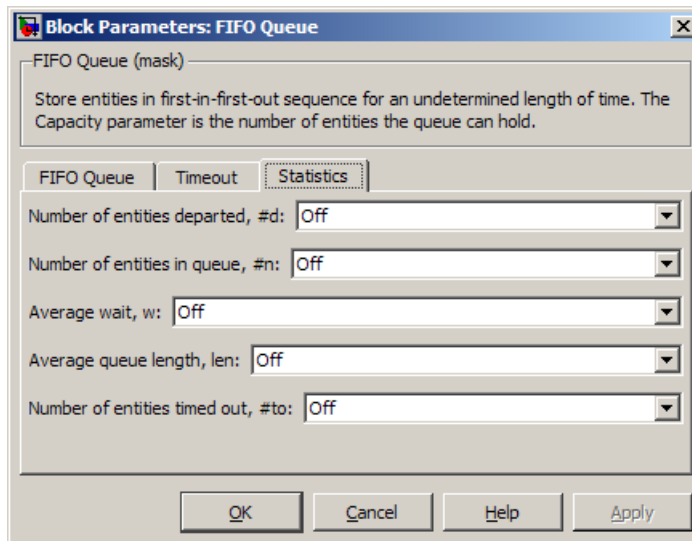
“Statistics-Related Parameters in Block Dialog Boxes” on page 12-5

“Accessing Statistics Throughout the Simulation” on page 12-6

“Accessing Statistics When Stopping or Pausing Simulation” on page 12-7

Statistics-Related Parameters in Block Dialog Boxes

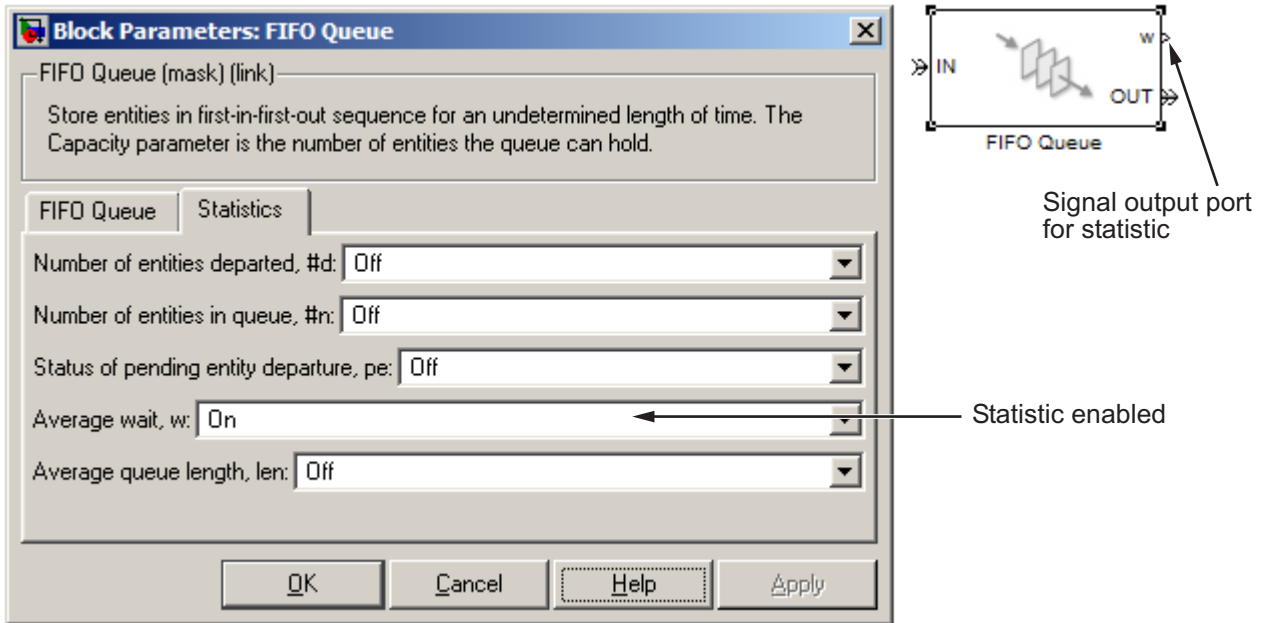
Most SimEvents blocks can produce one or more statistical outputs. To see which statistics are available, open the block’s dialog box. In most cases, the list of available statistics appears on the **Statistics** tab of the dialog box. For example, the figure below shows the **Statistics** tab of the FIFO Queue block’s dialog box.



In cases where the dialog box has no **Statistics** tab, such as the Entity Sink block, the dialog box has so few parameters that the statistical options are straightforward to locate.

Accessing Statistics Throughout the Simulation

To configure a block so that it outputs an available statistic throughout the simulation, set the corresponding parameter in the dialog box to On. After you apply the change, the block has a signal output port corresponding to that statistic. The figure below shows the dialog box and icon for the FIFO Queue block after the average wait statistic is enabled.



You can connect this signal output port to the signal input port of any block. For example, you can connect the statistical signal output port to

- A Signal Scope or X-Y Signal Scope block, to create a plot using the statistic.
- A Display block, which shows the statistic on the block icon throughout the simulation.
- A Discrete Event Signal to Workspace block, which writes the entire data set to the MATLAB workspace when the simulation stops or pauses. To learn more, see “Sending Data to the MATLAB Workspace” on page 4-15.

- A custom subsystem or computational block, for further data processing. See “Deriving Custom Statistics” on page 12-8 for some specific examples.

For more information about when SimEvents blocks update their statistical signals and when other blocks react to the updated values, see Chapter 4, “Working with Signals”.

Accessing Statistics When Stopping or Pausing Simulation

In some cases, you can configure a block to report a statistic only when the simulation stops or pauses. One way to pause a running simulation is to select **Simulation > Pause**. To configure the block in this way:

- 1 Find the dialog box parameter that corresponds to the given statistic.
- 2 Set it to **Upon stop or pause**, if available.

After you apply the change, the block has a signal output port corresponding to that statistic.

Because the statistic is not reported throughout the simulation, not all uses of the signal are appropriate. One appropriate use is in conjunction with a Discrete Event Signal to Workspace block with **Save format** set to **Structure With Time**.

Deriving Custom Statistics

In this section...

“Overview of Approaches to Custom Statistics” on page 12-8

“Graphical Block-Diagram Approach” on page 12-8

“Coded Approach” on page 12-9

“Post-Simulation Analysis” on page 12-9

“Example: Fraction of Dropped Messages” on page 12-9

“Example: Computing a Time Average of a Signal” on page 12-11

“Example: Resetting an Average Periodically” on page 12-13

Overview of Approaches to Custom Statistics

You can use the built-in statistical signals from SimEvents blocks to derive more specialized or complex statistics that are meaningful in your model. One approach is to compute statistics during the simulation within discrete event subsystems. Inside the subsystems, you can implement your computations using a graphical block-diagram approach, a nongraphical coded approach. Alternatively, you can compute statistics after the simulation is complete.

Graphical Block-Diagram Approach

The Math Operations library in the Simulink library set and the Statistics library in the Signal Processing Blockset™ library set can help you compute statistics using blocks. For examples using Simulink blocks, see

- “Example: Fraction of Dropped Messages” on page 12-9
- “Example: Detecting Changes in the Last-Updated Signal” on page 16-35, which computes the ratio of an instantaneous queue length to its long-term average
- The function-call subsystem within the DVS Optimizer subsystem in the Dynamic Voltage Scaling Using Online Gradient Estimation demo
- The Arrival Rate Estimation Computation subsystem within the Arrival Rate Estimator subsystem in the Building an Arrival Rate Estimator demo

Coded Approach

The blocks in the User-Defined Functions library in the Simulink library set can help you compute statistics using code. For examples using the Embedded MATLAB Function block, see

- “Example: Computing a Time Average of a Signal” on page 12-11
- “Example: Resetting an Average Periodically” on page 12-13

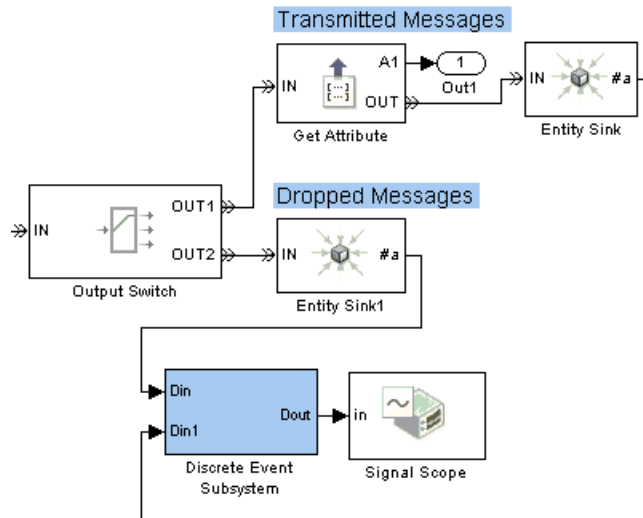
Note If you put an Embedded MATLAB Function block in a Discrete Event Subsystem block, use the Ports and Data Manager instead of Model Explorer to view or change properties such as the size or source of an argument. Model Explorer does not show the contents of Discrete Event Subsystem blocks.

Post-Simulation Analysis

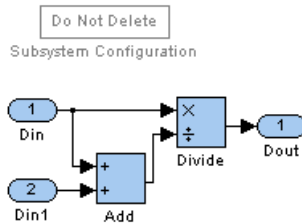
You can use the Discrete Event Signal to Workspace block to log data to the MATLAB workspace and compute statistics after the simulation is complete. For an example of post-simulation analysis, see “Example: Varying the Number of Servers Using MATLAB Code” on page 15-29.

Example: Fraction of Dropped Messages

The example below shows how to compute a ratio of event-based signals in a discrete event subsystem. The Output Switch block either transmits or drops the message corresponding to each entity. The goal is to compute the fraction of dropped messages, that is, the fraction of entities that depart via **OUT2** as opposed to **OUT1** of the Output Switch block.

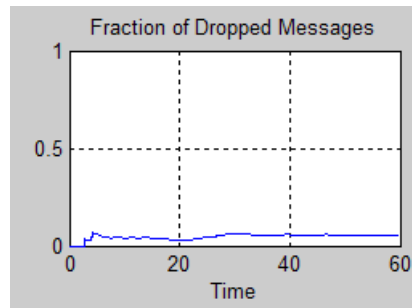


Upper-Level System



Subsystem Contents

Two Entity Sink blocks produce **#a** signals that indicate how many messages the communication link transmits or drops, respectively. The discrete event subsystem divides the number of dropped messages by the sum of the two **#a** signals. Because the discrete event subsystem performs the division only when one of the **#a** signals increases, no division-by-zero instances occur.

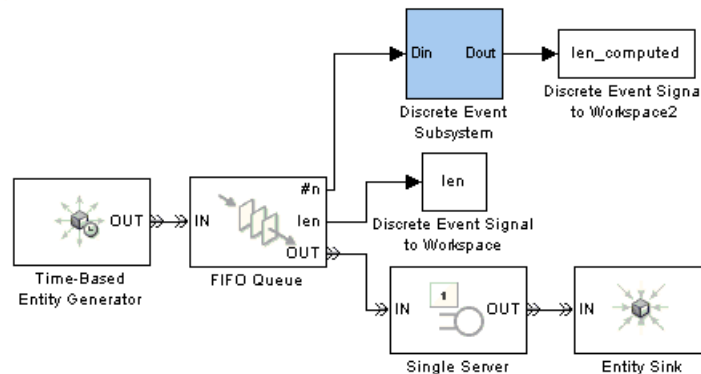


Example: Computing a Time Average of a Signal

This example illustrates how to compute a time average of a signal using the Embedded MATLAB Function block, and especially how to make the block retain data between calls to the function.

The model below implements a simple queuing system in which the FIFO Queue produces the output signals

- **#n**, the instantaneous length of the queue
- **len**, the time average of the queue length; this is the time average of **#n**.



Top-Level Model

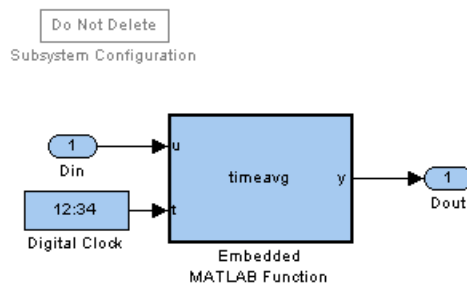
The discrete event subsystem uses **#n** to compute the time average. In this case, the time average should equal **len**. You can use a similar subsystem in your own models to compute the time averages of other signals.

Computation of the Time Average

In the example, the discrete event subsystem performs computations each time a customer arrives at or departs from the queue. Within the subsystem, an Embedded MATLAB Function block keeps a running weighted sum of the **#n** values that form the input, where the weighting is based on the length of time over which the signal assumes each value.

The block uses persistent variables for quantities whose values it must retain from one invocation to the next, namely, the running weighted sum and the previous values of the inputs.

Below are the subsystem contents and the function that the Embedded MATLAB Function block represents.



Subsystem Contents

```
function y = timeavg(u,t)
%TIMEAVG Compute time average of input signal U
%   Y = TIMEAVG(U,T) computes the time average of U,
%   where T is the current simulation time.

% Declare variables that must retain values between iterations.
persistent running_weighted_sum last_u last_t;
```



```

% Initialize persistent variables in the first iteration.
if isempty(last_t)
    running_weighted_sum = 0;
    last_u = 0;
    last_t = 0;
end

% Update the persistent variables.
running_weighted_sum = running_weighted_sum + last_u*(t-last_t);
last_u = u;
last_t = t;

% Compute the outputs.
if t > 0
    y = running_weighted_sum/t;
else
    y = 0;
end

```

Verifying the Result

After running the simulation, you can verify that the computed time average of **#n** is equal to **len**.

```

isequal([len.time, len.signals.values],...
        [len_computed.time, len_computed.signals.values])

```

The output indicates that the comparison is true.

```
ans =
```

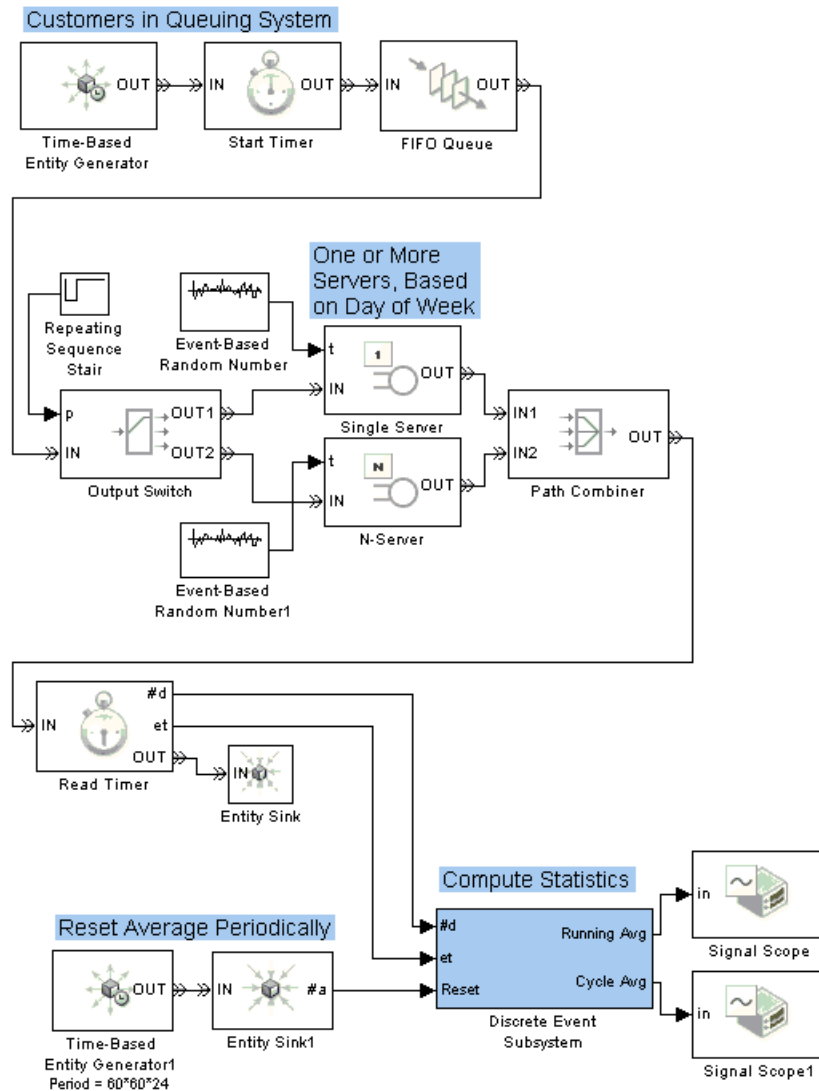
```
1
```

Example: Resetting an Average Periodically

This example illustrates how to compute a sample mean over each of a series of contiguous time intervals of fixed length, rather than the mean over the entire duration of the simulation. The example simulates a queuing system for 4 weeks' worth of simulation time, where customers have access to one server during the first 2 days of the week and five servers on the other days of

the week. The average waiting time for customers over a daily cycle depends on how many servers are operational that day. However, you might expect the averages taken over weekly cycles to be stable from one week to the next.

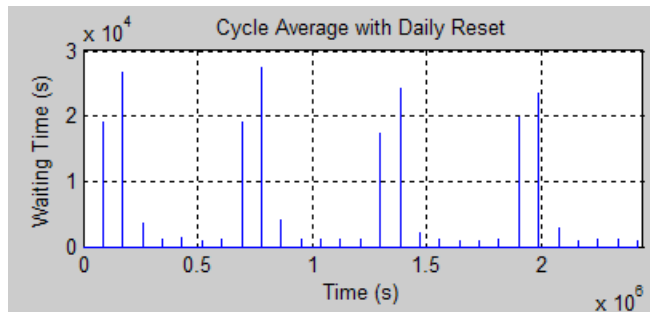
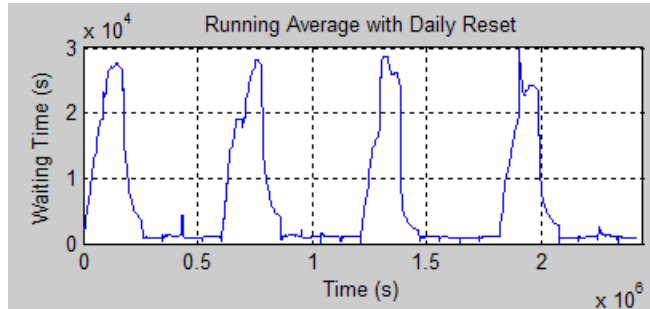
The model below uses a time-based Repeating Sequence Stair block to determine whether entities advance to a Single Server or N-Server block, thus creating variations in the number of operational servers. The Start Timer and Read Timer blocks compute each entity's waiting time in the queuing system. A computational discrete event subsystem processes the waiting time by computing a running sample mean over a daily or weekly cycle, as well as the final sample mean for each cycle. Details about this subsystem are in "Computation of the Cycle Average" on page 12-17.



Top-Level Model

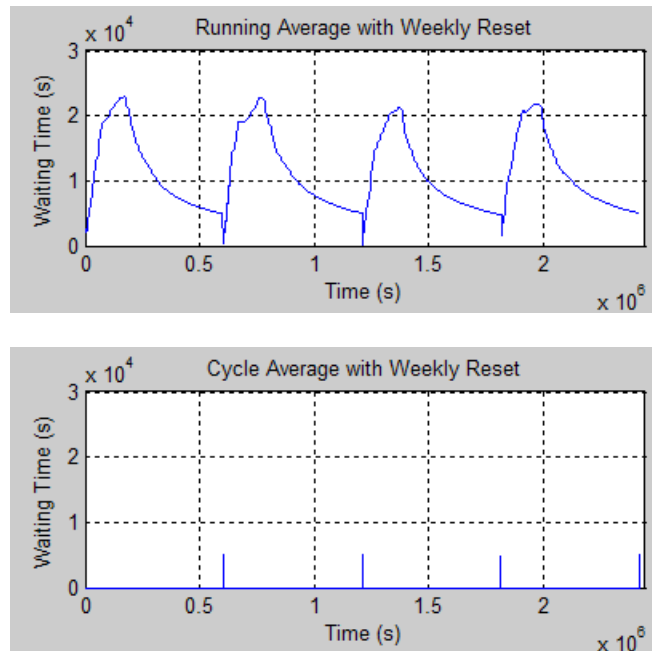
Performance of Daily Averages

When considering daily cycles, you can see that the cycle averages do not stabilize at a single value.



Performance of Weekly Averages

When considering weekly cycles, you can see less variation in the cycle averages because each cycle contains the same pattern of changing service levels. To compute the cycle average over a weekly cycle, change the **Period** parameter in the Time-Based Entity Generator1 block at the bottom of the model to $60 \times 60 \times 24 \times 7$, which is the number of seconds in a week.



Computation of the Cycle Average

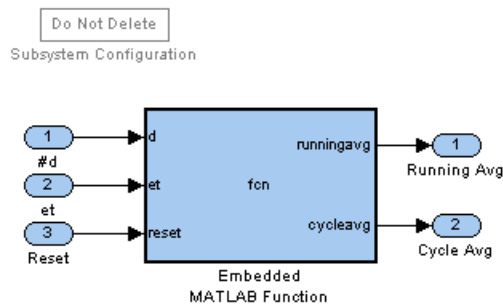
In the example, the discrete event subsystem performs computations each time a customer departs from the queuing system and at each boundary of a daily or weekly cycle. Within the subsystem, an Embedded MATLAB Function block counts the number of customers and the total waiting time among all customers at that point. The block resets these quantities to zero at each boundary of a cycle.

The block uses persistent variables for quantities whose values it must retain from one invocation to the next. The number of customers and total waiting time are important to retain for the computation of an average over time rather than an instantaneous statistic. Previous values of inputs are important to retain for comparison, so the function can determine whether it needs to update or reset its statistics.

The outputs of the Embedded MATLAB Function block are

- `runningavg`, the running sample mean of the input waiting times
- `cycleavg`, a signal that, at reset times, represents the sample mean over the cycle that just ended

Below are the subsystem contents and the function that the Embedded MATLAB Function block represents.



Subsystem Contents

```
function [runningavg, cycleavg] = fcn(d,et,reset)
%FCN    Compute average of ET, resetting at each update of RESET
% [RUNNINGAVG,CYCLEAVG] = FCN(D,ET,RESET) computes the average
% of ET over contiguous intervals. D is the number of samples
% of ET since the start of the simulation. Increases in
% RESET indicate when to reset the average.
%
% Assume this function is invoked when either D or RESET
% (but not both) increases. This is consistent with the
% behavior of the Discrete Event Subsystem block that contains
% this block in this example.
%
% RUNNINGAVG is the average since the start of the interval.
%
% At reset times, CYCLEAVG is the average over the interval
% that just ended; at other times, CYCLEAVG is 0.
```

```
% Declare variables that must retain values between iterations.
persistent total customers last_reset last_d;

% Initialize outputs.
cycleavg = 0;
runningavg = 0;

% Initialize persistent variables in the first iteration.
if isempty(total)
    total = 0;
    customers = 0;
    last_reset = 0;
    last_d = 0;
end

% If RESET increased, compute outputs and reset the statistics.
if (reset > last_reset)
    cycleavg = total / customers; % Average over last interval.
    runningavg = cycleavg; % Maintain running average.
    total = 0; % Reset total.
    customers = 0; % Reset number of customers.
    last_reset = reset;
end

% If D increased, then update the statistics.
if (d > last_d)
    total = total + et;
    customers = customers + 1;
    last_d = d;
    runningavg = total / customers;
end
```

Using Timers

In this section...
“Overview of Timers” on page 12-20
“Basic Example Using Timer Blocks” on page 12-21
“Basic Procedure for Using Timer Blocks” on page 12-22
“Timing Multiple Entity Paths with One Timer” on page 12-23
“Restarting a Timer from Zero” on page 12-24
“Timing Multiple Processes Independently” on page 12-26

Overview of Timers

Suppose you want to determine how long each entity takes to advance from one block to another, or how much time each entity spends in a particular region of your model. To compute these durations, you can attach a timer to each entity that reaches a particular spot in the model. Then you can

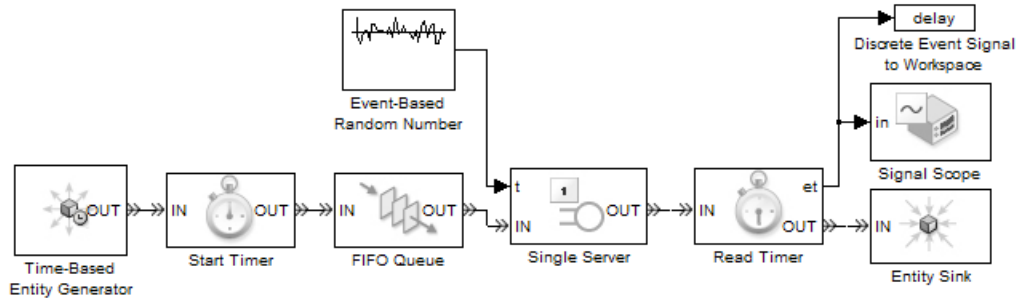
- Start the timer. The block that attaches the timer also starts it.
- Read the value of the timer whenever the entity reaches a spot in the model that you designate.
- Restart the timer, if desired, whenever the entity reaches a spot in the model that you designate.

The next sections describe how to arrange the Start Timer and Read Timer blocks to accomplish several common timing goals.

Note Timers measure durations, or relative time. By contrast, clocks measure absolute time. For details about implementing clocks, see the descriptions of the Clock and Digital Clock blocks in the Simulink documentation.

Basic Example Using Timer Blocks

A typical block diagram for determining how long each entity spends in a region of the model is in the figure below. The Start Timer and Read Timer blocks jointly perform the timing computation.



The model above measures the time each entity takes between arriving at the queue and departing from the server. The Start Timer block attaches, or associates, a timer to each entity that arrives at the block. Each entity has its own timer. Each entity's timer starts timing when the entity departs from the Start Timer, or equivalently, when the entity arrives at the FIFO Queue block. Upon departing from the Single Server block, each entity arrives at a Read Timer block. The Read Timer block reads data from the arriving entity's timer and produces a signal at the **et** port whose value is the instantaneous elapsed time for that entity. For example, if the arriving entity spent 12 seconds in the queue-server pair, then the **et** signal assumes the value 12.

Basic Example of Post-Simulation Analysis of Timer Data

The model above stores data from the timer in a variable called `delay` in the base MATLAB workspace. After running the simulation, you can manipulate or plot the data, as illustrated below.

```
% First run the simulation shown above, to create the variable
% "delay" in the MATLAB workspace.

% Histogram of delay values
edges = (0:20); % Edges of bins in histogram
counts = histc(delay.signals.values, edges); % Number of points per bin
figure(1); bar(edges, counts); % Plot histogram.
```

```
title('Histogram of Delay Values')

% Cumulative histogram of delay values
sums = cumsum(counts); % Cumulative sum of histogram counts
figure(2); bar(edges, sums); % Plot cumulative histogram.
title('Cumulative Histogram of Delay Values')
```

Basic Procedure for Using Timer Blocks

A typical procedure for setting up timer blocks is as follows:

- 1** Locate the spots in the model where you want to begin timing and to access the value of the timer.
- 2** Insert a Start Timer block in the model at the spot where you want to begin timing.
- 3** In the Start Timer block's dialog box, enter a name for the timer in the **Timer tag** field. This timer tag distinguishes the timer from other independent timers that might already be associated with the same entity.

When an entity arrives at the Start Timer block, the block attaches a named timer to the entity and begins timing.

- 4** Insert a Read Timer block in the model at the spot where you want to access the value of the timer.
- 5** In the Read Timer block's dialog box, enter the same **Timer tag** value that you used in the corresponding Start Timer block.

When an entity having a timer with the specified timer tag arrives at the block, the block reads the time from that entity's timer. Using the **Statistics** tab of the Read Timer block's dialog box, you can configure the block to report this instantaneous time or the average of such values among all entities that have arrived at the block.

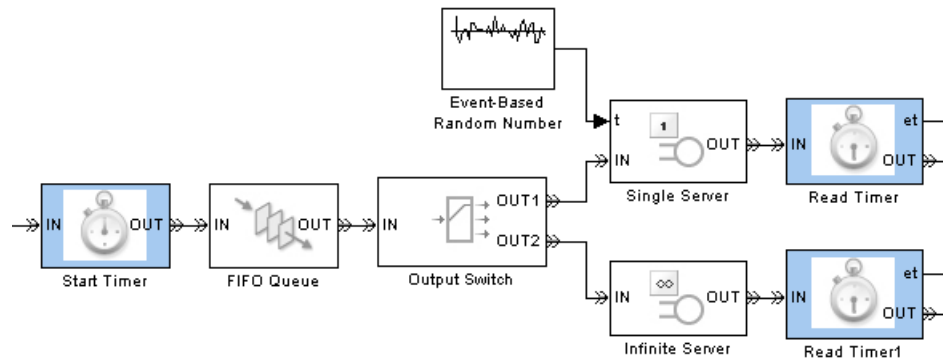
If you need multiple independent timers per entity (for example, to time an entity's progress through two possibly overlapping regions of the model), then follow the procedure above for each of the independent timers. For more information, see "Timing Multiple Processes Independently" on page 12-26.

Timing Multiple Entity Paths with One Timer

If your model includes routing blocks, then different entities might use different entity paths. To have a timer cover multiple entity paths, you can include multiple Start Timer or multiple Read Timer blocks in a model, using the same **Timer tag** parameter value in all timer blocks.

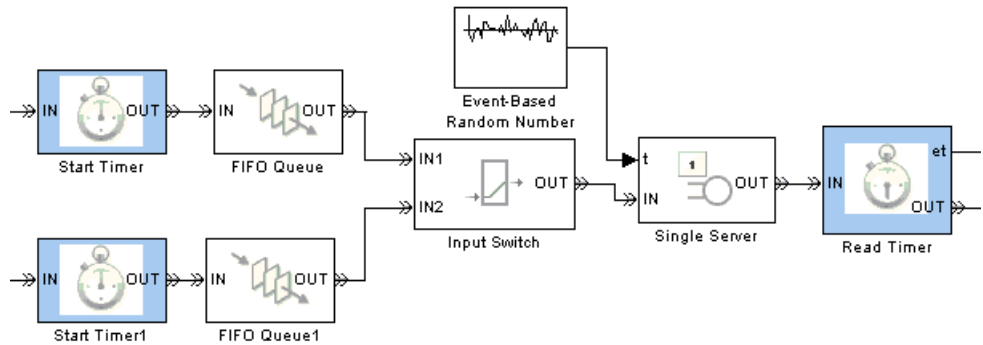
Output Switch Example

In the figure below, each entity advances along one of two different entity paths via the Output Switch block. The timer continues timing, regardless of the selected path. Finally, each entity advances to one of the two Read Timer blocks, which reads the value of the timer.



Input Switch Example

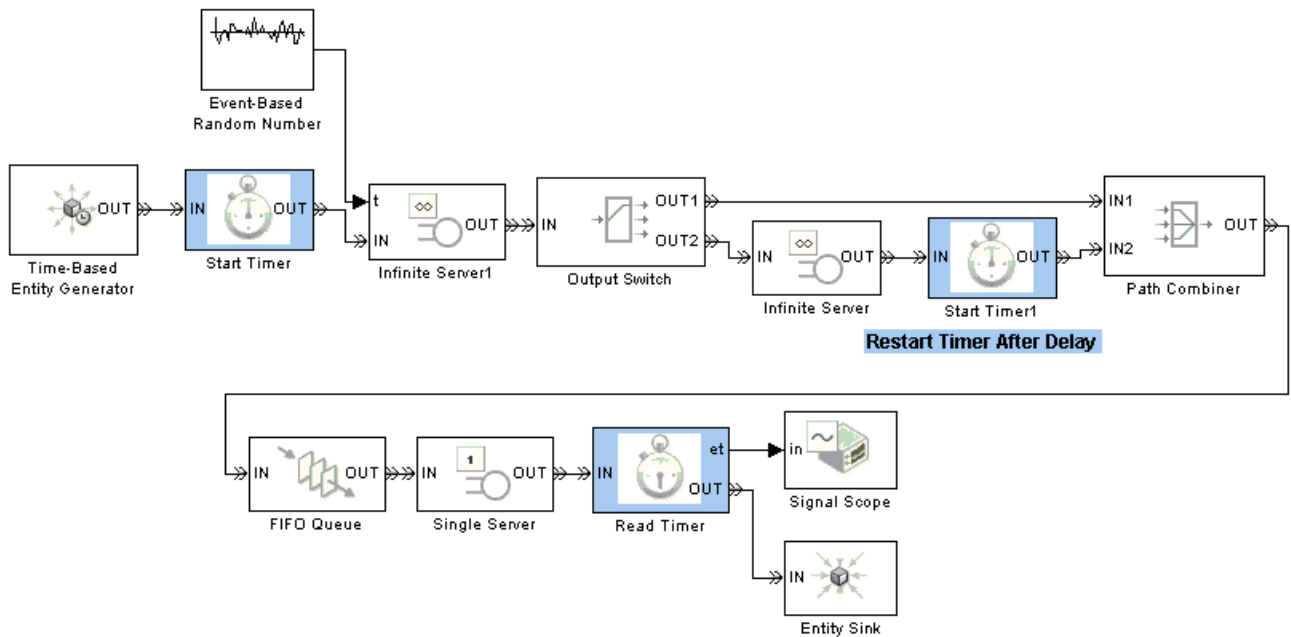
In the figure below, entities wait in two different queues before advancing to a single server. The timer blocks measure the time each entity spends in its respective queue-server pair. Two Start Timer blocks, configured with the same **Timer tag** parameter value, ensure that all entities possess a timer regardless of the path they take before reaching the server.



Restarting a Timer from Zero

You can restart an entity's timer, that is, reset its value to zero, whenever the entity reaches a spot in the model that you designate. To do this, insert a Start Timer block in the model where you want to restart the timer. Then set the block's **If timer has already started** parameter to Restart.

The figure below shows an example of restarting a timer.



All timer blocks share the same **Timer tag** parameter value. All entities that arrive at the first Start Timer block acquire a timer, which starts timing immediately. All entities incur an initial delay, modeled by an Infinite Server block. When entities reach the Output Switch block, they depart via one of the two entity output ports and receive different treatment:

- Entities that depart via the **OUT1** port advance to the queue with no further delay, and the timer continues timing.
- Entities that depart via the **OUT2** port incur an additional delay, modeled by another Infinite Server block. After the delay, the timer restarts from zero and the entity advances to the queue.

When entities finally advance from the server to the Read Timer block, the elapsed time is one of these quantities:

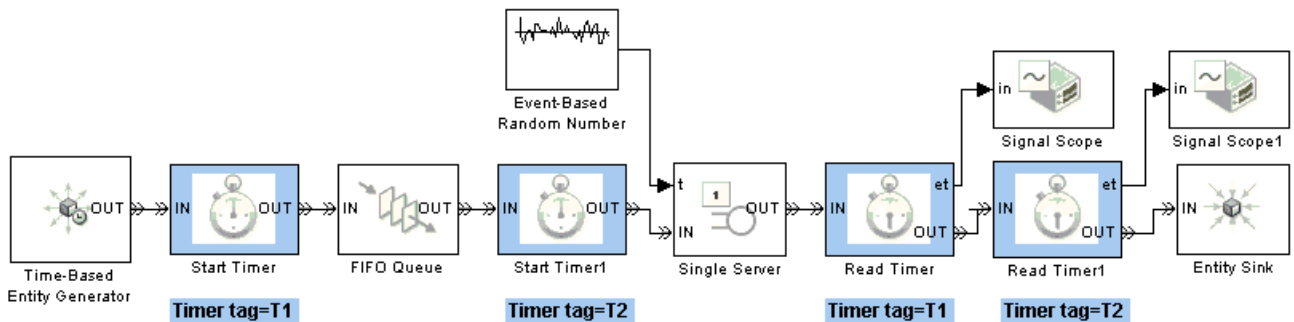
- The initial delay plus the time in the queue plus the time in the server, for entities that departed from the Output Switch block's **OUT1** port
- The time in the queue plus the time in the server, for entities that departed from the Output Switch block's **OUT2** port

Timing Multiple Processes Independently

You can measure multiple independent durations using the Start Timer and Read Timer blocks. To do this, create a unique **Timer tag** parameter for each independent timer. For clarity in your model, consider adding an annotation or changing the block names to reflect the **Timer tag** parameter in each timer block.

The figure below shows how to measure these quantities independently:

- The time each entity spends in the queue-server pair, using a timer with tag T1
- The time each entity spends in the server, using a timer with tag T2



The annotations beneath the blocks in the figure indicate the values of the **Timer tag** parameters. Notice that the T1 timer starts at the time when entities arrive at the queue, while the T2 timer starts at the time when entities depart from the queue (equivalently, at the time when entities arrive

at the server). The two Read Timer blocks read both timers when entities depart from the server. The sequence of the Read Timer blocks relative to each other is not relevant in this example because no time elapses while an entity is in a Read Timer block.

Varying Simulation Results by Managing Seeds

In this section...

- “Connection Between Random Numbers and Seeds” on page 12-28
- “Making Results Repeatable by Storing Sets of Seeds” on page 12-29
- “Setting Seed Values Programmatically” on page 12-30
- “Sharing Seeds Among Models” on page 12-30
- “Working with Seeds Not in SimEvents Blocks” on page 12-31
- “Choosing Seed Values” on page 12-34

See also “Detecting Nonunique Seeds and Making Them Unique” on page 14-88.

Connection Between Random Numbers and Seeds

When a simulation uses random numbers and you compute statistical results from it, you typically want to use different sequences of random numbers in these situations:

- In the random processes of a single simulation run
- Across multiple simulation runs

To vary a sequence of random numbers, vary the *initial seed* on which the sequence of random numbers is based. SimEvents blocks that have a parameter called **Initial seed** include:

- Time-Based Entity Generator
- Event-Based Random Number
- Entity Splitter
- Blocks in the Routing library

Some blocks in other library sets have parameters that represent initial seeds. For example, the Random Number and Uniform Random Number blocks in the Simulink Sources library have parameters called **Initial seed**.

Also, if your simulation is configured to randomize the sequence of certain simultaneous events, the Configuration Parameters dialog box has a parameter called **Seed for event randomization**. This parameter indicates the initial seed for the sequence of random numbers that affect processing of simultaneous events.

Making Results Repeatable by Storing Sets of Seeds

If you need to repeat the results of a simulation run and expect to change random number sequences, then you should store the seeds before changing them. You can later repeat the simulation results by resetting the stored seeds; see “Setting Seed Values Programmatically” on page 12-30 to learn more.

When all seeds are parameters of SimEvents blocks, use this procedure to store the seeds:

- 1 Decide whether you want to store seeds from SimEvents blocks in a system (including subsystems at any depth) or from a single block.
- 2 Create a string variable (called `sysid`, for example) that represents the system name, subsystem path name, or block path name.

Tip To avoid typing names, use `gcb` or `gcs`:

- Select a subsystem or block and assign `sysid = gcb`.
 - Click in a system or subsystem and assign `sysid = gcs`.
-

- 3 Use the `se_getseeds` function with `sysid` as the input argument. The output is a structure having these fields:
 - `system` — Value of the `sysid` input to `se_getseeds`
 - `seeds` — Structure array, of which each element has these fields:
 - `block` — Path name of a block that uses a random number generator, relative to `system`
 - `value` — Numeric seed value of the block

- 4 Store the output in an array, cell array, or MAT-file. Use a MAT-file if you might need to recover the values in a different session.

For an example that uses `se_getseeds`, see the Seed Management Workflow for Random Number Generators demo.

If your model uses random numbers in contexts other than SimEvents blocks, see “Working with Seeds Not in SimEvents Blocks” on page 12-31.

Setting Seed Values Programmatically

To set seed values programmatically in blocks that use random numbers, use one or more of these approaches:

- If you have a seed structure in the same format as the output of the `se_getseeds` function, use the `se_setseeds` function to set seed values in the corresponding blocks.

For an example, see the Seed Management Workflow for Random Number Generators demo.

- If you want the application to choose seed values for you and then set the values in some or all SimEvents blocks, use the `se_randomizeseeds` function.

For examples, see the Avoiding Identical Seeds for Random Number Generators demo and “Example: Computing an Ensemble Average Using MATLAB Code” on page 15-26. To learn about specific options for using `se_randomizeseeds`, see its reference page.

- If your model uses random numbers in contexts other than SimEvents blocks, use the `set_param` function to set seed values.

For examples, see “Working with Seeds Not in SimEvents Blocks” on page 12-31.

Sharing Seeds Among Models

Suppose you want to share seeds among multiple variants of a model or among models that have a common subsystem. The `se_getseeds` and `se_setseeds` functions provide a convenient way to apply seed values of the SimEvents blocks in one model to the corresponding blocks in a second model. Use this procedure:

- 1 Create string variables (for example, `sys1` and `sys2`) that represent the system names of the two models.
- 2 Open both models, if you have not already done so.
- 3 Use the `se_getseeds` function with `sys1` as the input argument. The result is a seed structure that represents the seeds in the SimEvents blocks in model `sys1`.
- 4 Use the `se_setseeds` function with the seed structure as the first input argument and `sys2` as the second input argument. The function uses information from the seed structure but overrides the system name stored in the seed structure. As a result, the function sets the seeds in model `sys2` to values from model `sys1`.

Working with Seeds Not in SimEvents Blocks

The seed management features in SimEvents software cover blocks in the SimEvents libraries. If your model uses random number sequences in other blocks or in the **Seed for event randomization** configuration parameter, you can use `get_param` and `set_param` commands to retrieve and set the seeds, respectively. These examples illustrate the techniques:

- “Example: Retrieving and Changing a Seed in a Custom Subsystem” on page 12-31
- “Example: Retrieving and Changing the Seed for Event Randomization” on page 12-33

Example: Retrieving and Changing a Seed in a Custom Subsystem

This example illustrates how to identify relevant variable names for seed parameters, query seed values, and set seed values. The specific block in this example is the Uniform Random Number block within a custom masked subsystem in a demo model.

- 1 Open the demo model.

```
sedemo_md1
```

- 2** Select the block labeled Exponential Generation and store its path name. Exponential Generation is a custom masked subsystem that has a seed parameter related to a Uniform Random Number block under the mask.

```
blk = gcb; % Pathname of current block
```

- 3** Query the dialog parameters of the block.

```
vars = get_param(blk, 'DialogParameters')  
  
vars =  
  
    seed: [1x1 struct]
```

The term `seed` in the output indicates a parameter's underlying variable name, which can differ from the text label you see in the block dialog box. You might guess that `seed` represents the seed of a random number generator. Optionally, you can confirm that this variable name corresponds to the **Initial seed** text label in the dialog box using this command:

```
textlabel = vars.seed.Prompt  
  
textlabel =  
  
Initial seed
```

- 4** Query the seed parameter for its value.

```
thisseed = get_param(blk, 'seed')  
  
thisseed =  
  
60790
```

- 5** Change the value of the seed parameter to a constant.

```
newseed = '60791'; % String whose value is a number  
set_param(blk, 'seed', newseed);
```

See “Choosing Seed Values” on page 12-34 for criteria related to the values you choose for seeds.

- 6** Change the value of the `seed` parameter to the name of a variable in the workspace. As a result, the dialog box shows the name of the variable instead of the value stored in the variable. This approach might be useful if you want to use `set_param` once and then change the workspace variable repeatedly (for example, within a loop) to vary the seed value.

```
seedvariable = 60792; % Numeric variable
set_param(blk,'seed',...
    'seedvariable'); % Parameter refers to variable
```

Example: Retrieving and Changing the Seed for Event Randomization

This example illustrates how to query and change the **Seed for event randomization** configuration parameter programmatically.

- 1** Open the demo model.

```
sys = 'sedemo_event_priorities';
open_system(sys);
```

- 2** Retrieve the value of the **Seed for event randomization** configuration parameter.

```
thiseventseed = get_param(sys,'propIdentEventSeed')

thiseventseed =

12345
```

- 3** Change the value to a constant.

```
neweventseed = '82937'; % String whose value is a number
set_param(sys,'propIdentEventSeed',neweventseed);
```

- 4** Change the value to the name of a variable in the workspace. As a result, the dialog box shows the name of the variable instead of the value stored in the variable. This approach might be useful if you want to use `set_param` once and then change the workspace variable repeatedly (for example, within a loop) to vary the seed value.

```
eventseedvariable = 82938; % Numeric variable
```

```
set_param(sys,'propIdentEventSeed,...  
    'seedvariable'); % Parameter refers to variable
```

Choosing Seed Values

Here are some recommendations for choosing appropriate values for seed parameters of blocks:

- If you choose a seed value yourself, choose an integer between 0 and $2^{32}-1$.
- To obtain the same sequence of random numbers the next time you run the same simulation, set the seed to a fixed value.
- To obtain a different sequence of random numbers the next time you run the same simulation, use one of these approaches:
 - Change the value of the seed, using the `se_randomize_seeds` function or any other means.
 - Set the value of the seed to a varying expression such as `mod(ceil(cputime*99999),2^32)`. See the `cputime` function for more details.
- If seed parameters appear in multiple places in your model, choose different values, or expressions that evaluate to different values, for all seed parameters. To have the application detect nonunique seeds in SimEvents blocks, use the “Identical seeds for random number generators” configuration parameter. To learn how to make seeds unique in SimEvents blocks across a model, see “Detecting Nonunique Seeds and Making Them Unique” on page 14-88.

Regulating the Simulation Length

In this section...

“Overview” on page 12-35

“Setting a Fixed Stop Time” on page 12-35

“Stopping Upon Processing a Fixed Number of Entities” on page 12-36

“Stopping Upon Reaching a Particular State” on page 12-37

Overview

When you gather statistics from a simulation, ending the simulation at the right time is more important than if you are only observing behavior qualitatively. Typical criteria for ending a discrete-event simulation include the following:

- A fixed amount of time passes
- The simulation processes a fixed number of packets, parts, customers, or other items that entities represent
- The simulation achieves a particular state, such as an overflow or a machine failure

Setting a Fixed Stop Time

To run a simulation interactively with a fixed stop time, do the following:

- 1** Open the **Configuration Parameters** dialog box by choosing **Simulation > Configuration Parameters** in the menu of the model window.
- 2** In the dialog box, set **Stop time** to the desired stop time.
- 3** Run the simulation by choosing **Simulation > Start**.

To fix the stop time when running a simulation programmatically, use syntax like

```
sim('model',timespan)
```

where `model` is the name of the model and `timespan` is the desired stop time.

Stopping Upon Processing a Fixed Number of Entities

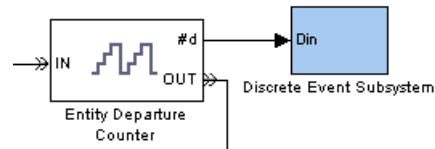
By counting entities, you can stop the simulation when the simulation processes a fixed number of entities. The basic procedure for stopping a simulation based on the total number of entity departures from a block is:

- 1 Find the parameter of the block that enables the departure counter as a signal output. Most blocks call the parameter **Number of entities departed**. Exceptions are in the following table.

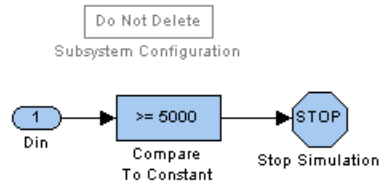
Block	Parameter
Entity Departure Counter	Write count to signal port #d
Entity Sink	Number of entities arrived

- 2 Set the parameter to On. This setting causes the block to have a signal output port corresponding to the entity count.
- 3 Connect the new signal output port to a Discrete Event Subsystem block, from the SimEvents Ports and Subsystems library.
- 4 Double-click the Discrete Event Subsystem block to open the subsystem it represents.
- 5 Delete the Discrete Event Output block labeled Dout.
- 6 Connect the Discrete Event Inport block labeled Din to a Compare To Constant block, from the Logic and Bit Operations library in the Simulink library set.
- 7 In the Compare To Constant block,
 - Set **Operator** to `>=`.
 - Set **Constant value** to the desired number of entity departures.
 - Set **Output data type mode** to `boolean`.
- 8 Connect the Compare To Constant block to a Stop Simulation block, from the Simulink Sinks library. The result should look like the following,

except that your SimEvents block might be a block other than Entity Departure Counter.



Top-Level Model



Subsystem Contents

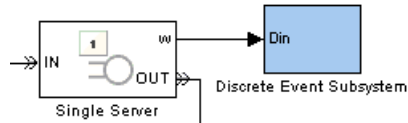
See the considerations discussed in “Tips for Using State-Based Stopping Conditions” on page 12-40 below. They are relevant if you are stopping the simulation based on an entity count, where “desired state” means the entity-count threshold.

Stopping Upon Reaching a Particular State

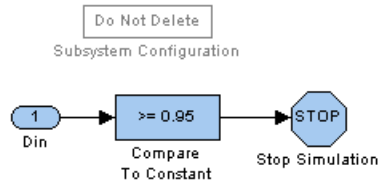
Suppose you want the simulation to end when it achieves a particular state, such as an overflow or a machine failure. The state might be the only criterion for ending the simulation, or the state might be one of multiple criteria, each of which is sufficient reason to end the simulation. An example that uses multiple criteria is a military simulation that ends when all identified targets are destroyed or all resources (ammunition, aircraft, etc.) are depleted, whichever occurs first.

Once you have identified a state that is relevant for ending the simulation, you typically create a Boolean signal that queries the state and connect the signal to a Stop Simulation block. Typical ways to create a Boolean signal that queries a state include the following:

- Connect a signal to a logic block to determine whether the signal satisfies some condition. See the blocks in the Simulink Logic and Bit Operations library. The following figure below illustrates one possibility.

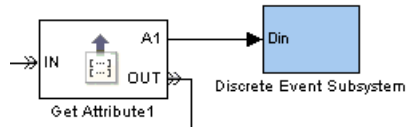


Top-Level Model

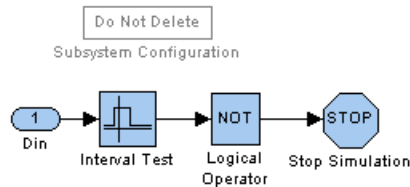


Subsystem Contents

- Use a Get Attribute block to query an attribute and a logic block to determine whether the attribute value satisfies some condition. The next figure illustrates one possibility.

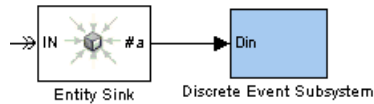


Top-Level Model

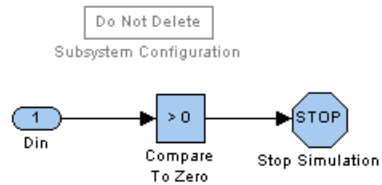


Subsystem Contents

- To end the simulation whenever an entity reaches a particular entity path, you can end that path with an Entity Sink block, enable that block's output signal to count entities, and check whether the output signal is greater than zero.

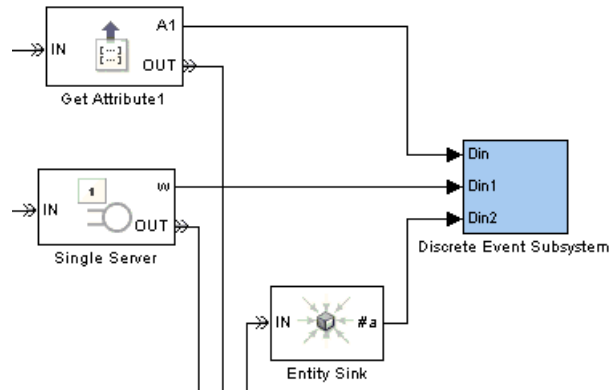


Top-Level Model

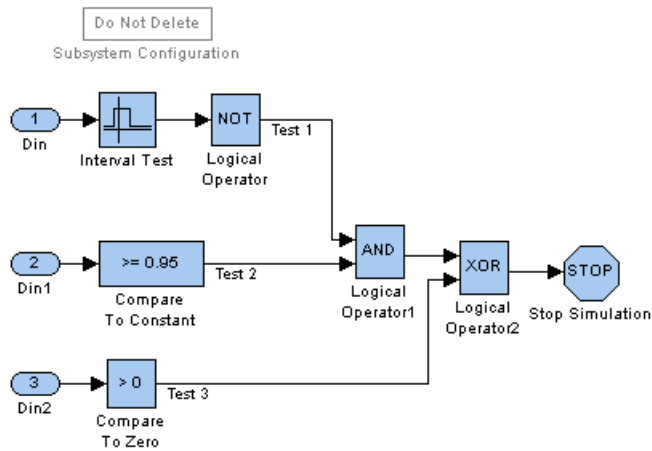


Subsystem Contents

- Logically combine multiple tests using logic blocks to build the final Boolean signal that connects to a Stop Simulation block. (A logical OR operation is implied if your model contains an independent Stop Simulation block for each of the multiple tests, meaning that the simulation ends when the first such block processes an input signal whose value is true.) The figure below illustrates one possibility using the exclusive-OR of two tests, one of which is in turn the logical AND of two tests.



Top-Level Model



Subsystem Contents

Tips for Using State-Based Stopping Conditions

When using a state rather than a time to determine when the simulation ends, keep in mind the following considerations:

- If the model has a finite stop time, then the simulation might end before reaching the desired state. Depending on your needs, this might be a desirable or undesirable outcome. If it is important that the simulation not

stop too early, then you can follow the instructions in “Setting a Fixed Stop Time” on page 12-35 and use `Inf` as the **Stop time** parameter.

- If you set the **Stop time** parameter to `Inf`, then you should ensure that the simulation actually stops. For example, if you want to stop based on an entity count but the simulation either reaches a deadlock or sends most entities on a path not involving the block whose departure count is the stopping criterion, then the simulation might not end.
- Checking for the desired state throughout the simulation might make the simulation run more slowly than if you used a fixed stop time.

Using Stateflow Charts in SimEvents Models

- “Role of Stateflow Charts in SimEvents Models” on page 13-2
- “Guidelines for Using Stateflow and SimEvents Blocks” on page 13-3
- “Examples Using Stateflow Charts and SimEvents Blocks” on page 13-4

Role of Stateflow Charts in SimEvents Models

SimEvents software works with Stateflow software to represent systems containing state-transition diagrams that can produce or be controlled by discrete events. Both software products are related to event-driven modeling, but they play different roles:

- SimEvents blocks can model the movement of entities through a system so you can learn how such movement relates to overall system activity. Entities can carry data with them. Also, SimEvents blocks can generate events at times that are truly independent of the time steps dictated by the ODE solver in Simulink software.
- Stateflow charts can model the state of a block or system. Charts enumerate the possible values of the state and describe the conditions that cause a state transition. Runtime animation in a Stateflow chart depicts transitions but does not indicate movement of data.

For scenarios that combine SimEvents blocks with Stateflow charts, see “Examples Using Stateflow Charts and SimEvents Blocks” on page 13-4.

You can interpret the Signal Latch block with the `st` output signal enabled as a two-state machine that changes state when read and write events occur. Similarly, you can interpret Input Switch and Output Switch blocks as finite-state machines whose state is the selected entity port. However, Stateflow software offers more flexibility in the kinds of state machines you can model and an intuitive development environment that includes animation of state transitions during the simulation.

Guidelines for Using Stateflow and SimEvents Blocks

When your model contains Stateflow charts in addition to SimEvents blocks, you must follow these rules:

- If the chart is capable of propagating its execution context, select this option as follows:
 - 1** Select the Stateflow block and choose **Edit > Subsystem Parameters** from the model window's menu bar.
 - 2** In the dialog box that opens, select **Propagate execution context across subsystem boundary** if it appears and click **OK**. If this parameter does not appear in the dialog box, just click **OK**.

Note If the chart does not offer this option, you might see a delay in the response of other blocks to the chart's output signals. The duration of the delay is the time between successive calls to the chart.

- If an output of the chart connects to a SimEvents block, do not configure the chart to be entered at initialization. To ensure that this configuration is correct,
 - 1** Select the **File > Chart Properties** from the chart window's menu bar.
 - 2** In the dialog box that opens, clear **Execute (enter) Chart At Initialization** and click **OK**. This check box is cleared by default.

When you design default transitions in your chart, keep in mind that the chart will not be entered at initialization. For example, notice that the default transition in the example in “Example: Failure and Repair of a Server” on page 5-22 indicates the state corresponding to the first actual event during the simulation, not an initial state.

- If the chart has an output signal, you can provide a nonzero initial condition using the Initial Value block as in “Specifying Initial Values of Event-Based Signals” on page 4-12. Because the chart is not entered at initialization, you cannot use the chart itself to provide a nonzero initial condition for the output signal.

Examples Using Stateflow Charts and SimEvents Blocks

In this section...
“Failure State of Server” on page 13-4
“Go-Back-N ARQ Model” on page 13-4

Failure State of Server

The examples in “Using Stateflow Charts to Implement a Failure State” on page 5-21 use Stateflow charts to implement the logic that determines whether a server is down, under repair, or operational. SimEvents blocks model the asynchronous arrival of customers, advancement of customers through a queue and server, and asynchronous failures of the server. While these examples could alternatively have represented the server’s states using signal values instead of states of a Stateflow chart, the chart approach is more intuitive and scales more easily to include additional complexity.

Go-Back-N ARQ Model

The Go-Back-N Automatic Repeat Request demo uses SimEvents and Stateflow blocks to model a communication system. SimEvents blocks implement the movement of data frames and acknowledgment messages from one part of the system to another. Stateflow blocks implement the logical transitions among finitely many state values of the transmitter and the receiver.

Receiver State

At the receiver, the chart decides whether to accept or discard an incoming frame of data, records the identifier of the last accepted frame, and regulates the creation of acknowledgment messages. Interactions between the Stateflow chart and SimEvents blocks include these:

- The arrival of an entity representing a data frame causes the generation of a function call that invokes the chart.
- The chart can produce a routing signal that determines which path entities take at an Output Switch block.

- The chart can produce a function call that causes the Event-Based Entity Generator block to generate an entity representing an acknowledgment message.

Transmitter State

At the transmitter, the chart controls the transmission and retransmission of frames. Interactions between the Stateflow chart and SimEvents blocks include these:

- The arrival of an entity representing a new data frame or an acknowledgment message causes the generation of a function call that invokes the chart.
- The completion of transmission of a frame (that is, the completion of service on an entity representing a frame) causes the generation of a function call that invokes the chart.
- The chart can produce a routing signal that determines which path entities take at an Output Switch block.
- The chart can produce a function call that causes the Release Gate block to permit the advancement of an entity representing a data frame to transmit (function call at Stateflow block's tx output port) or retransmit (function call at Stateflow block's retx output port).

Debugging Discrete-Event Simulations

- “Overview of Debugging Resources” on page 14-2
- “Overview of the SimEvents Debugger” on page 14-3
- “Starting the SimEvents Debugger” on page 14-5
- “The Debugger Environment” on page 14-7
- “Independent Operations and Consequences in the Debugger” on page 14-20
- “Stopping the Debugger” on page 14-24
- “Stepping Through the Simulation” on page 14-26
- “Inspecting the Current Point in the Debugger” on page 14-31
- “Inspecting Entities, Blocks, and Events” on page 14-34
- “Working with Debugging Information in Variables” on page 14-41
- “Viewing the Event Calendar” on page 14-46
- “Customizing the Debugger Simulation Log” on page 14-47
- “Debugger Efficiency Tips” on page 14-54
- “Defining a Breakpoint” on page 14-56
- “Using Breakpoints During Debugging” on page 14-62
- “Block Operations Relevant for Block Breakpoints” on page 14-65
- “Common Problems in SimEvents Models” on page 14-72
- “Recognizing Latency in Signal Updates” on page 14-90

Overview of Debugging Resources

To Read About...	Refer to...
Running the simulation using the SimEvents debugger	“Overview of the SimEvents Debugger” on page 14-3
Some modeling errors and ways to avoid them	“Common Problems in SimEvents Models” on page 14-72
Plotting signals, attribute values, event information, or entity information during the simulation	“Using Plots for Troubleshooting” on page 11-10
Gathering data during the simulation that reflects block behavior	“Accessing Statistics from SimEvents Blocks” on page 12-5 and “Sending Data to the MATLAB Workspace” on page 4-15
Examining the set or processing sequence of simultaneous events	“Exploring Simultaneous Events” on page 3-4

Overview of the SimEvents Debugger

SimEvents software includes a debugger that lets you use MATLAB functions to suspend a simulation at each step or breakpoint, and query simulation behavior. The debugger also creates a simulation log with detailed information about what happens during the simulation.

This table indicates sources of relevant information about the debugger. For information about debugging resources other than the debugger, see “Overview of Debugging Resources” on page 14-2.

To Read About...	Refer to...	Description
Functions	Debugger Function Reference	List of functions related to debugging
Examples	“Confirming Event-Based Behavior Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation	Stepping through a simulation
	“Exploring the D/D/1 System Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation	Querying the final state of a simulation
	A video tutorial on the Web, in two parts: <ul style="list-style-type: none"> • Basic Single Stepping and Querying • Breakpoints and Advanced Querying 	Stepping, querying, and using breakpoints
	“Example: Choices of Values for Event Priorities” on page 3-11	Interpreting the simulation log
	“Example: Preemption by High-Priority Entities” on page 5-12	Stepping forward from a breakpoint and interpreting the simulation log
	“Example: Deadlock Resulting from Loop in Entity Path” on page 14-83	Inferring the reasons for a simulation deadlock

To Read About...	Refer to...	Description
Procedures	“Starting the SimEvents Debugger” on page 14-5	How to start the debugger
	“Stepping Through the Simulation” on page 14-26 and “Using Breakpoints During Debugging” on page 14-62	How to control the simulation
	“Inspecting Entities, Blocks, and Events” on page 14-34 and “Customizing the Debugger Simulation Log” on page 14-47	How to get information
Background	“The Debugger Environment” on page 14-7	Working in debug mode and interpreting debugger displays

Starting the SimEvents Debugger

Before you simulate a system using the SimEvents debugger, make sure that the system is not currently simulating and that you are not running the Simulink debugger on any system.

To simulate a system using the SimEvents debugger, enter

```
sedebug(sys)
```

at the MATLAB command prompt. `sys` is a string representing the system name. An example is `sedebug('sedemo_outputswitch')`.

The results of starting the debugger are:

- The output in the MATLAB Command Window indicates that the debugger is active and includes hyperlinks to sources of information.

```
*** SimEvents Debugger ***
```

```
Functions | Help | Watch Video Tutorial
```

```
%=====
```

```
Initializing Model sedemo_outputswitch
```

```
sedebug>>
```

- The command prompt changes to `sedebug>>`. This notation is the debugger prompt where you enter commands.
- The system starts initializing. At this point, you can inspect initial states or configure the debugger before blocks have started performing operations.
- You cannot modify the system, modify parameters in the system or in its blocks, close the system, or end the MATLAB session until the debugger session ends. To end the debugger session, see “Stopping the Debugger” on page 14-24.

For more information on how to proceed, see one of these sections:

- “The Debugger Environment” on page 14-7
- “Stepping Through the Simulation” on page 14-26

- “Confirming Event-Based Behavior Using the SimEvents Debugger”, part of an example in the SimEvents getting started documentation
- A video tutorial on the Web, in two parts:
 - Basic Single Stepping and Querying
 - Breakpoints and Advanced Querying

The Debugger Environment

In this section...

“Debugger Command Prompt” on page 14-7

“Simulation Log in the Debugger” on page 14-8

“Identifiers in the Debugger” on page 14-18

Debugger Command Prompt

When the SimEvents debugger is active, the command prompt is `sedebug>>` instead of `>>`. The different prompt reminds you that the simulation is suspended in debugging mode.

When you enter commands at the `sedebug>>` prompt, you can:

- Invoke debugger functions using only the function names, without the package qualifier, `sedb..` For example, you can enter `step` at the `sedebug>>` prompt even though the fully qualified name of the function is `sedb.step`.

Debugger functions that you invoke at the `sedebug>>` prompt are in the `sedb` package, so their fully qualified names start with “`sedb.`”. You can either include or omit the `sedb.` prefix when entering commands at the `sedebug>>` prompt because the `sedebug` function imports the `sedb` package for convenience.

- See a list of debugger functions by entering `help` with no input arguments.

At the `>>` prompt, the same list is available via the syntax `help sedb`.

When you enter commands at the `sedebug>>` prompt, follow these rules:

- Do not append additional debugger commands on the same line as a command that causes the simulation to proceed. For example, to step twice, you must enter `step` on two lines instead of entering the single command `step; step`.
- Do not invoke `sedebug` or `sldebug`.

Functions in the `sedb` package are valid only when the SimEvents debugger is active.

Simulation Log in the Debugger

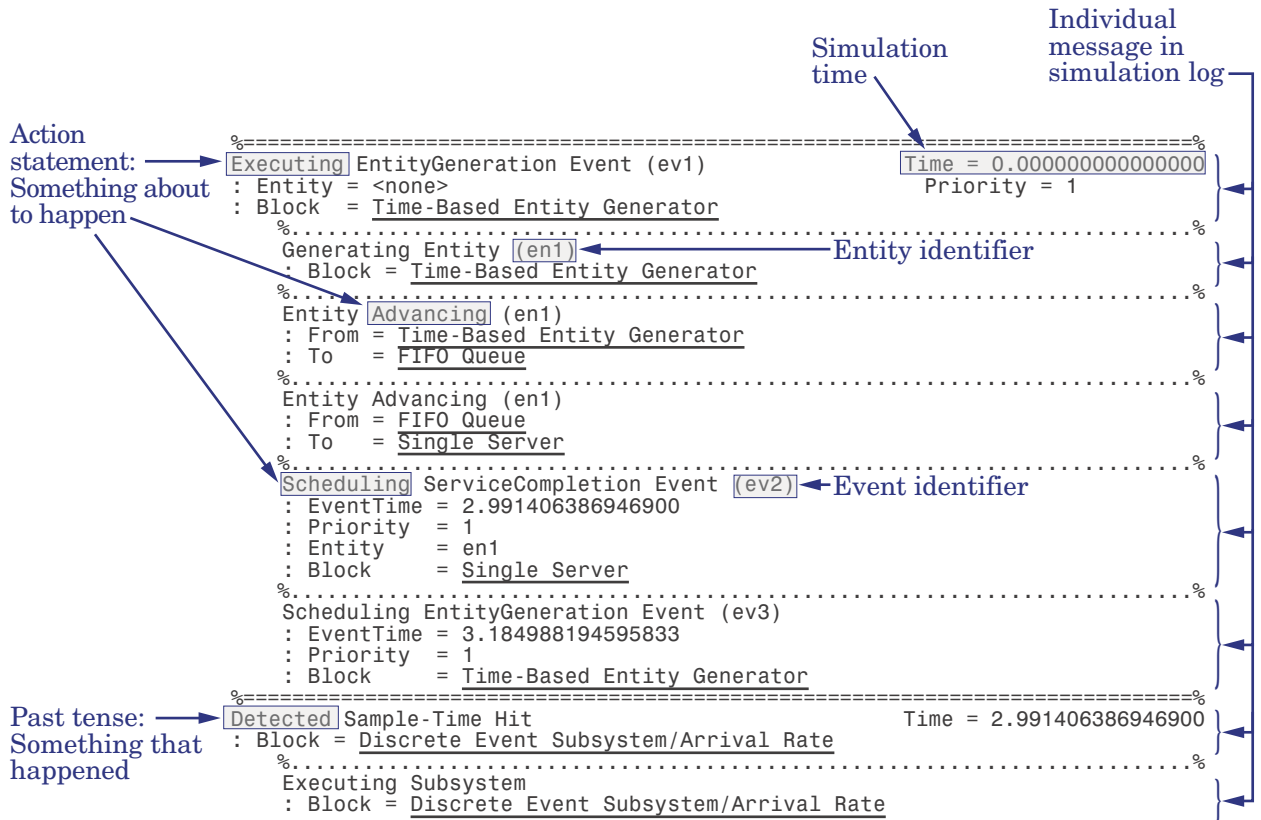
When the simulation proceeds during debugging mode, the debugger displays messages in the Command Window to indicate what is about to happen or what has just happened in the simulation. Collectively, these messages make up the simulation log. These topics describe what you see in the simulation log:

- “Key Parts of the Simulation Log” on page 14-8
- “Optional Displays of Event Calendar in the Simulation Log” on page 14-10
- “Interpreting the Simulation Log” on page 14-11
- “Event Scheduling, Execution, and Cancellation Messages” on page 14-13
- “Detection Messages” on page 14-14
- “Entity Operation Messages” on page 14-16
- “Monitoring Block Messages” on page 14-17
- “Initialization Messages” on page 14-18

To learn how to obtain additional information via query commands, see “Inspecting Entities, Blocks, and Events” on page 14-34.

Key Parts of the Simulation Log

Here is a sample excerpt of the simulation log in its default configuration.



The sample highlights these typical features of messages in the simulation log:

- Individual messages typically span multiple lines in the Command Window. Lines that look like `%====%` or `%...%` precede each individual message.
- The first line of each message provides a summary. The phrasing of the summary indicates whether the message describes something that is about to happen or that has just happened. Knowing the difference is important when you inspect states.

Phrasing of Summary	Examples	Interpretation
Action statement	Executing, scheduling, advancing, generating, and others	The action is about to happen and has not happened yet.
Verb in the past tense	Detected	The action has just happened.

- Some messages include the simulation time. Messages that do not include the simulation time describe activities at the simulation time most recently included in the simulation log.
- Some messages use tokens in parentheses to identify events or entities uniquely. For details, see “Identifiers in the Debugger” on page 14-18.
- Some messages are indented to indicate operations that are consequences of other operations. For details, see “Independent Operations and Consequences in the Debugger” on page 14-20.
- Block names appear underlined and act as hyperlinks.

If you click the path name, the model window opens and highlights the block. To clear the highlighting, select **View > Remove Highlighting**.

Optional Displays of Event Calendar in the Simulation Log

You can configure the debugger to list the events on the event calendar, as part of the simulation log. Event calendar listings appear before execution messages. The following illustration is a sample event calendar listing.

```

%-----%
Events in the Event Calendar
  ID      EventTime      EventType      Priority  Entity      Block
  ev2     2.991406386946900   ServiceCompletion  1        en1         Single Server
  ev3     3.184988194595833   EntityGeneration  1        <none>      TB Entity Gen
%=====
Executing ServiceCompletion Event (ev2)      Time = 2.991406386946900
: Entity = en1                               Priority = 1
: Block = Single Server

```

The event calendar display spans multiple lines in the Command Window. A line that looks like %- - - % precedes the event calendar display.

Each listing in the event calendar display includes these characteristics of the event:

- Event identifier, which is a token that identifies the event uniquely during the debugger session. For details, see “Identifiers in the Debugger” on page 14-18.
- Scheduled time of the event.
- Event type. For details, see “Supported Events in SimEvents Models” on page 2-2.
- Event priority. The value can be a number, SYS1, or SYS2. For details, see “Event Sequencing” on page 16-2 or “Overview of Simultaneous Events” on page 3-2.
- Entity identifier of an entity associated with the event, if applicable. If no entity is associated with the event, <none> appears.
- Partial path of the block that processes the event when it occurs. The partial path omits the name of the system, but is otherwise the same as the full path name of the block.

Block paths appear underlined and act as hyperlinks.

If you click the path, the model window opens and highlights the block. To clear the highlighting, select **View > Remove Highlighting**.

When comparing the simulation log with the event calendar display, note that the sequence on the calendar and the sequence in which the events were scheduled might differ, even for simultaneous events that have equal priority. For details, see “Procedure for Specifying Equal-Priority Behavior” on page 3-9.

To learn how to configure the debugger to show event calendar displays, see “Customizing the Debugger Simulation Log” on page 14-47.

Interpreting the Simulation Log

The most recent message in the simulation log shows you the simulation status while the debugger is waiting for you to enter the next command.

When you inspect states, if the phrasing of the summary uses an action statement to indicate that the action has not happened yet, the state does not reflect the completion of the action. For example, if the most recent message in the simulation log indicates that a server block is scheduling a service completion event, the scheduling has not occurred yet and the `evcal` function does not list the event on the event calendar. If you want to see where the event is on the event calendar relative to other scheduled events, you must proceed in the simulation by one step and then call `evcal`.

As a whole, the simulation log shows you what is happening during the simulation. When you simulate in debugging mode, the actions in the simulation log are the units of debugging behavior. By using the debugger on those units of debugging behavior, you can control the simulation process and query simulation behavior. When you proceed in the simulation step by step, this means suspending the simulation before or after the actions that the simulation log reports.

By contrast, some approaches that the simulation log does *not* intend to reflect are:

- A block-by-block simulation process. In a discrete-event simulation, a block can take multiple actions at a given time, potentially interspersed with actions of other blocks. Alternatively, a block can take no action for long periods of time, even while other blocks in the system are active. Messages in the simulation log might not be in the same sequence that you see in the topology of the block diagram, even if the topology of the block diagram is linear.
- An entity-by-entity simulation process. An entity can advance through multiple blocks or can be the object of multiple actions at a given time, potentially interspersed with actions on other entities. Alternatively, an entity can go for long periods of time without moving, changing, or affecting anything else in the simulation, even while other entities in the system are active.
- One message per time value. In a discrete-event simulation, many actions can occur at the same time. Therefore the debugger lets you query simulation behavior at intermediate points.

The view into simulation behavior that the simulation log reveals is useful during debugging because the simulation log reflects what is happening

during the simulation. When you see what is happening, you can diagnose problems, explore solutions, improve your modeling proficiency, and learn about the underlying system you are modeling.

Event Scheduling, Execution, and Cancellation Messages

When the event calendar is about to change during a debugging session, the simulation log displays a message that describes the event that the block is about to schedule, execute, or cancel. The event calendar has not yet changed.

When a block is about to execute a subsystem, function call, memory read, or memory write event that is not on the event calendar, the simulation log displays a message that describes the event. The execution, which is a dependent operation, has not yet occurred.

Sample Scheduling Message

The following sample shows that the Single Server block is about to add a service completion event to the event calendar. The event has priority 500, a scheduled time of $T=5$, and identifier `ev3`. The event represents the completion of service on the entity whose identifier is `en2`.

```
%.....%
Scheduling ServiceCompletion Event (ev3)
: EventTime = 5.0000000000000000
: Priority   = 500
: Entity    = en2
: Block     = Single Server
```

Sample Independent Execution Message

The following sample shows that the Single Server block is about to execute the service completion event for the entity whose identifier is `en2`. The event time of $T=5$ equals the current simulation time. The event has priority 500, which is relevant if multiple events on the event calendar share the same event time. The event identifier is `ev3`.

```
%=====
Executing ServiceCompletion Event (ev3)           Time = 5.0000000000000000
: Entity = en2                                   Priority = 500
: Block  = Single Server
```

Sample Dependent Execution Message

The following sample shows that the Discrete Event Subsystem is about to execute based on the behavior of its Discrete Event Inport block labeled Din1. A parameter in the Din1 block causes the subsystem execution not to be on the event calendar, which is why the execution is not an independent operation.

```

%.....%
Executing Subsystem
: Block = Discrete Event Subsystem/Din1
    
```

Sample Cancelation Message

The following sample shows that the Single Server block is about to cancel the service completion event for the entity whose identifier is en2, because the entity has timed out. The service completion event has priority 500 and identifier ev4.

```

%=====
Executing Timeout Event (ev3)                                Time = 1.0000000000000000
: Entity = en2                                              Priority = 1700
: Block = Single Server
%.....%
Canceling ServiceCompletion Event (ev4)
: EventTime = 5.0000000000000000
: Priority = 500
: Entity = en2
: Block = Single Server
    
```

To learn more about how the event calendar works, see “Event Sequencing” on page 16-2 or “Example: Event Calendar Usage for a Queue-Server Model” on page 2-10.

Detection Messages

Reactive ports, listed in “Notifying, Monitoring, and Reactive Ports” on page 16-21, listen for relevant updates in the input signal and cause an appropriate reaction in the block possessing the port. When labeled reactive ports detect relevant updates during a debugging session, the simulation log displays a message whose summary starts with **Detected**. These messages indicate that the update in the signal has happened and that the block has detected it.

The block has not yet responded to the update. The response depends on the particular block; for details, see individual block reference pages.

The appearance of detection messages depends on their source:

- If the input signal is time-based, the update is an independent operation and the detection message is not indented. This kind of detection message is one of the few ways in which time-based blocks affect the simulation log.
- If the input signal is event-based, the update is a dependent operation and the detection message is indented.

Note If a port not in the list of reactive ports, such as the **function()** input port of a Function-Call Subsystem block, detects a trigger or function call, the simulation log does not display a detection message.

Sample Detection Message

The following sample shows a detection message that indicates that the Event-Based Entity Generator block has detected that its input signal changed to a new value of 1 from a previous value of 0. Because the block is configured to respond to rising value changes, the signal update is relevant. The block responds by scheduling an entity generation event for the current simulation time (denoted by Now), as the subsequent message indicates.

```
%.....%
Detected Rising Value Change
: NewValue = 1
: PrevValue = 0
: Block    = Event-Based Entity Generator
%.....%
Scheduling EntityGeneration Event (ev2)
: EventTime = 0.000000000000000 (Now)
: Priority   = SYS1
: Block     = Event-Based Entity Generator
```

Entity Operation Messages

Various blocks produce messages in the simulation log that describe what they are about to do to entities. The details of entity operation messages depend on what is relevant for the particular block and operation.

Sample Entity Advancement Message

An entity whose identifier is en2 is about to depart from the Schedule Timeout block and arrive at the Single Server block. The entity has not yet advanced.

```
%.....%  
Entity Advancing (en2)  
: From = Schedule Timeout  
: To   = Single Server
```

Sample Queuing Message

A FIFO Queue block places entity en4 in the third position. Two other entities are ahead in the queue, which has a total capacity of 25.

```
%.....%  
Queuing Entity (en4)  
: FIFO Pos = 3 of 3  
: Capacity = 25  
: Block = FIFO Queue
```

Sample Attribute Assignment Message

A Set Attribute block assigns the value 3 to the attribute named RepCount of entity en1.

```
%.....%  
Setting Attribute on Entity (en1)  
: RepCount = 3  
: Block = Set Attribute
```

Sample Preemption Message

A Single Server block replaces entity en1 with entity en4 when preemption occurs, based on values of the entities' PriorityAttributeName attribute.

```

%.....%
Preempting Entity (en1)
: NewEntity = en4 (PriorityAttributeName = 1)
: OldEntity = en1 (PriorityAttributeName = 2)
: Block = Single Server1

```

Sample Entity Replication Message

A Replicate block replicates entity en1 to produce entity en2.

```

%.....%
Replicating Entity (en1)
: Replica 1 of 2 = en2
: Block = Replicate

```

Sample Entity Destruction Message

An Entity Sink block destroys entity en1.

```

%.....%
Destroying Entity (en1)
: Block = Entity Sink

```

Monitoring Block Messages

When a block is about to react to an update in a signal at a monitoring port, the block produces a message in the simulation log.

Sample Scope Message

A Signal Scope block updates its plot.

```

%.....%
Executing Scope
: Block = Signal Scope

```

Sample Workspace Message

A Discrete-Event Signal to Workspace block receives a new data value. The workspace variable is available only after the debugger session ends.

```
%.....%
Executing Discrete-Event Signal To Workspace
: Block = Discrete Event Signal to Workspace
```

Initialization Messages

Early in the debugging session, the debugger indicates initialization activities.

Message	Description
Initializing Model sedemo_timeout	Appears when the debugging session starts and the model is in the initialization stage.
Initializing Time-Based Entity Generators	Appears after the model initialization, if the model contains at least one Time-Based Entity Generator block. Initializing a Time-Based Entity Generator block means scheduling its first entity generation event.

Identifiers in the Debugger

The simulation log uses tokens to identify blocks, entities, events on the event calendar, and breakpoints uniquely during the debugger session. You use these identifiers as input arguments when requesting information about blocks, entities, or events, or when manipulating breakpoints.

If you repeat a debugging session in the same version of MATLAB without changing the model structure or parameters, all identifiers of blocks, entities, and events are the same from one session to the next.

The next table summarizes the notation for identifiers.

Type of Identifier	Prefix of Identifier	Example
Block	blk	blk1
Entity	en	en2
Event	ev	ev3
Breakpoint	b	b4

In displays of state information in the debugger, the abbreviation ID refers to an identifier.

To learn more about uses of identifiers, see “Inspecting Entities, Blocks, and Events” on page 14-34 or “Using Breakpoints During Debugging” on page 14-62.

Independent Operations and Consequences in the Debugger

In this section...
“Significance of Independent Operations” on page 14-20
“Independent Operations” on page 14-20
“Consequences of Independent Operations” on page 14-21

Significance of Independent Operations

This section describes a hierarchy of operations that the SimEvents debugger uses when interpreting `step out` or `step over` commands, and that you see in the indentation of the simulation log. Learning these definitions can help you use the `step` function more effectively and gain more insight from the simulation log.

Independent Operations

The simulation clock and the event calendar processor jointly drive the simulation of a SimEvents model, and act as sources of these *independent operations* in the debugger simulation log:

- Initialization of the model or any Time-Based Entity Generator blocks in the model. For more information, see “Initialization Messages” on page 14-18.
- Execution of an event on the event calendar. However, if the application executes an event without scheduling it on the event calendar, the event cannot be the basis of an independent operation. To learn which events are scheduled on the event calendar, see “Role of the Event Calendar” on page 16-3.
- Detection by a reactive port of a relevant update in a time-based input signal. You can think of these relevant updates as zero crossings or level crossings. However, if the input signal is an event-based signal or if the input port is not a reactive port, the update is not an independent operation.
- Execution of a block whose monitoring port connects to a time-based input signal.

Other operations that appear in the simulation log are consequences of an independent operation.

In the simulation log, an independent operation is not indented after a line that looks like %====%.

Consequences of Independent Operations

Consequences of independent operations that appear in the simulation log include, but are not limited to, the following:

- Scheduling of an event on the event calendar
- Cancellation of an event on the event calendar
- Detection by a reactive port of a relevant update in an event-based input signal
- Execution of a block whose monitoring port connects to an event-based input signal
- Entity operations, such as:
 - Advancement of an entity from one block to another
 - Queuing of an entity in a queue block
 - Assignment of an attribute to an entity
 - Preemption of an entity in a queue by an arriving entity
 - Replication of an entity
 - Destruction of an entity
- Execution of these events when they are not on the event calendar:
 - Subsystem execution
 - Function call creation
 - Memory read event
 - Memory write event

Consequences are also called dependent operations. In the simulation log, a dependent operation is indented, underneath the independent operation that causes it and after a line that looks like `%. . . %`.

Relationships Among Multiple Consequences

If an independent operation has multiple consequences, they appear in a sequence that reflects the simulation behavior.

Multiple consequences of an independent operation might or might not be causally related to each other. For example, in the following simulation log excerpt, each indented message represents a consequence of the execution of the `ev1` event. Among the indented messages, the `Scheduling ServiceCompletion` Event message is a direct consequence of the preceding message but is not directly related to the message that follows.

```
%=====
Executing EntityGeneration Event (ev1)                Time = 0.1000000000000000
: Entity = <none>                                       Priority = 300
: Block = Time-Based Entity Generator
sedebug>>step over
%. . . . .%
  Generating Entity (en1)
  : Block = Time-Based Entity Generator
%. . . . .%
  Entity Advancing (en1)
  : From = Time-Based Entity Generator
  : To   = Replicate
%. . . . .%
  Replicating Entity (en1)
  : Replica 1 of 2 = en2
  : Block = Replicate
%. . . . .%
  Entity Advancing (en2)
  : From = Replicate
  : To   = Set Attribute
%. . . . .%
  Setting Attribute on Entity (en2)
  : RepIndex = 1
  : Block = Set Attribute
```

```

%.....%
Entity Advancing (en2)
: From = Set Attribute
: To   = Infinite Server
%.....%
Scheduling ServiceCompletion Event (ev2)
: EventTime = 1.1000000000000000
: Priority   = 500
: Entity    = en2
: Block     = Infinite Server
%.....%
Replicating Entity (en1)
: Replica 2 of 2 = en3
: Block = Replicate
%.....%
Entity Advancing (en3)
: From = Replicate
: To   = Infinite Server1
%.....%
Scheduling ServiceCompletion Event (ev3)
: EventTime = 1.1000000000000000
: Priority   = 500
: Entity    = en3
: Block     = Infinite Server1
%.....%
Destroying Entity (en1)
: Block = Replicate
%.....%
Scheduling EntityGeneration Event (ev4)
: EventTime = 0.2000000000000000
: Priority   = 300
: Block     = Time-Based Entity Generator
%=====
Executing EntityGeneration Event (ev4)           Time = 0.2000000000000000
: Entity = <none>                               Priority = 300
: Block  = Time-Based Entity Generator

```

Stopping the Debugger

In this section...

“How to End the Debugger Session” on page 14-24

“Comparison of Simulation Control Functions” on page 14-24

How to End the Debugger Session

To end the debugger session without completing the simulation, enter one of these commands at the `sedebug>>` prompt:

```
sedb.quit
```

```
quit
```

The simulation ends, the debugging session ends, and the MATLAB command prompt returns.

At the `sedebug>>` prompt, `quit` is equivalent to `sedb.quit`. However, specifying the `sedb` package prevents you from inadvertently ending the MATLAB session if you enter the command at the incorrect command prompt.

Comparison of Simulation Control Functions

The functions in the next table have different behavior and purposes, but any of the functions can cause the debugger session to end.

Function	Behavior with Respect to Ending the Debugger Session	Primary Usage
<code>sedb.quit</code> or <code>quit</code>	Ends the debugger session without completing the simulation.	To end the debugger session immediately without spending time on further simulation log entries, plots, or other simulation behavior.
<code>runtoend</code>	Completes the simulation and then ends the debugger session.	To see the simulation log or plots but not enter commands.

Function	Behavior with Respect to Ending the Debugger Session	Primary Usage
cont	Ends the debugger session only if the simulation is suspended at the built-in breakpoint at the end of the simulation.	Primarily with breakpoints. For more information, see “Using Breakpoints During Debugging” on page 14-62 .
step	Ends the debugger session only if the simulation is suspended at the built-in breakpoint at the end of the simulation.	Primarily for proceeding step by step through the simulation. For more information, see “Stepping Through the Simulation” on page 14-26 .

Stepping Through the Simulation

In this section...
“Overview of Stepping” on page 14-26
“How to Step” on page 14-27
“Choosing the Granularity of a Step” on page 14-28
“Tips for Stepping Through the Simulation” on page 14-29

Overview of Stepping

Using the SimEvents debugger, you can proceed step by step in the simulation. After each step, you can inspect states or issue other commands at the `sedebug>>` prompt.

When to Step

Stepping is appropriate if one of these is true:

- You want to see what happens next in the simulation and you want frequent opportunities to inspect states as the simulation proceeds.
- You want the simulation to proceed but cannot formulate a condition suitable for a breakpoint.

When Not to Step

Stepping is not the best way to proceed with the simulation in the debugger if one of these is true:

- You want the simulation to proceed until it satisfies a condition that you can formulate using a breakpoint, and you do not need to enter debugging commands until the condition is satisfied. In this case, using breakpoints might require you to enter fewer commands compared to stepping; for details, see “Using Breakpoints During Debugging” on page 14-62.
- You want the simulation to proceed until the end, and you do not need to enter other commands. In this case, at the `sedebug>>` prompt, enter `runtoend` instead of stepping repeatedly.

- You want the simulation to proceed until the end, and you need to enter commands only at the end of the simulation. In this case, remove or disable any breakpoints you might have set earlier, and, at the `sedebug>>` prompt, enter `cont` instead of stepping repeatedly.

How to Step

If you have decided that stepping is appropriate for your debugging needs, at the `sedebug>>` prompt, enter one of the commands in the next table. To learn about the choices for granularity of steps, see “Choosing the Granularity of a Step” on page 14-28.

If Latest Message in Simulation Log Is...	And You Want to...	At <code>sedebug>></code> Prompt, Enter...
An independent operation (not indented)	Take the smallest possible step	<code>step</code> or <code>step in</code>
	Skip consequences of the current operation and stop at the next independent operation that appears in the simulation log	<code>step over</code>
A dependent operation (indented)	Take the smallest possible step	<code>step</code> or <code>step in</code> or <code>step over</code>
	Skip remaining consequences of the previous independent operation and stop at the next independent operation that appears in the simulation log	<code>step out</code>

As a result, the simulation proceeds and the simulation log displays one or more messages to reflect the simulation progress.

For an example, see “Building a Simple Hybrid Model” (“Confirming Event-Based Behavior Using the SimEvents Debugger” section).

Choosing the Granularity of a Step

Using the SimEvents debugger, you can proceed in the simulation by an amount that corresponds to one message in the simulation log, or a collection of messages. The endpoint of a step depends on these factors:

- What is happening in the simulation.

As the section “Interpreting the Simulation Log” on page 14-11 describes, the simulation log does not use a strictly time-based, block-based, or entity-based approach to determine the messages that appear. Similarly, proceeding step by step through a simulation in the SimEvents debugger does not use a strictly time-based, block-based, or entity-based approach to determine the endpoints of the steps.

- The detail settings in effect before you invoke `step`.

If you change the detail settings from their default values to cause the simulation log to omit entity messages or event messages, you cannot step to an operation that corresponds to an omitted message unless a breakpoint coincides with the operation. For instance, see Example: Skipping Entity Operations When Stepping on page 14-50.

In particular, if your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other than breakpoints.

- The step size you choose when you invoke `step`. Choices are described, following.

Taking the Smallest Possible Step

The smallest possible step in the SimEvents debugger corresponds to one message in the simulation log. Taking the smallest possible step is appropriate if one of these is true:

- You are not sure what the simulation will skip if you take a larger step, and you want as many opportunities as possible to inspect states as the simulation proceeds. This might be true if you are new to SimEvents software, new to the debugger, unfamiliar with the model you are debugging, or unsure where to look for a simulation problem you are trying to diagnose.

- You know that if you take a larger step, the simulation will skip a point in the simulation at which you want to inspect states.

Taking a Larger Step By Skipping Consequences

A potentially larger step in the SimEvents debugger corresponds to a series message in the simulation log. The last message in the series is not indented, while other messages in the series are indented to show that they represent consequences of an earlier operation. Taking a larger step is appropriate if one of these is true:

- The current operation is not relevant to you and you want to skip over its immediate consequences.
- You find it more efficient to scan a series of messages visually than enter a command interactively after viewing each message, and you do not need to inspect states at intermediate points in the larger step.
- You find it easier to understand a series of related messages when all are visible, and you do not need to inspect states at intermediate points in the larger step.

For Further Information

- “Simulation Log in the Debugger” on page 14-8
- “Customizing the Debugger Simulation Log” on page 14-47
- “Independent Operations and Consequences in the Debugger” on page 14-20

Tips for Stepping Through the Simulation

- **Ensuring the action has happened** — If you want to inspect states to confirm the effect of the action in the most recent message in the simulation log, first ensure that the action has happened. If the message uses an action statement such as “executing,” use `step` before inspecting states. If the message uses a verb in the past tense, such as “detected,” the extra step is not necessary because the action has already happened.
- **Clicking shortcuts** — If you use a certain `step` command frequently, a shortcut you can click might provide an efficient way to issue the command repeatedly. To learn about shortcuts, see “Running Frequently Used

Statement Groups with MATLAB Shortcuts” in the MATLAB Desktop Tools and Development Environment documentation.

- **Connection between step and detail** — If your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other than breakpoints. In the absence of breakpoints, a step causes the simulation to proceed until the end. If you inadvertently reach the end of the simulation in this way and want to return to the point in the simulation from which you tried to step, use information in the Command Window to set a breakpoint in a subsequent debugging session. For example, if the last message in the simulation log before you inadvertently stepped too far indicates the execution of event `ev5`, you can enter `evbreak ev5; cont` in the next debugger session.

Inspecting the Current Point in the Debugger

In this section...

“Viewing the Current Operation” on page 14-31

“Obtaining Information Associated with the Current Operation” on page 14-32

Viewing the Current Operation

The simulation log displays information about what is about to happen or what has just happened in the simulation. If the log is no longer visible in the Command Window because of subsequent commands and output displays, you can redisplay the most recent log entry by entering this command at the `sedebug>>` prompt:

```
currentop
```

If the most recent log entry represents a dependent operation, the output in the Command Window also includes the current top-level independent operation being executed. To learn more about dependent and independent operations, see “Independent Operations and Consequences in the Debugger” on page 14-20

Comparing Current Operation Display with Simulation Log

The sample output shows that the display of the current operation includes the last entry in the simulation log, which is the current operation. The display also includes the last unindented entry in the simulation log, representing the latest independent operation.

```

*** SimEvents Debugger ***

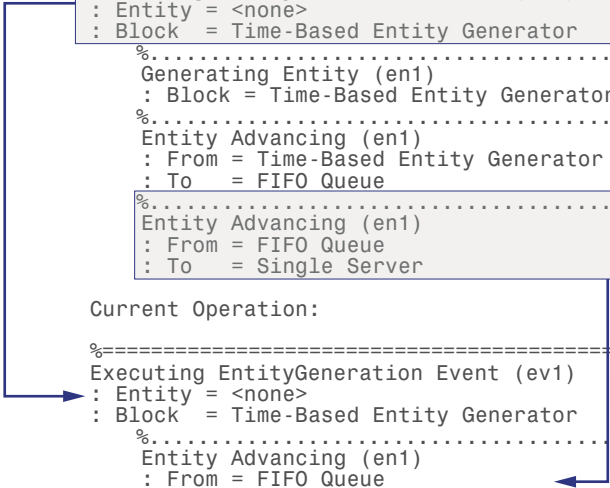
Functions | Quick Start | Debugger Help

%=====
Initializing Model sedemo_event_priorities
%=====
Initializing Time-Based Entity Generators
%.....
Scheduling EntityGeneration Event (ev1)
: EventTime = 0.000000000000000 (Now)
: Priority = 2
: Block = Time-Based Entity Generator
%.....
Executing EntityGeneration Event (ev1)
: Entity = <none>
: Block = Time-Based Entity Generator
%.....
Generating Entity (en1)
: Block = Time-Based Entity Generator
%.....
Entity Advancing (en1)
: From = Time-Based Entity Generator
: To = FIFO Queue
%.....
Entity Advancing (en1)
: From = FIFO Queue
: To = Single Server
%.....
Current Operation:
%=====
Executing EntityGeneration Event (ev1)
: Entity = <none>
: Block = Time-Based Entity Generator
%.....
Entity Advancing (en1)
: From = FIFO Queue
: To = Single Server
%.....
SimEvents Debugger: Abort

```

Independent operation

Dependent operation



Obtaining Information Associated with the Current Operation

You can get some information about the current operation in the form of variables in the workspace using commands like those listed in the next table. Variables containing identifiers can be useful as inputs to state inspection functions. For details, see “Inspecting Entities, Blocks, and Events” on page 14-34.

Information	At sedebug>> Prompt, Enter...
Current simulation time	t = simtime
Identifier of the entity that undergoes the current operation	enid = gcen
Identifier of the block associated with the current operation	blkid = gcebid
Path name of the block associated with the current operation	blkname = gceb
Identifier of the event being scheduled, executed, or canceled on the event calendar as part of the current operation	evid = gcev

Inspecting Entities, Blocks, and Events

In this section...

“Inspecting Entities” on page 14-34

“Inspecting Blocks” on page 14-36

“Inspecting Events” on page 14-38

“Obtaining Identifiers of Entities, Blocks, and Events” on page 14-38

Inspecting Entities

These sections provide procedures and background information about inspecting entities:

- “Inspecting Location, Scalar Attributes, Timeouts, and Timers” on page 14-34
- “Inspecting Nonscalar Attribute Values” on page 14-35
- “Interpretation of Entity Location” on page 14-36

For an example, see the `sedb.eninfo` reference page.

Inspecting Location, Scalar Attributes, Timeouts, and Timers

If you expect all attributes of an entity to have scalar values or if knowing the sizes of nonscalar attribute values is sufficient, then this procedure is the simplest way to inspect the entity:

- 1 Find the entity identifier using the simulation log or one of the approaches listed in Obtaining Entity Identifiers on page 14-39.
- 2 At the `sedebg>>` prompt, enter this command, where `enid` is the entity identifier:

```
eninfo enid % enid is the identifier.
```

The resulting display includes this information:

- Current simulation time

- Location of the entity
- Names of attributes of the entity
- Scalar attribute values and sizes of nonscalar attribute values
- Tags, scheduled times, and event identifiers of timeouts
- Tags and elapsed times of timers

Inspecting Nonscalar Attribute Values

To inspect nonscalar values of attributes of an entity:

- 1** Find the entity identifier using the simulation log or one of the functions listed in Obtaining Entity Identifiers on page 14-39.
- 2** At the `sedebug>>` prompt, enter this command. `enid` is a string representing the entity identifier and `en_struct` is the name of a variable you want to create in the workspace:

```
en_struct = eninfo(enid); % enid is the identifier.
```

The output variable `en_struct` is a structure that stores this information:

- Current simulation time
 - Location of the entity
 - Names and values of the attributes of the entity
 - Tags, scheduled times, and event identifiers of timeouts
 - Tags and elapsed times of timers
- 3** Use dot notation to access values of any attribute as part of the `Attributes` field of `en_struct`.

For example, to access the value of an attribute called `Attribute1`, use the notation `en_struct.Attributes.Attribute1`. The field name `Attributes` is fixed but the names `en_struct` and `Attribute1` depend on names you choose for the variable and attribute.

Interpretation of Entity Location

Blocks that possess entity input ports act as storage or nonstorage blocks. Only storage blocks are capable of holding an entity for a nonzero duration. If the debugger reports a nonstorage block as the location of an entity, it means that the debugger has suspended the simulation while the entity is in the process of advancing through a nonstorage block toward a storage block or a block that destroys entities. Before the simulation clock moves ahead, the entity will either arrive at a storage block or be destroyed.

For lists of storage and nonstorage blocks, see “Storage and Nonstorage Blocks” on page 16-37.

To inspect blocks instead of entities, see “Inspecting Blocks” on page 14-36.

Inspecting Blocks

- “Procedure for Inspecting Blocks” on page 14-36
- “Result of Inspecting Queue Blocks” on page 14-37
- “Result of Inspecting Server Blocks” on page 14-37
- “Result of Inspecting Other Storage Blocks” on page 14-37
- “Result of Inspecting Nonstorage Blocks” on page 14-38

Procedure for Inspecting Blocks

Before inspecting a block, determine if it is capable of providing information. One way to do this is to see whether the block appears in the output of `blklist`. If the block is capable of providing information, use this procedure to get the information:

- 1 Find a unique way to refer to the block using one of these approaches:
 - Find the block identifier using one of the approaches listed in Obtaining Block Identifiers on page 14-39.
 - Find the path name of the block by selecting the block and using `gcb`, or by typing the path name. If you type the path name, be careful to reflect space characters and line breaks accurately.

- 2** At the `sedebug>>` prompt, enter this command , where `thisblk` is a string representing the block identifier or path name:

```
blkinfo(thisblk); % thisblk is the identifier or path name.
```

The resulting information depends on the kind of block you are inspecting.

Result of Inspecting Queue Blocks

When you inspect queue blocks, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing
- Number of entities that the block can store at a time
- Status of each stored entity with respect to the block
- Length of time each stored entity has been in the queue

Result of Inspecting Server Blocks

When you inspect server blocks, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing
- Number of entities that the block can store at a time
- Status of each stored entity with respect to the block
- Identifier and scheduled time of the service completion event for each entity

Result of Inspecting Other Storage Blocks

When you inspect storage blocks other than queues and servers, the display includes this information:

- Current simulation time
- Identifiers of entities that the block is currently storing

For the Output Switch block with the **Store entity before switching** option selected, the resulting display also indicates which entity output port is the selected port.

For a list of storage blocks, see “Storage and Nonstorage Blocks” on page 16-37.

Result of Inspecting Nonstorage Blocks

When you inspect nonstorage blocks, the display includes this information:

- Current simulation time
- Identifier of an entity that is currently advancing through the block. To learn what it means for a nonstorage block to be the location of an entity, see “Interpretation of Entity Location” on page 14-36.

Depending on the block, the display might also include additional information. For details, see the Block and Description columns of the Fields of Output Structure table on the `sedb.blkinfo` reference page.

Inspecting Events

To get details about a particular event on the event calendar:

- 1 Find the event identifier using the simulation log or one of the approaches listed in Obtaining Event Identifiers on page 14-40.
- 2 At the `sedebug>>` prompt, enter this command, where `evid` is the event identifier:

```
evinfo evid % evid is the identifier.
```

Alternatively, at the `sedebug>>` prompt, enter `evcal` to get details about all events on the event calendar. For more information, see the `sedb.evcal` reference page.

Obtaining Identifiers of Entities, Blocks, and Events

In some state inspection functions, you must refer to an entity, block, or event using its identifier. For background information about identifiers, see

“Identifiers in the Debugger” on page 14-18. The next tables suggest ways to obtain identifiers to use as input arguments in state inspection commands.

Obtaining Entity Identifiers

To Display Identifier of Entity Associated with...	At <code>sedebug>></code> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcen</code>	<code>sedb.gcen</code>
A particular block whose identifier or path name you know	<code>blkinfo(...)</code> . Look in the ID column in the resulting tabular display.	<code>sedb.blkinfo</code>
Events on the event calendar	<code>evcal</code> . Look in the Entity column in the resulting tabular display.	<code>sedb.evcal</code>
A particular event whose identifier you know	<code>evinfo(...)</code> . Look at the Entity entry.	<code>sedb.evinfo</code>

Obtaining Block Identifiers

To Display Identifier of Block Associated with...	At <code>sedebug>></code> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcebid</code>	<code>sedb.gcebid</code>
All blocks whose states you can inspect	<code>blklist</code> . Look in the first column in the resulting tabular display.	<code>sedb.blklist</code>

Obtaining Event Identifiers

To Display Identifier of Event Associated with...	At sedebug>> Prompt, Enter...	Link to Reference Page
The current operation	<code>gcev</code>	<code>sedb.gcev</code>
Events on the event calendar	<code>evcal</code> . Look in the ID column in the resulting tabular display.	<code>sedb.evc1</code>

Working with Debugging Information in Variables

In this section...

“Comparison of Variables with Inspection Displays” on page 14-41

“Functions That Return Debugging Information in Variables” on page 14-41

“How to Create Variables Using State Inspection Functions” on page 14-42

“Tips for Manipulating Structures and Cell Arrays” on page 14-43

“Example: Finding the Number of Entities in Busy Servers” on page 14-43

Comparison of Variables with Inspection Displays

State inspection functions in the SimEvents debugger enable you to view information in the Command Window, as the sections “Inspecting Entities, Blocks, and Events” on page 14-34 and “Viewing the Event Calendar” on page 14-46 describe. An alternative way to capture the same information is in a variable in the MATLAB base workspace. Capturing information in a workspace variable lets you accomplish these goals:

- Use information from one command as an input to another command, without having to type or paste information from the Command Window.
- Cache information at a certain point in the simulation, for comparison with updated information at a later point in the simulation. The workspace variable remains fixed as the simulation proceeds.
- Store information in a MAT-file and use it in a different MATLAB software session.

Functions That Return Debugging Information in Variables

The table lists state inspection functions that can return variables in the workspace.

Function	Returns	Class
<code>blkinfo</code>	Block information	Structure
<code>blklist</code>	Blocks and their identifiers	Cell

Function	Returns	Class
<code>eninfo</code>	Entity information	Structure
<code>evcal</code>	Event calendar	Structure
<code>evinfo</code>	Event information	Structure
<code>gceb</code>	Path name of the block associated with the current operation	String (char)
<code>gcebid</code>	Identifier of the block associated with the current operation	String (char)
<code>gcen</code>	Identifier of the entity that undergoes the current operation	String (char)
<code>gcev</code>	Identifier of the event associated with the current operation	String (char)
<code>simtime</code>	Current simulation time	Numeric (double)

How to Create Variables Using State Inspection Functions

To create a variable using a state inspection function, follow these rules:

- Name an output variable in the syntax that you use to invoke the function, such as `my_entity_id = gcen`. The function creates the output variable in the workspace.
- If the function requires input arguments, express them using the functional form of the syntax, such as `evinfo('ev1')`, rather than the command form, `evinfo ev1`.

To learn about functional and command forms of syntax, see “Calling Functions” in the MATLAB Programming Fundamentals documentation.

To learn how to formulate input arguments for `blkinfo`, `eninfo`, and `evinfo`, see “Obtaining Identifiers of Entities, Blocks, and Events” on page 14-38.

For details about the information each function returns in the output variable, see the corresponding function reference page.

Tips for Manipulating Structures and Cell Arrays

Full details about structures and cell arrays are in “Structures” and “Cell Arrays”, both in the MATLAB Programming Fundamentals documentation. Tips that you might find useful for working with the variables that debugger functions return are:

- If a cell array shows something like `[1x51 char]` instead of exact block path names when you enter the variable name alone at the command prompt, use content indexing to display the cell contents explicitly. Content indexing uses curly braces, `{}`. For an example, see the `sedb.blklist` reference page.
- To access information in nested structures, append nested field names using dot notation. For an example, see the `sedb.eninfo` reference page.
- You can assign numeric values of like-named fields in a structure array to a numeric vector. To do this, enclose the `array.field` expression in square brackets, `[]`. For an example, see the `sedb.evcal` reference page.
- You can assign string values of like-named fields in a structure array to a cell array. To do this, enclose the `array.field` expression in curly braces, `{}`. For examples, see the `sedb.blkinfo` and `sedb.breakpoints` reference pages.
- You can gather information about like-named fields in a structure array that satisfy certain criteria, by invoking `find` and using its output to index into the structure array. For examples, see the `sedb.evcal` and `sedb.breakpoints` reference pages.

Example: Finding the Number of Entities in Busy Servers

This example illustrates ways that you can use information from one command as an input to another command. Suppose your system includes several server blocks and you want to see how many entities are in each server block that is currently busy serving an entity. The following code inspects the event calendar to locate service completion events, uses the events to locate server blocks that are currently busy, and inspects server blocks to find out how many entities are in them. An entity in the server might be in service or waiting to depart.

- 1 Begin a debugger session for a particular model by entering this command at the MATLAB command prompt:

```
sedebug('sedemo_star_routing')
```

- 2 Proceed in the simulation. At the `sedebug>>` prompt, enter:

```
tbreak 5
cont
```

The output ends with a message describing the context of the simulation shortly after $T = 5$:

```
Hit b1 : Breakpoint for first operation at or after time 5.000000

%=====
Executing ServiceCompletion Event (ev29)           Time = 5.189503930558476
: Entity = en4                                     Priority = 5
: Block = Distribution Center/Infinite Server
```

- 3 To find out how many entities are in each server that is currently busy serving, use a series of state inspection and variable manipulation commands:

```
% Get the event calendar.
eventcalendar = evcal;

% Combine executing and pending events, to search both.
allevents = [eventcalendar.ExecutingEvent; eventcalendar.PendingEvents];

% Find service completion events.
idx = cellfun(@(x) isequal(x,'ServiceCompletion'), {allevents.EventType});
svc_completions = allevents(idx);

% Find the unique server blocks.
svrs = unique({svc_completions.Block});

% Compute the number of server blocks.
num = length(svrs);

% Preallocate an array for the results.
n = zeros(1,num);
```



```
% Loop over the server blocks and find the number of entities
% in each block.
for jj=1:num
    s = blkinfo(svrs{jj});
    n(jj)=length(s.Entities);
    disp(sprintf('%s: %d',svrs{jj},n(jj)))
end
```

The output is:

```
sedemo_star_routing/Distribution Center/Infinite Server: 1
sedemo_star_routing/Service Station 3/Infinite Server3: 1
sedemo_star_routing/Service Station 4/Infinite Server4: 2
```

4 End the debugger session. At the `sedebg>>` prompt, enter:

```
sedb.quit
```

Viewing the Event Calendar

To view a list of events on the event calendar, at the `sedebug>>` prompt, enter this command:

```
evcal
```

The output in the Command Window includes the current simulation time and a tabular display of all events in the event calendar. Each listing in the event calendar display includes the event characteristics described in “Optional Displays of Event Calendar in the Simulation Log” on page 14-10.

The event in progress or selected for execution appears with notation `=>` to the left of the event identifier. This event appears in the display until the completion of all immediate consequences, and then the event is removed from the calendar.

When comparing the simulation log with the event calendar display, the sequence on the calendar and the sequence in which the events were scheduled might differ, even for simultaneous events having equal priority.

For Further Information

- “Event Sequencing” on page 16-2
- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-10

Customizing the Debugger Simulation Log

In this section...

“Customizable Information in the Simulation Log” on page 14-47

“Tips for Choosing Appropriate Detail Settings” on page 14-48

“Effect of Detail Settings on Stepping” on page 14-49

“How to View Current Detail Settings” on page 14-51

“How to Change Detail Settings” on page 14-52

“How to Save and Restore Detail Settings” on page 14-53

Customizable Information in the Simulation Log

You can configure the SimEvents debugger to include or omit certain kinds of messages in its simulation log. You can focus on the messages that are relevant to you by omitting irrelevant messages.

The overall configuration regarding including or omitting messages is called the debugger’s *detail settings*. Each individual category of messages that you can include or omit corresponds to an individual detail setting. A particular detail setting is on if the simulation log includes messages in the corresponding category. The setting is off if the simulation log omits messages in the category.

The `detail` function lets you configure and view detail settings. The following table lists the available detail settings and the programmatic names that you see as inputs or field names when you use the `detail` function.

Category of Messages	Programmatic Name in detail Function	Further Information About Messages in Category
Event operations, except independent operations representing event executions	ev	“Event Scheduling, Execution, and Cancellation Messages” on page 14-13
Entity operations	en	“Entity Operation Messages” on page 14-16
Event calendar information	cal	“Optional Displays of Event Calendar in the Simulation Log” on page 14-10

Messages that you cannot selectively omit from the simulation log include messages about independent operations, such as the execution of events on the event calendar. Messages about independent operations appear whenever any of the detail settings is on. The reason for including messages about independent operations is that they provide a context in which you can understand other messages. To learn more about independent operations, see “Independent Operations and Consequences in the Debugger” on page 14-20.

Tips for Choosing Appropriate Detail Settings

Different debugging scenarios benefit from different detail settings. Use these suggestions to help you choose appropriate settings for your needs:

- When using `step` to proceed in the simulation, include event or entity operations in the simulation log if you want the debugger to suspend the simulation at such operations to let you inspect states.
- When using breakpoints to skip a large portion of the simulation that is irrelevant to you, omitting all messages (`detail none`) can prevent the Command Window from accumulating a lot of simulation log text that does not interest you.

Tip If you use `step` from a breakpoint onward, turn on at least one of the detail settings, or else you will likely step too far. For details, see “Effect of Detail Settings on Stepping” on page 14-49.

- If you want to inspect final states but are not interested in the simulation log, omit all messages (`detail none`) and then enter `cont`. The simulation proceeds until the built-in breakpoint at the end of the simulation, with minimal text in the simulation log.
- If you want to focus on entities instead of events, try `detail('en',1,'ev',0)`. The simulation log still shows independent operations, but the dependent operations in the simulation log exclude event scheduling, cancelation, and execution operations.
- If you want to focus on events instead of entities, try `detail('en',0,'ev',1)` or `detail('en',0,'ev',1,'cal',1)`.
- If you are not sure which messages in the simulation log might be useful for later reference, consider these approaches:
 - If you include less information in the simulation log, the simulation might run more quickly and the simulation log might be more manageable. However, you risk missing important information and having to run the simulation again to see that information.
 - If you include more information in the simulation log, it might be harder for you to find the relevant portions. Using **Edit > Find** in the Command Window might help you see the portions of interest to you.
- Reducing textual output in the Command Window can save time.

Effect of Detail Settings on Stepping

The behavior of `detail` and `step` are related. If you change the detail settings from their default values to cause the simulation log to omit entity messages or event messages, then you cannot step to an operation that corresponds to an omitted message, unless a breakpoint coincides with the operation. A specific example of this behavior follows.

If your detail settings cause the simulation log to omit all messages (equivalent to the `detail none` command), you cannot step to anything other

than breakpoints. In the absence of breakpoints, a step causes the simulation to proceed until the end. If you inadvertently reach the end of the simulation in this way and want to return to the point in the simulation from which you tried to step, use a time or event identifier that appears in the Command Window to set a breakpoint in a subsequent debugging session. For example, if the last message in the simulation log, before you inadvertently stepped too far, indicates the execution of event `ev5`, then you can exit the debugger session, restart it, and enter `evbreak ev5; cont`.

Example: Skipping Entity Operations When Stepping

When the detail setting for entity operations is off, the simulation log omits messages about entity operations and you cannot step to entity operations.

- 1 Open and debug a model. At the MATLAB command prompt, enter:

```
simeventsdocex('doc_sl-demo_f14_des')
sedebug('doc_sl-demo_f14_des')
```

- 2 Omit entity operation messages. At the `sedebug>>` prompt, enter:

```
detail('en',0)
```

- 3 Proceed with the simulation. At the `sedebug>>` prompt, enter the following command six times in succession:

```
step
```

The output reflects that when you step through the simulation, the debugger does *not* suspend the simulation upon the entity's generation, advancement, attribute assignment, or destruction. The reason is that the detail setting for entity operations is off.

```
%=====
Detected Sample-Time Hit                               Time = 0.0000000000000000
: Block = Subsystem/Event-Based Entity Generator
%.....%
Scheduling EntityGeneration Event (ev1)
: EventTime = 0.0000000000000000 (Now)
: Priority   = SYS1
: Block     = Subsystem/Event-Based Entity Generator
```

```

%=====
Executing EntityGeneration Event (ev1)                Time = 0.0000000000000000
: Entity = <none>                                     Priority = SYS1
: Block = Subsystem/Event-Based Entity Generator
%.....%
Scheduling ServiceCompletion Event (ev2)
: EventTime = 0.055383948677907
: Priority = 500
: Entity = en1
: Block = Subsystem/Infinite Server
%=====
Executing ServiceCompletion Event (ev2)                Time = 0.055383948677907
: Entity = en1                                       Priority = 500
: Block = Subsystem/Infinite Server
%=====
Detected Sample-Time Hit                             Time = 0.1000000000000000
: Block = Subsystem/Event-Based Entity Generator

```

To see which entity operations the debugger omits, compare your output with the items in the table in “Confirming Event-Based Behavior Using the SimEvents Debugger” in the SimEvents getting started documentation. Items 4, 5, 6, 7, 10, 11, and 12 in the table do not appear in this example because these items represent entity operations.

4 End the debugger session. At the `sedebg>>` prompt, enter:

```
sedb.quit
```

How to View Current Detail Settings

To view the current detail settings in the Command Window, at the `sedebg>>` prompt, enter:

```
detail
```

The output looks like this, where the `on` and `off` values depend on your current detail settings:

```

Event Operations (ev)   : off
Entity Operations (en)  : on
Event Calendar (cal)   : off

```

If a line in the output says `on`, the simulation log shows the corresponding type of message. Otherwise, the simulation log omits the corresponding type of message.

How to Change Detail Settings

To change detail settings, enter a `detail` command that includes one or more inputs. For available syntaxes, see the reference page for the `sedb.detail` function.

Tips to help you select a suitable syntax are:

- To change one or more detail settings but not all detail settings, use inputs that specify the name and new value of the detail settings you want to change. Detail settings you omit from the syntax remain unchanged.
- To change all detail settings, use inputs that specify the name and new value of all detail settings. One way to ensure that you include all settings in the command is to use a structure variable containing the current settings as a starting point. For an example, see [Example: Including Event Calendar Information Using a Structure](#) on page 14-53.
- To make it easier to restore previous settings later, use a syntax that includes an output. For details, see [“How to Save and Restore Detail Settings”](#) on page 14-53.
- To learn which messages in the simulation log correspond to each detail setting, see [“Customizable Information in the Simulation Log”](#) on page 14-47.
- For suggestions on choosing which messages to include or omit based on your debugging situation, see [“Tips for Choosing Appropriate Detail Settings”](#) on page 14-48.

Example: Including Event Calendar Information Using a Parameter/Value Pair

To cause the simulation log to include event calendar information but not change any other detail settings, at the `sedebug>>` prompt, enter:

```
detail('cal',1)
```


The output includes the following line confirming the change:

```
Event Calendar    (cal)  :  on
```

Example: Including Event Calendar Information Using a Structure

Entering these commands at the `sedebg>>` prompt has the same effect as the example above and also creates a structure variable, `s`, that records the new detail settings:

```
s = detail; % Get current detail settings.
s.cal = 1; % Change value of cal field of structure s.
detail(s); % Use s to change cal detail setting.
```

How to Save and Restore Detail Settings

If you expect to change detail settings frequently or temporarily during a debugger session, you can use an output from the `detail` function to facilitate restoring previous settings. Use this procedure:

- 1** When changing detail settings, enter a `detail` command that includes an output. The output variable records the settings *before* they change to correspond to the inputs that you specify in your command.
- 2** When you want to restore the earlier detail settings, use the variable as an input to `detail`.

As a special case, you can restore default detail settings. At the `sedebg>>` prompt, enter `detail default`.

Example: Omitting and Reinstating Entity Messages

To cause the simulation log to include event calendar information and also create a structure variable, `prev`, that records the previous detail settings, at the `sedebg>>` prompt, enter:

```
prev = detail('cal',1) % Record settings and then change them.
```

The next command restores the earlier settings:

```
detail(prev) % Restore previous settings.
```

Debugger Efficiency Tips

In this section...

“Executing Commands Automatically When the Debugger Starts” on page 14-54

“Creating Shortcuts for Debugger Commands” on page 14-55

Executing Commands Automatically When the Debugger Starts

If you want to execute one or more commands in the debugger immediately after initializing the model, you can include those commands when you invoke `sedebug`. You might find executing commands automatically to be useful for setting the same breakpoints or detail settings across multiple debugging sessions, or helping someone reproduce a problem that you are seeing in a model.

To start a debugger session that executes commands automatically:

- 1 Create an empty options structure using the `se_getdbopts` function.

```
opts_struct = se_getdbopts;
```

- 2 Define the `StartFcn` field of the options structure as a cell array of strings, where each string is an individual command that you want to execute after initializing the model. Here is an example:

```
opts_struct.StartFcn = {'detail('cal',1)', 'tbreak 5'};
```

- 3 Start a debugger session using the `sedebug` function with the options structure as the second input argument. Here is an example:

```
sedebug('sedemo_event_priorities', opts_struct)
```

- 4 End the debugger session. At the `sedebug>>` prompt, enter:

```
sedb.quit
```

For an example, see the `se_getdbopts` reference page.

Tips for Creating a StartFcn Cell Array

- If you want to execute commands and then exit the debugging session automatically, include the string 'quit' at the end of the StartFcn array. If you want to execute commands automatically and then interact with the debugger, do not include the string 'quit' in the StartFcn array.
- If you want to set breakpoints at the beginning of the debugging session, have just ended a debugging session on the same model and have not changed the model, you can use the identifiers that occurred in the previous debugging session.

Creating Shortcuts for Debugger Commands

If you use a particular debugger command frequently, such as `step` or `step over`, a shortcut you can click might provide an efficient way to issue the command repeatedly. To learn about shortcuts, see “Running Frequently Used Statement Groups with MATLAB Shortcuts” in the MATLAB Desktop Tools and Development Environment documentation.

Defining a Breakpoint

In this section...
“What Is a Breakpoint?” on page 14-56
“Identifying a Point of Interest” on page 14-56
“Setting a Breakpoint” on page 14-58
“Viewing All Breakpoints” on page 14-60

What Is a Breakpoint?

In the SimEvents debugger, a breakpoint is a point of interest in the simulation at which the debugger can suspend the simulation and let you enter commands. You decide which points are of interest to you and then use debugger functions to designate those points as debugging breakpoints. After you define one or more breakpoints, you can use them to control the simulation process efficiently. At the `sedebug>>` prompt, the `cont` command causes the simulation to proceed until the next breakpoint, bypassing points that you are not interested in and letting you inspect states at a point of interest. To learn more about controlling the simulation after defining breakpoints, see “Using Breakpoints During Debugging” on page 14-62.

Identifying a Point of Interest

Before defining a breakpoint, you must decide what points in the simulation you want to inspect and then determine a way to refer to the point explicitly when invoking a function to define a breakpoint. The SimEvents debugger supports the following kinds of breakpoints.

Type of Breakpoint	Debugger Suspends Simulation Upon...	When You Might Use Breakpoint Type
Timed breakpoint	First operation whose associated time is equal to or greater than the value of the timed breakpoint	<ul style="list-style-type: none"> • You know a time at which something of interest to you occurs • You want to proceed in the simulation by a fixed amount of time • A point of interest is not associated with an event on the event calendar or a block in the model • You do not have an event identifier to use to define an event breakpoint
Event breakpoint	Execution or cancelation of the specified event	An event on the event calendar is a point of interest
Block breakpoint	Operation involving the specified block, for blocks that support block breakpoints	<ul style="list-style-type: none"> • You want to understand how a block behaves • A block seems to cause or reflect the problem you are investigating

Tips for Identifying Points of Interest

- To see a list of all events on the event calendar and their event identifiers, at the `sedebg>>` prompt, enter `evcal`.
- To see a list of blocks in the model that support block breakpoints, at the `sedebg>>` prompt, enter `blklist`. The list also shows the block identifiers. For a list of block operations at which the debugger can suspend the simulation, see “Block Operations Relevant for Block Breakpoints” on page 14-65.
- To proceed in the simulation by a fixed amount of time, define a timed breakpoint whose value is relative to the current simulation time using syntax such as `tbreak(simtime + fixed_amount)`. For an example, see the `sedb.simtime` reference page.
- To investigate the behavior of a block only during particular time intervals, use a combination of timed breakpoints and block breakpoints. During a time interval of interest, the block breakpoint helps you investigate the

behavior of the block. When the simulation advances beyond that time interval, you can disable the block breakpoint and use a timed breakpoint to advance to another time interval of interest. To learn more about disabling breakpoints, see “Ignoring or Removing Breakpoints” on page 14-63.

- To see all actions that happen at a particular time, use a pair of timed breakpoints, as in “Using Nearby Breakpoints to Focus on a Particular Time” on page 3-5.
- You might need or want to iteratively refine your points of interest across multiple simulation runs. For example:
 - A plot of a signal against time might indicate when something of interest to you happens in the simulation. You can read the approximate time from the plot. You can use the approximate time when defining a timed breakpoint in a subsequent run of the simulation.
 - You can use a pair of timed breakpoints to examine simulation behavior in a time interval and find a relevant event on the event calendar. You can use the event identifier when defining an event breakpoint in a subsequent run of the simulation.
- An event breakpoint is not the same as a timed breakpoint whose value equals the scheduled time of the event. The two breakpoints can cause the simulation to stop at different points if the execution or cancelation of the event is not the first thing that happens at that value of time. For an example, see the `sedb.evbreak` reference page.

Setting a Breakpoint

After you have identified a point of interest, you can set a breakpoint by entering one of the commands in the table.

Type of Breakpoint	At sedebug>> Prompt, Enter...
Timed breakpoint at simulation time $T = t_0$	tbreak(t0) or tbreak t0
Event breakpoint at event whose identifier is the string, evid	evbreak(evid)
Block breakpoint at block whose identifier is the string, blkid, or whose path name is the string, blkname	blkbreak(blkid) or blkbreak(blkname)

Warning When Setting Certain Breakpoints

The debugger warns you if it determines that it might not hit the breakpoint that you want to define or if it does not recognize an event identifier that you specify. The warning can alert you to a mistake in your command, but might also follow a correct command. For example, suppose you obtain an event identifier during one run of the simulation and set an event breakpoint on that event in a subsequent run of the simulation, *before* the event has been scheduled. Setting an event breakpoint before the event has been scheduled is legitimate because event identifiers are the same from one debugging session to the next. However, the debugger cannot distinguish this situation from a mistake in your input argument to `evbreak`.

If you often intentionally set breakpoints that cause this warning and you want to suppress such warnings in the future, enter `warning off last` immediately after the warning occurs. For more information about this command, see “Warning Control” in the MATLAB Programming Fundamentals documentation.

Viewing All Breakpoints

To see a tabular display of all breakpoints that you have set, at the `sedebug>>` prompt, enter this command:

```
breakpoints
```

The output includes this information about each breakpoint.

Label	Description
ID	A token that uniquely identifies the breakpoint
Type	Block, Event, or Timed
Value	The block identifier of a block breakpoint
	The event identifier of an event breakpoint
	The time of a timed breakpoint
Enabled	yes, if the debugger considers the breakpoint when determining where to suspend the simulation
	no, if the debugger ignores the breakpoint

The list of breakpoints does not guarantee that the simulation reaches each point before the simulation ends. The sequence of breakpoints in the list does not necessarily represent the sequence in which the simulation reaches each point.

The list of breakpoints does not show a special built-in breakpoint that the debugger always observes at the end of the simulation. You do not set this breakpoint explicitly and you cannot disable or remove it.

Sample Breakpoint List

The sample output of `breakpoints` shows five breakpoints. Two are timed breakpoints, one is an event breakpoint, and two are block breakpoints. One of the block breakpoints is disabled.

List of Breakpoints:

ID	Type	Value	Enabled
b1	Event	ev4	yes

b2	Block	blk11	yes
b3	Timed	100	yes
b4	Timed	101	yes
b5	Block	blk15	no

To learn how to delete or disable breakpoints in the list, see “Ignoring or Removing Breakpoints” on page 14-63.

To learn how to enable breakpoints in the list, see “Enabling a Disabled Breakpoint” on page 14-64.

Using Breakpoints During Debugging

In this section...

“Running the Simulation Until the Next Breakpoint” on page 14-62

“Ignoring or Removing Breakpoints” on page 14-63

“Enabling a Disabled Breakpoint” on page 14-64

Running the Simulation Until the Next Breakpoint

By default, the debugger has a special built-in breakpoint at the end of the simulation. You can define your own breakpoints, as described in “Defining a Breakpoint” on page 14-56. To proceed in the simulation until the debugger reaches the next breakpoint, at the `sedebug>>` prompt, enter this command:

```
cont
```

Point at Which the Debugger Suspends the Simulation

When you enter a `cont` command, the debugger proceeds in the simulation until it reaches the first point in the simulation that meets one of these criteria:

- **At or after specified time** — The simulation time is equal to or greater than the specified time of a timed breakpoint, and the point in the simulation corresponds to an operation that the simulation log is able to show.

If no event executions or relevant updates in signals at reactive ports occur at the specified time of a timed breakpoint, the debugger reaches that breakpoint when the simulation time is strictly later. For example, if time-based blocks in a hybrid simulation have a discrete sample time of 1 and running the simulation without breakpoints causes the simulation log to report operations only at $T = 0, 2, 4, \dots$, then a timed breakpoint at $T = 3$ is equivalent to a timed breakpoint at $T = 4$.

- **At execution or cancelation** — The simulation is about to execute or cancel the specified event of an event breakpoint.
- **At operation of a block** — The block associated with a block breakpoint is about to perform an operation that the simulation log is able to show. For

a list of block operations at which the debugger can suspend the simulation, see “Block Operations Relevant for Block Breakpoints” on page 14-65.

- **At end** — The simulation is about to end. This condition corresponds to the built-in breakpoint at the end of the simulation.

The debugger reaches a given timed or event breakpoint zero or one time during the simulation. The debugger can reach a given block breakpoint an arbitrary number of times during the simulation.

Unless all breakpoints are timed breakpoints, you might not be able to predict which breakpoint the debugger reaches next. Even though events have scheduled times, the debugger might reach an event breakpoint upon the cancelation of an event. You might not be able to predict the cancelation.

Ignoring or Removing Breakpoints

The table describes options for preventing the debugger from observing a particular breakpoint.

Treatment of Breakpoints	At <code>sedebug>></code> Prompt, Enter...	Result
Ignore a particular breakpoint while keeping it in the list of breakpoints and being able to reinstate it easily	<code>disable b1</code> , where <code>b1</code> is the breakpoint identifier	The list of breakpoints indicates the breakpoint as disabled and the debugger does not observe the breakpoint. You can reverse this operation using <code>enable b1</code> .
Ignore a particular breakpoint without keeping it in the list of breakpoints and without being able to reinstate it easily	<code>bdelete b1</code> , where <code>b1</code> is the breakpoint identifier	The breakpoint no longer appears in the list of breakpoints, so the debugger does not observe it.
Ignore all timed and event breakpoints <i>and</i> run the simulation until the end	<code>runtoend</code>	The simulation runs to completion and the debugger session ends.

To view breakpoint identifiers, at the `sedebug>>` prompt, enter `breakpoints`.

Tip You can apply `disable`, `enable`, or `bdelete` to multiple breakpoints in one command by using `all` or a cell array as an input argument. For exact syntax, see the reference page for each function.

Enabling a Disabled Breakpoint

To reinstate a breakpoint that you previously disabled:

- 1 View breakpoint identifiers. At the `sedebug>>` prompt, enter this command:

```
breakpoints
```

- 2 Enter a command like the following, replacing `b1` with the identifier of the breakpoint that you want to reinstate:

```
enable b1
```

You might want to disable and enable a breakpoint to focus on behavior of a block during a particular time interval. A block breakpoint helps you focus on that block. Disabling the block breakpoint, when the simulation time is outside the time interval of interest, helps you focus on only those periods that are relevant to you.

Block Operations Relevant for Block Breakpoints

For each block that supports block breakpoints, the following lists indicate the operations that the block can perform. These operations are the only operations that appear in the debugger simulation log and that can cause the debugger to suspend the simulation at a block breakpoint. The actual operations that occur during a given simulation depend on block configuration and simulation behavior.

Attribute Function

- Entity advancing
- Setting attribute on entity

Attribute Scope

- Destroying entity
- Canceling event
- Entity advancing

Cancel Timeout

- Canceling event
- Entity advancing

Discrete Event Signal to Workspace

- Executing discrete-event signal to workspace

Discrete Event Subsystem

- Scheduling event
- Executing event
- Detected signal update
- Executing subsystem

Enabled Gate

- Scheduling event

- Executing event
- Entity advancing
- Detected signal update

Entity Combiner

- Destroying entity
- Entity advancing
- Combining entities

Entity Departure Counter

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Entity Departure Event to Function-Call Event

- Entity advancing
- Executing function call

Entity Sink

- Destroying entity
- Canceling event
- Entity advancing

Entity Splitter

- Destroying entity
- Entity advancing
- Splitting entity

Entity-Based Function-Call Event Generator

- Scheduling event

- Executing event
- Entity advancing
- Executing function call

Event-Based Entity Generator

- Generating entity
- Scheduling event
- Executing event
- Detected signal update

FIFO Queue

- Entity advancing
- Queuing entity

Get Attribute

- Entity advancing

Infinite Server

- Scheduling event
- Executing event
- Entity advancing

Input Switch

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Instantaneous Entity Counting Scope

- Destroying entity
- Canceling event

- Entity advancing

Instantaneous Event Counting Scope

- Executing scope

LIFO Queue

- Entity advancing
- Queuing entity

N-Server

- Scheduling event
- Executing event
- Entity advancing

Output Switch

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Path Combiner

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Priority Queue

- Entity advancing
- Queuing entity

Read Timer

- Entity advancing

Release Gate

- Scheduling event
- Executing event
- Entity advancing
- Detected signal update

Replicate

- Destroying entity
- Scheduling event
- Executing event
- Entity advancing
- Replicating entity

Schedule Timeout

- Scheduling event
- Executing event
- Entity advancing

Set Attribute

- Entity advancing
- Setting attribute on entity

Signal Latch

- Scheduling event
- Executing event
- Detected signal update
- Executing memory read

- Executing memory write

Signal Scope

- Executing scope

Signal-Based Event to Function-Call Event

- Scheduling event
- Executing event
- Detected signal update
- Executing function call

Signal-Based Function-Call Event Generator

- Scheduling event
- Executing event
- Detected signal update
- Executing function call

Single Server

- Scheduling event
- Executing event
- Canceling event
- Entity advancing
- Preempting entity

Start Timer

- Entity advancing
- Starting timer on entity

Time-Based Entity Generator

- Generating entity
- Scheduling event

- Executing event
- Entity advancing

X-Y Attribute Scope

- Destroying entity
- Canceling event
- Entity advancing

X-Y Signal Scope

- Executing scope

Common Problems in SimEvents Models

In this section...

- “Unexpectedly Simultaneous Events” on page 14-72
- “Unexpectedly Nonsimultaneous Events” on page 14-73
- “Unexpected Processing Sequence for Simultaneous Events” on page 14-73
- “Time-Based Block Not Recognizing Certain Trigger Edges” on page 14-74
- “Incorrect Timing of Signals” on page 14-74
- “Unexpected Use of Old Value of Signal” on page 14-76
- “Effect of Initial Condition on Signal Loops” on page 14-80
- “Loops in Entity Paths Without Sufficient Storage Capacity” on page 14-83
- “Unexpected Timing of Random Signal” on page 14-86
- “Unexpected Correlation of Random Processes” on page 14-88

Unexpectedly Simultaneous Events

An unexpected simultaneity of events can result from roundoff error in event times or other floating-point quantities, and might cause the processing sequence to differ from your expectation about when each event should occur. Computers’ use of floating-point arithmetic involves a finite set of numbers with finite precision. Events scheduled on the event calendar for times T and $T+\Delta t$ are considered simultaneous if $0 \leq \Delta t \leq 128 * \text{eps} * T$, where eps is the floating-point relative accuracy in MATLAB software and T is the simulation time.

If you have a guess about which events’ processing is suspect, adjusting event priorities or using the Instantaneous Event Counting Scope block can help you diagnose the problem. For examples involving event priorities, see “Example: Choices of Values for Event Priorities” on page 3-11 and the Event Priorities demo. For an example using the Instantaneous Event Counting Scope block, see “Example: Counting Events from Multiple Sources” on page 2-35.

Unexpectedly Nonsimultaneous Events

An unexpected lack of simultaneity can result from roundoff error in event times or other floating-point quantities. Computers' use of floating-point arithmetic involves a finite set of numbers with finite precision. Events scheduled on the event calendar for times T and $T+\Delta t$ are considered simultaneous if $0 \leq \Delta t \leq 128 * \text{eps} * T$, where eps is the floating-point relative accuracy in MATLAB software and T is the simulation time.

If roundoff error is very small, the scope blocks might not reveal enough precision to confirm whether events are simultaneous or only close. An alternative technique is to use the Discrete Event Signal to Workspace block to collect data in the MATLAB workspace.

If your model requires that certain events be simultaneous, use modeling techniques aimed at effecting simultaneity. For an example, see “Example: Choices of Values for Event Priorities” on page 3-11.

Unexpected Processing Sequence for Simultaneous Events

An unexpected sequence for simultaneous events could result from the arbitrary or random handling of events having equal priorities, as described in “Processing Sequence for Simultaneous Events” on page 16-2. The sequence might even change when you run the simulation again. When the sequence is arbitrary, do not make any assumptions about the sequence or its repeatability.

If you copy and paste blocks that have an event priority parameter, the parameter values do not change unless you manually change them.

An unexpected processing sequence for simultaneous block operations, including signal updates, could result from interleaving of block operations. For information and examples, see “Interleaving of Block Operations” on page 16-25.

The processing sequence for simultaneous events could have unexpected consequences in the simulation. To learn more about the processing sequence that occurs in your simulation, use the SimEvents debugger. For tips on using

the debugger to examine the processing sequence for simultaneous events, see “Exploring Simultaneous Events” on page 3-4.

To learn which events might be sensitive to priority, try perturbing the model by using different values of blocks’ **Resolve simultaneous signal updates according to event priority** or **Event priority** parameters. Then run the simulation again and see if the behavior changes.

Time-Based Block Not Recognizing Certain Trigger Edges

Time-based blocks have a slightly different definition of a trigger edge compared to event-based blocks. If you use event-based signals with Triggered Subsystem blocks or Stateflow blocks with trigger inputs, the blocks might not run when you expect them to. For more information and an example, see “Zero-Duration Values and Time-Based Blocks” on page 16-31.

Incorrect Timing of Signals

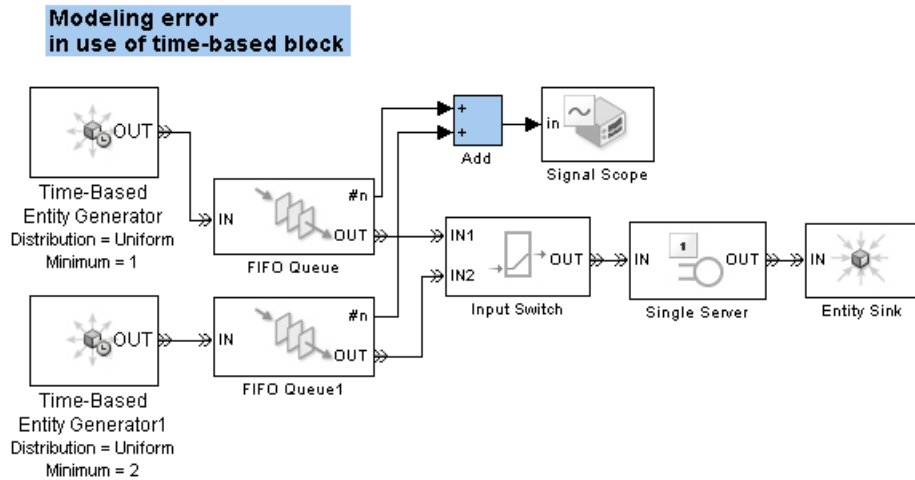
If you use a time-based block to process event-based signals, the output signal might be a time-based signal. Depending on your model, you might notice that:

- The output signal assumes a new value at a later time than the event that caused the last update of the event-based signal.
- The output signal assumes incorrect values.
- An event-based block that uses the output signal, such as an Event-Based Entity Generator block, operates with incorrect timing.

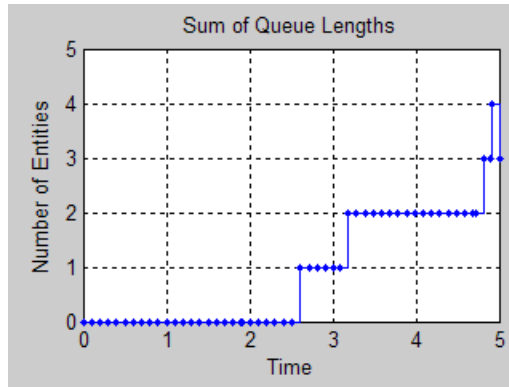
You can avoid these problems by putting the time-based block in a discrete event subsystem, as described in Chapter 10, “Controlling Timing with Subsystems”. If your time-based block is in a Function-Call Subsystem, be sure to select **Propagate execution context across subsystem boundary** as described in “Setting Up Function-Call Subsystems in SimEvents Models” on page 10-34.

Example: Time-Based Addition of Event-Based Signals

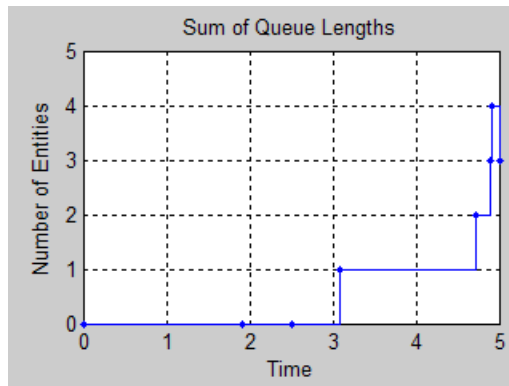
The following model adds the lengths of two queues. The queue lengths are event-based signals, while the Add block is a time-based block. It is important that the Add block use up-to-date values of its input signals each time the queue length changes and that the output signal's updates correspond to updates in one of the queue length signals.



If you build this model without having used `simeventsstartup` previously, or without using `simeventsconfig` later, you might see the following plot. The incorrect timing is evident because the sum signal has updates at regular intervals that are smaller than the minimum intergeneration time of the entity generators.



If you correct the simulation parameters by using `simeventsconfig` on this model (with either the 'des' or 'hybrid' input argument), the plot reveals correct update times but incorrect values. To verify the values, you can connect the inputs and outputs of the Add block to separate Discrete Event Signal to Workspace blocks and examine the data in the MATLAB workspace.



A better model uses the discrete event subsystem illustrated in the Time-Driven and Event-Driven Addition demo.

Unexpected Use of Old Value of Signal

During a discrete-event simulation, multiple events or signal updates can occur at a fixed value of the simulation clock. If these events and signal

updates are not processed in the sequence that you expect, you might notice that a computation or other operation uses a signal value from a previous time instead of from the current time. Some common situations occur when:

- A block defers the update of an output signal until a departing entity has either finished advancing to a subsequent storage block or been destroyed, but an intermediate nonstorage block in the sequence uses that signal in a computation or to control an operation. Such deferral of updates applies to most SimEvents blocks that have both an entity output port and a signal output port.

For examples, see “Example: Using a Signal or an Attribute” on page 14-78 and the Managing Race Conditions demo.

For details, see “Interleaving of Block Operations” on page 16-25.

For a technique you can use when the situation involves the Output Switch block’s **p** input signal, see “Using the Storage Option to Prevent Latency Problems” on page 6-2.

- An entity-departure subsystem updates its output signal after a block uses the value. A typical reason is that the block is responding to the arrival of an entity departing from the same block that generates the function call that calls the subsystem.

For an example showing how a Single Server block helps avoid this scenario, see “Example: Using Entity-Based Timing for Choosing a Port” on page 10-30.

- A computation involving multiple signals is performed before all of the signals have been updated.

For details and an example, see “Update Sequence for Output Signals” on page 16-34.

- A time-based block’s use of a value of an event-based signal persists until the next time step of the time-based simulation clock, even if the block producing the event-based signal has already updated the value. In many cases, this is the correct behavior of the time-based block.

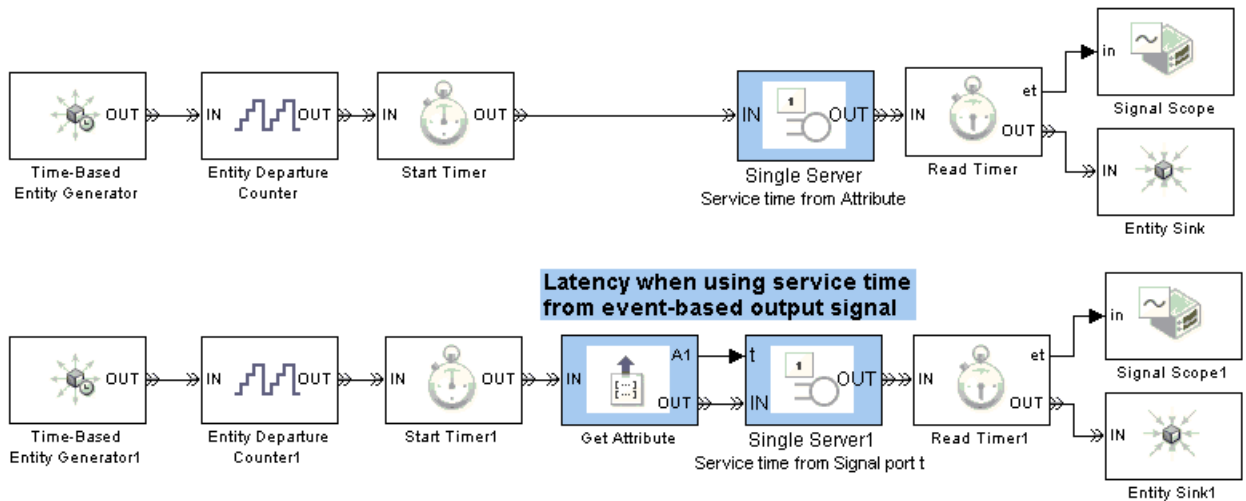
For an example, see “Example: Plotting Entity Departures to Verify Timing” on page 11-11.

If you need a time-based block to respond to events, consider using a discrete event subsystem, as described in Chapter 10, “Controlling Timing with Subsystems”.

If you want notification of some of these situations, use the configuration parameters related to race conditions. For details, see “SimEvents Diagnostics Pane”.

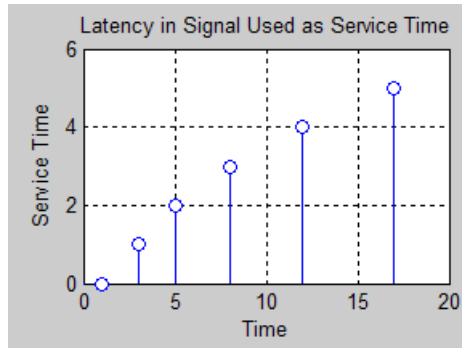
Example: Using a Signal or an Attribute

The goal in the next model is to use a service time of N seconds for the Nth entity. The Entity Counter block stores each entity’s index, N, in an attribute. The top portion of the model uses the attribute directly to specify the service time. The bottom portion creates a signal representing the attribute value and attempts to use the signal to specify the service time. These might appear to be equivalent approaches, but only the top approach satisfies the goal.

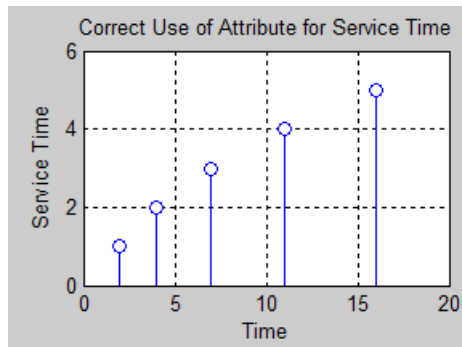


The plot of the time in the bottom server block and a warning message in the Command Window both reveal a modeling error in the bottom portion of the model. The first entity’s service time is 0, not 1, while the second entity’s service time is 1, not 2. The discrepancy between entity index and service time occurs because the Get Attribute block processes the departure of the

entity before the update of the signal at the **A1** signal output port. That is, the server computes the service time for the newly arrived entity before the **A1** signal reflects the index of that same entity. For more information about this phenomenon, see “Interleaving of Block Operations” on page 16-25.



The top portion of the model, where the server directly uses the attribute of each arriving entity, uses the expected service times. The sequential processing of an entity departure and a signal update does not occur because each entity carries its attributes with it.



Tip If your entity possesses an attribute containing a desired service time, switching criterion, timeout interval, or other quantity that a block can obtain from either an attribute or signal, use the attribute directly rather than creating a signal with the attribute's value and having to ensure that the signal is up-to-date when the entity arrives.

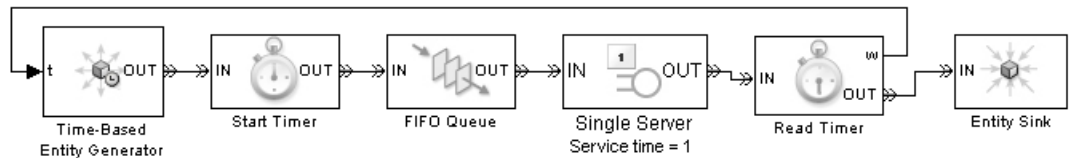
Effect of Initial Condition on Signal Loops

When you create a loop in a signal connection, consider the effect of initial conditions. If you need to specify initial conditions for event-based signals, see “Specifying Initial Values of Event-Based Signals” on page 4-12.

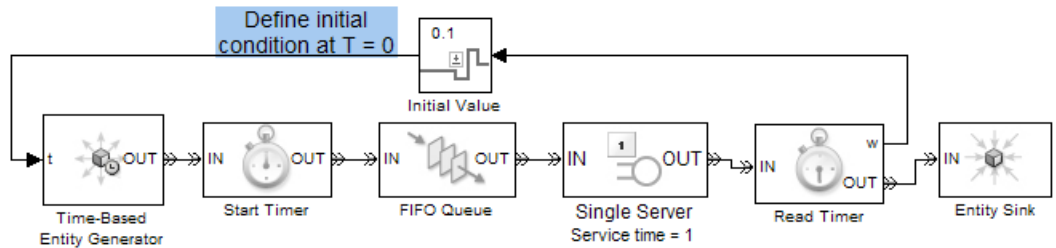
Example: Intergeneration Time of Zero at Simulation Start

The following model is problematic at $T=0$ because the initial reading of the t input signal representing the intergeneration time is 0. This signal does not assume a positive value until the first entity departs from the Read Timer block, which occurs after the first completion of service at $T=1$.

Modeling error at $T = 0$

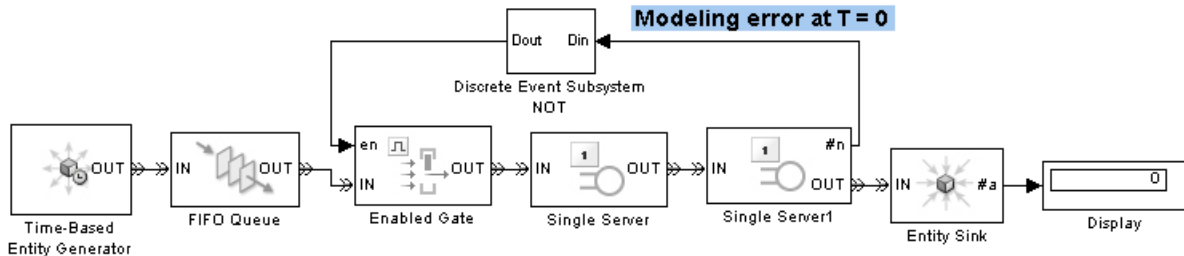


A better model specifies a nonzero initial condition for the w output signal from the Read Timer block.

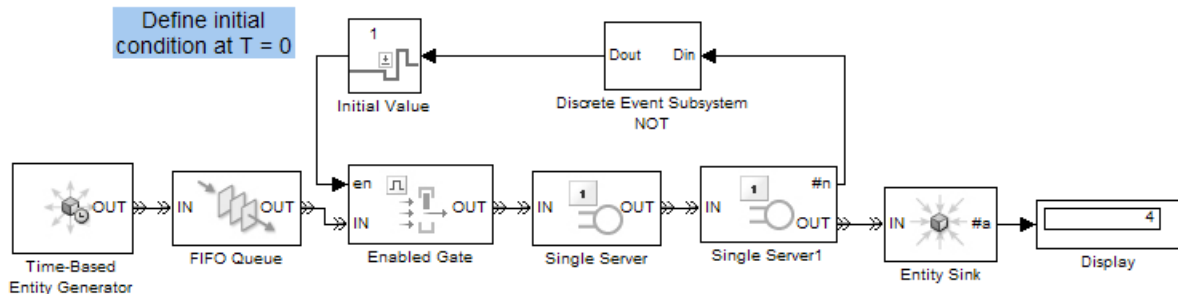


Example: Absence of Sample Time Hit at Simulation Start

In the following model, the second server's #n signal has no updates before the first entity arrival there. As a result, the discrete event subsystem, whose role is to perform a NOT operation on the #n signal, does not execute before the first entity arrival at the server. However, no entity can arrive at the server until the gate opens. This logic causes entities to accumulate in the queue instead of advancing past the gate and to the servers.



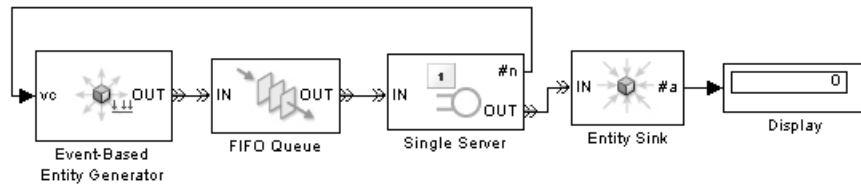
A better model defines a positive initial condition for the **en** input signal to the gate.



Example: Faulty Logic in Feedback Loop

The following model generates no entities because the logic is circular. The entity generator is waiting for a change in its input signal, but the server's output signal never changes until an entity arrives or departs at the server.

Modeling error at $T = 0$



To use the SimEvents debugger to see that the example has a modeling error:

- 1 Begin a debugger session for the model. At the MATLAB command prompt, enter:

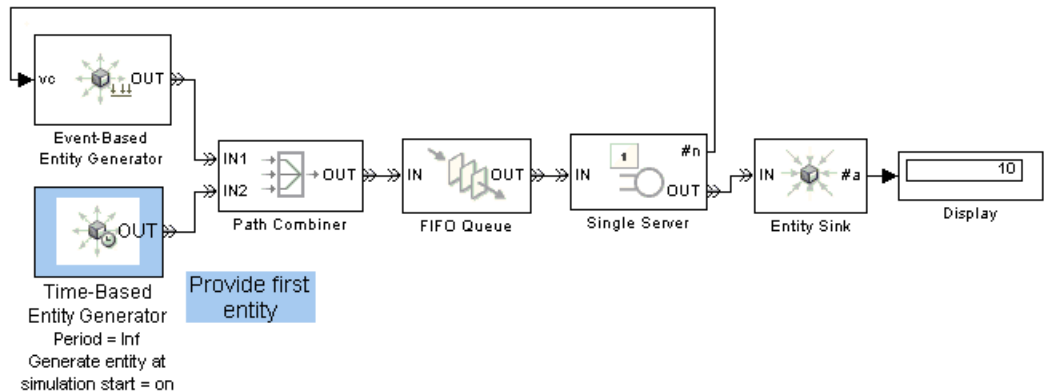
```
simeventsdocex('doc_ic_noentities')
sedebug('doc_ic_noentities')
```

- 2 Run the entire simulation. At the `sedebug>>` prompt, enter:

```
runtoend
```

If the simulation generated entities, the debugger would display messages in the Command Window to indicate that. The lack of output in this case shows that the simulation generates no entities.

A better model provides the first entity in a separate path. In the revised model, the Time-Based Entity Generator block generates exactly one entity during the simulation, at $T=0$.



You can use the debugger again to confirm that the revised model generates entities. If you use the preceding procedure, but substitute 'doc_ic_no_entities_fix' in place of 'doc_ic_no_entities', you can see that the debugger reports entity generations and other operations during the simulation of the revised model.

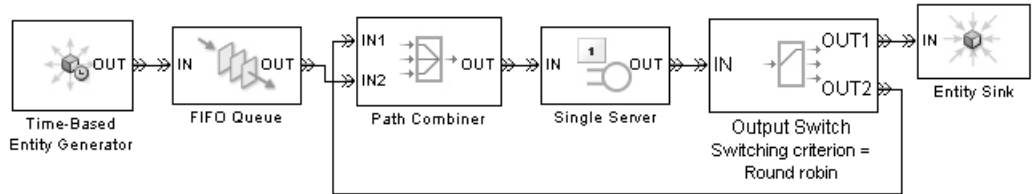
Loops in Entity Paths Without Sufficient Storage Capacity

An entity path that forms a loop should contain storage that will not become exhausted during the simulation. Storage blocks include queues and servers; for a list of storage blocks, see “Storage and Nonstorage Blocks” on page 16-37. The following example illustrates how the storage block can prevent a deadlock.

Example: Deadlock Resulting from Loop in Entity Path

The following model contains a loop in the entity path from the Output Switch block to the Path Combiner block. The problem occurs when the switch selects the entity output port **OUT2**. The entity attempting to depart from the server looks for a subsequent storage block where it can reside. It cannot reside in a routing block. Until the entity confirms that it can advance to a storage block or an entity-destroying block, the entity cannot depart. However, until it departs, the server is not available to accept a new arrival. The result is a deadlock.

Modeling error when switch uses OUT2



To use the SimEvents debugger to identify the deadlock:

- 1 Begin a debugger session for the model. At the MATLAB command prompt, enter:

```
simeventsdocex('doc_loop')
sedebug('doc_loop')
```

- 2 Run the simulation until the built-in breakpoint at the end of the simulation. At the `sedebug>>` prompt, enter:

```
cont
```

The debugger displays log messages in the Command Window so you can see what happens in the simulation. The latest time stamp in the messages is at $T = 3$:

```
%=====
Executing EntityGeneration Event (ev9)           Time = 3.0000000000000000
: Entity = <none>                               Priority = 300
: Block = Time-Based Entity Generator
%.....%
Generating Entity (en4)
: Block = Time-Based Entity Generator

Hit built-in breakpoint for the end of simulation.
```

The lack of log messages after $T = 3$ reflects the deadlock.

- 3 If you inspect the final state of the switch, you see that it selects the entity output port **OUT2**:


```
blkinfo('doc_loop/Output Switch')
```

The output is:

```
Output Switch Current State          T = 10.000000000000000
Block (blk4): Output Switch

Advancing Entity    = <none>
Selected Output Port = 2
```

- 4** If you inspect the final state of the server, you see that it is holding an entity that completed its service at $T=2$:

```
blkinfo('doc_loop/Single Server')
```

The output is:

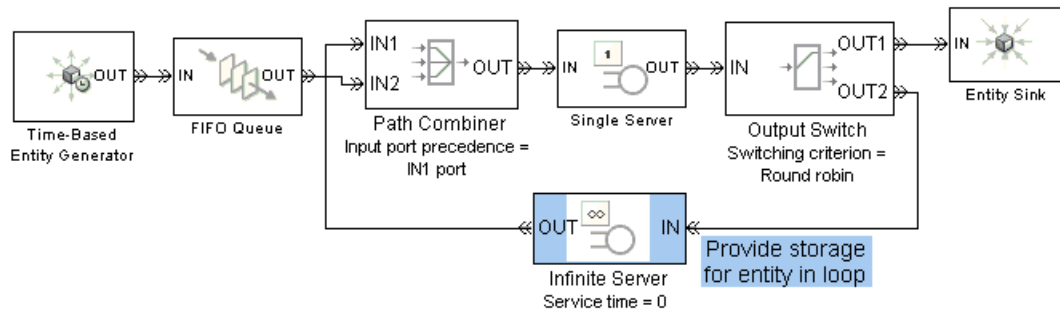
```
Single Server Current State          T = 10.000000000000000
Block (blk5): Single Server

Entities (Capacity = 1):
Pos    ID      Status           Event    EventTime
1      en2     Service Completed ev8      2
```

- 5** End the debugger session. At the `sedebg>>` prompt, enter:

```
sedb.quit
```

A better model includes a server with a service time of 0 in the looped entity path. This storage block provides a place for an entity to reside after it departs from the Output Switch block. After the service completion event is processed, the entity advances to the Path Combiner block and back to the Single Server block. The looped entity path connects to the Path Combiner block's **IN1** entity input port, not **IN2**. This ensures that entities on the looped path, not new entities from the queue, arrive back at the Single Server block.

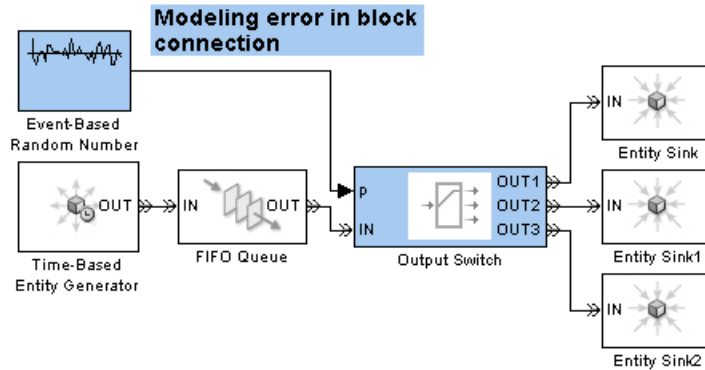


Unexpected Timing of Random Signal

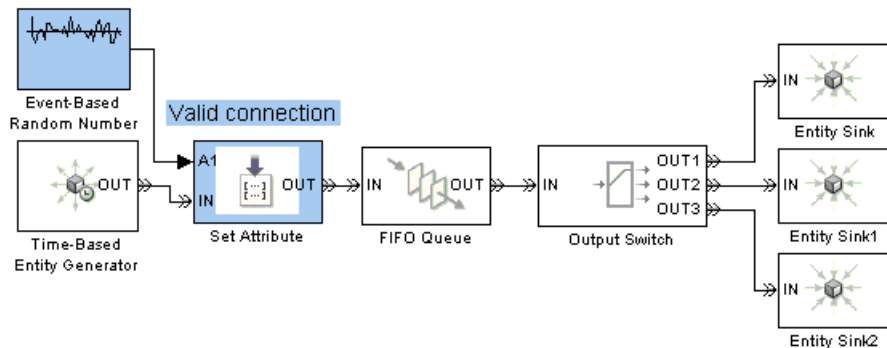
When you use the Event-Based Random Number block to produce a random event-based signal, the block infers from a subsequent block the events upon which to generate a new random number from the distribution. The sequence of times at which the block generates a new random number depends on the port to which the block is connected and on events occurring in the simulation. To learn how to use this block, see “Generating Random Signals” on page 4-4.

Example: Invalid Connection of Event-Based Random Number Generator

The following model is incorrect because the Event-Based Random Number block cannot infer from the **p** input port of an Output Switch block when to generate a new random number. The Output Switch block is designed to listen for changes in its **p** input signal and respond when a change occurs. The Output Switch cannot cause changes in the input signal value or tell the random number generator when to generate a new random number. The **p** input port of the Output Switch block is called a reactive port and it is not valid to connect a reactive signal input port to the Event-Based Random Number block.



If you want to generate a new random number corresponding to each entity that arrives at the switch, a better model connects the Event-Based Random Number block to a Set Attribute block and sets the Output Switch block's **Switching criterion** parameter to **From attribute**. The random number generator then generates a new random number upon each entity arrival at the Set Attribute block. The connection of the Event-Based Random Number block to the **A1** input port of the Set Attribute block is a supported connection because the **A2** port is a notifying port. To learn more about reactive ports and notifying ports, see the reference page for the Event-Based Random Number block.



Unexpected Correlation of Random Processes

An unexpected correlation between random processes can result from nonunique initial seeds in different dialog boxes. If you copy and paste blocks that have an **Initial seed** or **Seed** parameter, the parameter values do not change unless you manually change them. Such blocks include:

- Time-Based Entity Generator
- Event-Based Random Number
- Entity Splitter
- Blocks in the Routing library
- Uniform Random Number
- Random Number
- Masked subsystems that include any of the preceding blocks

Detecting Nonunique Seeds and Making Them Unique

To detect and correct some nonunique initial seeds, use a diagnostic setting:

- 1** Open the Configuration Parameters dialog box for the model using **Simulation > Configuration Parameters**.
- 2** Navigate to the **SimEvents Diagnostics** pane of the dialog box.
- 3** Set **Identical seeds for random number generators** to warning.

When you run the simulation, the application checks for nonunique **Initial seed** parameter values in SimEvents library blocks. Upon detecting any, the application issues a warning message that includes instructions for solving the problem.

Exceptions You must monitor seed values manually if your model contains random-number-generating blocks that come from libraries other than the SimEvents libraries.

To learn how to query and change seed values of such blocks, use the techniques illustrated in “Working with Seeds Not in SimEvents Blocks” on page 12-31.

Recognizing Latency in Signal Updates

In some cases, the updating of an output signal or the reaction of a block to updates in its input signal can experience a delay:

- When you use an event-based signal as an input to a time-based block that is not in a discrete event subsystem, the block might not react to changes in the input at exact event times but instead might react at the next time-based sample time hit for that block.

To make time-based blocks react to changes immediately when an event occurs in another block, use a discrete event subsystem. For details and examples, see Chapter 10, “Controlling Timing with Subsystems”.

- The update of an output signal in one block might occur after other operations occur at that value of time, in the same block or in other blocks. This latency does not last a positive length of time, but might affect your simulation results. For details and an example, see “Interleaving of Block Operations” on page 16-25.
- The reaction of a block to an update in its input signal might occur after other operations occur at that value of time, in the same block or in other blocks. This latency does not last a positive length of time, but might affect your simulation results. For details, see “Choosing How to Resolve Simultaneous Signal Updates” on page 16-7.
- When the definition of a statistical signal suggests that its value can vary *continuously* as simulation time elapses, the block increases efficiency by updating the signal value only at key moments during the simulation. As a result, the signal has a somewhat outdated “approximate” value between such key moments, but corrects the value later.

The primary examples of this phenomenon are the signals that represent time averages, such as a server’s utilization percentage. The definitions of time averages involve the current time, but simulation performance would suffer drastically if the block recomputed the percentage at each time-based simulation step. Instead, the block recomputes the percentage only under these circumstances:

- Upon the arrival or departure of an entity
- When the simulation ends

- When you pause the simulation using **Simulation > Pause** or other means

For an example, see the reference page for the Single Server block.

When plotting statistics that, by definition, vary continuously as simulation time elapses, consider using a continuous-style plot. For example, set **Plot type** to **Continuous** in the Signal Scope block.

Running Discrete-Event Simulations Programmatically

- “Accelerating Discrete-Event Simulations Using Rapid Simulation” on page 15-3
- “Sharing Executables for Discrete-Event Simulations” on page 15-5
- “Prerequisites for Using Generated Code” on page 15-9
- “Choice of Environment for Varying Parameters Between Simulation Runs” on page 15-10
- “Varying Parameters Between Simulation Runs Using MATLAB Code” on page 15-13
- “Varying Parameters Between Rapid Simulation Runs” on page 15-15
- “Designing Models to Accept Event-Based Data During Rapid Simulation” on page 15-18
- “Example: Computing an Ensemble Average Using MATLAB Code” on page 15-26
- “Example: Varying the Number of Servers Using MATLAB Code” on page 15-29
- “Example: Computing an Ensemble Average Using Rapid Simulation” on page 15-32
- “Example: Varying Attribute Values Between Runs Using Rapid Simulation” on page 15-37

- “Example: Varying Queue Capacity Between Runs Using Rapid Simulation” on page 15-43
- “Limitations of Rapid Simulation for Discrete-Event Simulations” on page 15-48

Accelerating Discrete-Event Simulations Using Rapid Simulation

In this section...

“Objective” on page 15-3

“Prerequisites” on page 15-3

“Procedure” on page 15-3

“See Also” on page 15-4

Objective

For simulating your SimEvents models more quickly, run an executable that Real-Time Workshop® software builds using the Rapid Simulation target. A comprehensive workflow for using rapid simulation is in “General Rapid Simulation Workflow” in the Real-Time Workshop documentation. Following is a simpler procedure that highlights the slight differences for SimEvents models and focuses on acceleration instead of other rapid simulation benefits.

Prerequisites

- Real-Time Workshop software license
- Write access to the current folder in the MATLAB session

Procedure

- 1 Design your model so that it records the signals that represent relevant simulation results.

Simulation Result	Design Step
Event-based signal	Connect a Discrete Event Signal to Workspace block to the signal line.
Time-based signal	Connect a To Workspace, To File, or Outputport block to the signal line.

- 2** Configure and build the model for rapid simulation. Follow the procedure in “Configuring and Building a Model for Rapid Simulation” in the Real-Time Workshop documentation. However, do not instruct the target to use a fixed-step solver and do not set up external mode communications.
- 3** Run the simulation by running the executable. It automatically creates a MAT-file containing simulation results. For details on running the executable, see “Running Rapid Simulations” in the Real-Time Workshop documentation.
- 4** Analyze simulation results. To load results from the MAT-file into the MATLAB workspace, use `load`, the Current Folder browser, or the Import Wizard.

See Also

- “Sharing Executables for Discrete-Event Simulations” on page 15-5
- “Varying Parameters Between Rapid Simulation Runs” on page 15-15

Sharing Executables for Discrete-Event Simulations

In this section...

“Objective” on page 15-5

“Model Designer and Model User Roles” on page 15-5

“Procedure for Model Designer” on page 15-5

“Procedure for Model User” on page 15-7

Objective

A model designer can share capabilities of a SimEvents model with someone who wants to simulate the model without modifying its structure or viewing it graphically.

Model Designer and Model User Roles

Role	Description
Model designer	A person who works in the Simulink environment to develop, modify, view, simulate, or build (that is, generate code for) a model.
Model user	A person who simulates a model and analyzes results.

Procedure for Model Designer

Prerequisites for Model Designer and Model User

- As the model designer, you and the model user must work on the same platform.
- You and the model user must have the same versions of Simulink and SimEvents software.
- Know which simulation results the model user wants to save or analyze.

- Know which quantities the model user wants to provide or vary when running the simulation. Also, determine whether you or the model user plans to set up the corresponding input data.

Software Prerequisites

- Real-Time Workshop software license
- Write access to the folder that is current in the MATLAB environment

Procedure

- 1 Design your model to record the signals representing simulation results that the model user wants to save or analyze.

Simulation Result	Design Step
Event-based signal	Connect a Discrete Event Signal to Workspace block to the signal line.
Time-based signal	Connect a To Workspace, To File, or Outport block to the signal line.

- 2 Design your model to accept the input data that the model user wants to provide or vary when running the simulation. For details, see the sections in the table.

Data	Related Topics
Values of tunable parameters, including event-based signals from data that you provide	“Designing Models to Accept Event-Based Data During Rapid Simulation” on page 15-18
Time-based input signal	“Configuring Inport Blocks to Provide Rapid Simulation Source Data” in the Real-Time Workshop documentation

- 3 Configure and build the model for rapid simulation. Follow the procedure in “Configuring and Building a Model for Rapid Simulation” in the

Real-Time Workshop documentation. However, do not instruct the target to use a fixed-step solver and do not set up external mode communications.

- 4** Unless the model user is setting up input data, set up input data by creating one or more MAT-files. The input data can be values of tunable parameters, including event-based signals from data that you provide, or time-based input signals. The files contain input data that the model user provides when running the simulation. For details, see “Setting Up Rapid Simulation Input Data” in the Real-Time Workshop documentation.
- 5** Provide these files to the model user:
 - Generated executable
 - Input MAT-files, if you created them

Procedure for Model User

Prerequisites: See “Prerequisites for Using Generated Code” on page 15-9.

- 1** Unless the model designer set up input data, do so by creating one or more MAT-files. The files contain input data to provide to the executable when you run the simulation.

Data	Related Topics
Values of tunable parameters, including event-based signals from data that you provide	“Creating a MAT-File That Includes a Model Parameter Structure” in the Real-Time Workshop documentation
Time-based input signal	“Creating a MAT-File for an Inport Block” in the Real-Time Workshop documentation

- 2** From a folder where you have write access, run the simulation by running the executable that you received from the model designer. Use these command-line options, as applicable.

Command-Line Option	Purpose
<code>-p param_filename.mat</code>	Read a parameter set from the input MAT-file, <i>param_filename.mat</i>
<code>-i input_filename.mat</code>	Read a time-based input signal from the MAT-file, <i>input_filename.mat</i>
<code>-o output_filename.mat</code>	Write simulation results to the output MAT-file, <i>output_filename.mat</i>

Running the executable automatically creates a MAT-file containing simulation results.

For details on running the executable, including additional command-line options, see “Running Rapid Simulations” in the Real-Time Workshop documentation.

For details on scripting for running the simulation, see “Programming Scripts for Batch and Monte Carlo Simulations” in the Real-Time Workshop documentation.

- 3** Analyze simulation results. To load results from the output MAT-file into the MATLAB workspace, use `load`, the Current Folder browser, or the Import Wizard.

Prerequisites for Using Generated Code

The prerequisites for using the generated code from a SimEvents model are:

- Use the same platform as the platform that compiled the code.
- Ensure that you have these products installed:
 - MATLAB
 - Simulink
 - SimEvents

The installation must be from the same release as the release that generates the code.

- Set environment variables, as described in “Requirements for Running Rapid Simulations”, in the Real-Time Workshop documentation.

Note If you have not installed the Real-Time Workshop product, you can find its documentation on the MathWorks™ Web site:

<http://www.mathworks.com/access/helpdesk/help/helpdesk.html>

When you use the generated code from a SimEvents model, you do not check out a Simulink or SimEvents license.

Choice of Environment for Varying Parameters Between Simulation Runs

In this section...

“Comparison of Approaches” on page 15-10

“Adapting Existing Models and MATLAB Code to Work with Rapid Simulation” on page 15-11

Comparison of Approaches

When you run simulations repeatedly and vary aspects of the model from one simulation run to the next, you can:

- Refine your design by improving parameter values or estimating optimal values
- Check the robustness of your design or answer “what if” questions
- Simulate with different streams of random numbers or perform Monte Carlo analysis

The table compares approaches for running a simulation repeatedly in a noninteractive (batch) process.

Approach	Advantages	Details
Use <code>sim</code> repeatedly to simulate the model in the MATLAB environment.	<ul style="list-style-type: none"> • Simpler procedure • Greater flexibility to change the model between simulation runs • No need for Real-Time Workshop software license 	“Varying Parameters Between Simulation Runs Using MATLAB Code” on page 15-13
Build a rapid simulation executable from the model. Then	<ul style="list-style-type: none"> • Greater simulation speed 	“Varying Parameters Between Rapid

Approach	Advantages	Details
run the executable repeatedly.	<ul style="list-style-type: none"> • Ability to share the executable with users who do not need to modify your model • Ability to run the executable in the MATLAB environment or outside it 	Simulation Runs” on page 15-15

To organize your simulations and their results more easily in a graphical way, you can use SystemTest™ software. It is compatible with either of the approaches for running a simulation. For an example that uses SystemTest software with a SimEvents model, see Distributed Processing Resource Modeling.

Adapting Existing Models and MATLAB Code to Work with Rapid Simulation

Suppose you are migrating from the `sim` approach to the rapid simulation approach. Both approaches can involve MATLAB code that performs setup, simulation, and analysis tasks. When adapting your model and code from the `sim` approach to the rapid simulation approach:

- Design your model so that the quantity that you want to vary between simulation runs corresponds to a tunable parameter. For details, see Designing Models to Accept Event-Based Data During Rapid Simulation.
- Where you invoke `sim` to simulate the model, instead invoke `system` to run the executable.
- Do not use `set_param` or `se_randomize_seeds` to set numerical parameter values directly in a block. Instead, change your model and code as follows:

Changes in Your Model

- 1 In the block dialog box containing the tunable parameter that you want to vary, enter the name of a workspace variable.

- 2 Select the **Inline parameters** configuration parameter and identify the workspace variable as a tunable parameter. For details, see the step involving **Inline parameters** in “Example: Varying Attribute Values Between Runs Using Rapid Simulation” on page 15-37.

Changes in Your MATLAB Code

- 1 After building the model, set up input data using a MAT-file.
 - 2 When running the executable, use the `-p` command-line option to refer to the MAT-file containing the input data.
- Use `load` to load results from the output MAT-file into the MATLAB workspace. Running a rapid simulation writes results to an output MAT-file and does not put them directly in the MATLAB workspace.

Varying Parameters Between Simulation Runs Using MATLAB Code

In this section...

“Objective” on page 15-13

“Procedure” on page 15-13

“See Also” on page 15-14

Objective

For parameter sweeps or Monte Carlo analysis, you can vary or tune parameters from one simulation run to the next. Following is a procedure that you can use in the MATLAB environment to vary parameters between simulation runs. Alternatively, to vary parameters between runs of a rapid simulation executable, see “Varying Parameters Between Rapid Simulation Runs” on page 15-15.

Procedure

Use a MATLAB program to run simulations repeatedly. A typical procedure is:

- 1 Design your model so that in the MATLAB workspace, it records the signals that represent relevant simulation results. These signals are the results that you want to save or analyze after running the simulation.

Simulation Result	Design Step
Event-based signal	Connect a Discrete Event Signal to Workspace block to the signal line.
Time-based signal	Connect a To Workspace block to the signal line.

- 2 (*Optional*): Design the model to accept values of the parameters that you want to provide or vary when you run the simulation. In the block dialog

box, set the value to the name of a variable. In step 4 of this procedure, your code must define the variable before each simulation run.

Alternative If you want to vary values but do not perform this step, you can use `se_randomizeseeds` or `set_param` commands in step 4 of this procedure.

- 3 *(Optional)*: Remove plotting and Display blocks to increase simulation speed. When running simulations unattended, you are not viewing plots or displays.
- 4 Execute MATLAB code that runs the simulation and analyzes the results. Typical constructs include:
 - `for` loop, to repeat simulation runs.
 - `se_randomizeseeds` and `se_getseeds` commands, to change random number sequences between simulation runs and store the seeds for future reference. For details, see “Varying Simulation Results by Managing Seeds” on page 12-28.
 - `set_param` commands, to vary parameters or other aspects of the model between simulation runs.
 - `sim` command to run the simulation. To learn more about `sim`, see “Running a Simulation Programmatically” in the Simulink documentation.
 - `save` or array-building commands that store results from each simulation run, without overwriting results of earlier simulation runs.

See Also

- “Example: Computing an Ensemble Average Using MATLAB Code” on page 15-26
- “Example: Varying the Number of Servers Using MATLAB Code” on page 15-29

Varying Parameters Between Rapid Simulation Runs

In this section...
“Objective” on page 15-15
“Prerequisites” on page 15-15
“Procedure” on page 15-15
“For Further Information” on page 15-17

Objective

For parameter sweeps or Monte Carlo analysis, you can vary or tune certain parameters from one simulation run to the next. Following is a procedure to vary parameters between simulation runs of a rapid simulation executable that you build. Parameters that support this procedure are *tunable parameters*.

Alternatives To vary tunable or nontunable parameters between simulation runs in the MATLAB environment, see “Varying Parameters Between Simulation Runs Using MATLAB Code” on page 15-13.

To use a rapid simulation executable that someone else built, see “Sharing Executables for Discrete-Event Simulations” on page 15-5.

Prerequisites

- Real-Time Workshop software license
- Write access to the current folder in the MATLAB session

Procedure

- 1 Design your model to record the signals that represent relevant simulation results.

Simulation Result	Design Step
Event-based signal	Connect a Discrete Event Signal to Workspace block to the signal line.
Time-based signal	Connect a To Workspace, To File, or Outputport block to the signal line.

- 2 Design your model to accept the input data that you want to provide or vary when you run the simulation. For details, see the sections in the table.

Data	Related Topics
Values of tunable parameters, including event-based signals from data that you provide	“Designing Models to Accept Event-Based Data During Rapid Simulation” on page 15-18
Time-based input signal	“Configuring Inport Blocks to Provide Rapid Simulation Source Data” in the Real-Time Workshop documentation

- 3 Configure and build the model for rapid simulation. Follow the procedure in “Configuring and Building a Model for Rapid Simulation” in the Real-Time Workshop documentation. However, do not instruct the target to use a fixed-step solver and do not set up external mode communications.
- 4 Set up input data by creating one or more MAT-files. The input data can be values of tunable parameters, including event-based signals from data that you provide, or time-based input signals. The files contain input data that you provide when running the simulation. For details, see “Setting Up Rapid Simulation Input Data” in the Real-Time Workshop documentation.
- 5 Run the simulation by running the executable. Use these command-line options.

Command-Line Option	Purpose
<code>-p param_filename.mat</code>	Read a parameter set from the input MAT-file, <i>param_filename.mat</i>
<code>-i input_filename.mat</code>	Read a time-based input signal from the MAT-file, <i>input_filename.mat</i> . Use this option if you configured Inport blocks to provide rapid simulation source data.
<code>-o output_filename.mat</code>	Write simulation results to the output MAT-file, <i>output_filename.mat</i>

Running the executable automatically creates a MAT-file containing simulation results.

For details on running the executable, including additional command-line options, see “Running Rapid Simulations” in the Real-Time Workshop documentation.

For details on scripting for running the simulation, see “Programming Scripts for Batch and Monte Carlo Simulations” in the Real-Time Workshop documentation.

- 6 Analyze simulation results. To load results from the output MAT-file into the MATLAB workspace, use `load`, the Current Folder browser, or the Import Wizard.

For Further Information

- Batch Discrete-Event Simulations Using Rapid Simulation Target demo
- Using RSim Target for Parameter Survey Real-Time Workshop demo
- “Example: Computing an Ensemble Average Using Rapid Simulation” on page 15-32

Designing Models to Accept Event-Based Data During Rapid Simulation

In this section...
“Event-Based Data for Rapid Simulations” on page 15-18
“Varying the Stream of Random Numbers” on page 15-21
“Varying Intergeneration Times” on page 15-21
“Varying Attribute Values” on page 15-22
“Varying Service Times” on page 15-23
“Varying Timeout Intervals” on page 15-24
“Varying Event-Based Signals” on page 15-25

Event-Based Data for Rapid Simulations

When you run a rapid simulation, you can provide data for certain event-based quantities. Similarly, you can vary those quantities between simulation runs, without rebuilding the model, if you do not change:

- The array size of the values that you are varying
- The complexity of the values that you are varying

To enable providing or varying a quantity in this way, design your model so that the quantity corresponds to a tunable parameter:

- If the quantity is a parameter listed in the following table, it is a tunable parameter. Set it to the name of a workspace variable. You can change the value of this variable in each simulation run. The size and complexity of the value must remain the same for all simulation runs.

Tunable Parameters in SimEvents Blocks

Block	Tunable Parameters
Event-Based Random Number	All block parameters, except Distribution
Event-Based Sequence	Vector of output values

As a special case, see “Varying the Stream of Random Numbers” on page 15-21.

Other tunable parameters are in blocks in the Simulink library, and custom blocks in your own libraries or models.

- If the block can read the quantity from a notifying port on a SimEvents block, you can vary the quantity after modifying your model slightly. Procedures for cases that fall into this category are in:
 - “Varying Intergeneration Times” on page 15-21
 - “Varying Attribute Values” on page 15-22
 - “Varying Service Times” on page 15-23
 - “Varying Timeout Intervals” on page 15-24
 - “Varying Event-Based Signals” on page 15-25
- Otherwise, consider redesigning a portion of your model to relate the quantity to a tunable parameter. For ideas, see these examples.

Quantity	Example	Tips
Capacity of a queue	Variable-Capacity Queue	Connect the cap port of the block labeled Variable-Capacity Queue to a Constant block. In the Constant block, the Constant value parameter is tunable. As a result, the queue capacity is constant in each simulation run, and varies between simulation runs.
Number of servers in an N-server	Batch Discrete-Event Simulations Using	The Allocate Employees to Customers portion of the model reinterprets an N-server problem as a resource allocation problem. The block labeled Employee Pool is a custom block whose parameter is tunable. To look inside this block, select it, select Edit > Look Under

Quantity	Example	Tips
	Rapid Simulation Target	Mask , and double-click the block labeled Preload Pool. In the custom block the parameter value becomes the value in a Constant block that specifies the number of resources (analogous to the number of servers).
Number of copies to make when replicating an entity	Variable Entity Replication	Connect the A1 port of the Set Attribute block to a Constant block. In the Constant block, the Constant value parameter is tunable. As a result, the number of copies is constant in each simulation run, and varies between simulation runs.
Route in an Output Switch block	“Generating Sequences Based on Arbitrary Events” on page 4-10	The switch routes entities based on a signal value that changes with each arriving entity. In the Event-Based Sequence block, the Vector of output values parameter is tunable. For example, you can do one simulation run using round-robin switching and another simulation run using a random vector of values.
Enabling or disabling certain data processing	Bit Timing Recovery Using Fixed-Rate Resampling and SimEvents	The Timing Control Unit subsystem uses a Switch block labeled Enable Timing Control to determine whether the processing incorporates timing control. A Constant block provides the signal for the switching criterion. In the Constant block, the Constant value parameter is tunable. As a result, you can do one simulation run using timing control and another simulation run without timing control.

Note If you substitute a time-based block for a SimEvents block, the behavior of your model is likely to change. Substituting blocks to take advantage of tunable parameters in the time-based block might cause problems. For more information on problems from inappropriate use of time-based blocks in a discrete-event simulation, see “Incorrect Timing of Signals” on page 14-74.

After the quantity corresponds to a tunable parameter, vary the value of the quantity between simulation runs by varying the parameter value. For details, see “Creating a MAT-File That Includes a Model Parameter Structure” and the description of the `-p` command-line option in the Real-Time Workshop documentation.

If the quantity cannot be the value of a tunable parameter, see “Choice of Environment for Varying Parameters Between Simulation Runs” on page 15-10 to determine whether `sim` provides a viable alternative to rapid simulation.

Varying the Stream of Random Numbers

Varying the initial seed of the random number generator varies the stream of random numbers. The following sections include several common scenarios in which the Event-Based Random Number block generates the numbers. When you follow the procedures in the following sections, set the **Initial seed** parameter to the name of a workspace variable:

- “Random Intergeneration Time” on page 15-22
- “Random Attribute Value” on page 15-23
- “Random Service Time from a Signal” on page 15-23
- “Deterministic or Random Service Time, Stored with Each Entity” on page 15-24
- “Random Timeout Interval” on page 15-24
- “Random Event-Based Signal” on page 15-25

Changing the value of the initial seed changes the stream of random numbers that the Event-Based Random Number block creates during the simulation.

Varying Intergeneration Times

In a Time-Based Entity Generator block, to facilitate varying the intergeneration time, use one of these procedures:

Deterministic Intergeneration Time

- 1** Set **Generate entities with** to Intergeneration time from port **t** and click **Apply**.
- 2** Connect the **t** signal input port to an Event-Based Sequence block.
- 3** In the Event-Based Sequence block, set **Vector of output values** to the name of a workspace variable that stores a vector of service time values. You can change the values in this vector in each simulation run. The size of this vector must be the same for all simulation runs.

Random Intergeneration Time

- 1** Set **Generate entities with** to Intergeneration time from port **t** and click **Apply**.
- 2** Connect the **t** signal input port to an Event-Based Random Number block.
- 3** In the Event-Based Random Number block, select a type of distribution. Then set one or more remaining parameters to the names of workspace variables. You can change the values of these variables in each simulation run. The array size of each value must be the same for all simulation runs.

Varying Attribute Values

In a Set Attribute block, to facilitate varying attribute values, use one of these procedures:

Deterministic Attribute Value

- 1** Set **Value From** to **Signal** port and click **Apply**.
- 2** Connect the corresponding signal input port to an Event-Based Sequence block.
- 3** In the Event-Based Sequence block, set **Vector of output values** to the name of a workspace variable that stores a vector of service time values. You can change the values in this vector in each simulation run. The size and complexity of this vector must be the same for all simulation runs.

Random Attribute Value

- 1 Set **Value From** to **Signal** port and click **Apply**.
- 2 Connect the corresponding signal input port to an Event-Based Random Number block.
- 3 In the Event-Based Random Number block, select a type of distribution. Then set one or more remaining parameters to the names of workspace variables. You can change the values of these variables in each simulation run. The array size of each value must be the same for all simulation runs.

Varying Service Times

In a server block, to facilitate varying the service time, use one of these procedures:

Deterministic Service Time from a Signal

- 1 Set **Service time from** to **Signal** port **t** and click **Apply**.
- 2 Connect the **t** signal input port to an Event-Based Sequence block.
- 3 In the Event-Based Sequence block, set **Vector of output values** to the name of a workspace variable that stores a vector of service time values. You can change the values in this vector in each simulation run. The size of this vector must be the same for all simulation runs.

Random Service Time from a Signal

- 1 Set **Service time from** to **Signal** port **t** and click **Apply**.
- 2 Connect the **t** signal input port to an Event-Based Random Number block.
- 3 In the Event-Based Random Number block, select a type of distribution. Then set one or more remaining parameters to the names of workspace variables. You can change the values of these variables in each simulation run. The array size of each value must be the same for all simulation runs.

Deterministic or Random Service Time, Stored with Each Entity

- 1** In the server block, set **Service time from** to **Attribute**. Set **Attribute name** to the name of an attribute that stores service times. Click **Apply**.
- 2** In the entity path that precedes the server block, insert a **Set Attribute** block. Each entity that arrives at the server must possess the attribute that stores the service time.
- 3** In the **Set Attribute** block, set **Name** to the attribute name from step 1.
- 4** Follow one of the procedures in “Varying Attribute Values” on page 15-22.

Varying Timeout Intervals

In a **Schedule Timeout** block, to facilitate varying the timeout interval, use one of these procedures:

Deterministic Timeout Interval

- 1** Set **Timeout interval from** to **Signal port ti** and click **Apply**.
- 2** Connect the **ti** signal input port to an **Event-Based Sequence** block.
- 3** In the **Event-Based Sequence** block, set **Vector of output values** to the name of a workspace variable that stores a vector of service time values. You can change the values in this vector in each simulation run. The size of this vector must be the same for all simulation runs.

Random Timeout Interval

- 1** Set **Timeout interval from** to **Signal port ti** and click **Apply**.
- 2** Connect the **ti** signal input port to an **Event-Based Random Number** block.
- 3** In the **Event-Based Random Number** block, select a type of distribution. Then set one or more remaining parameters to the names of workspace variables. You can change the values of these variables in each simulation run. The array size of each value must be the same for all simulation runs.

Varying Event-Based Signals

To facilitate varying an event-based signal that you create based on arbitrary events, use one of these procedures:

Deterministic Event-Based Signal

- 1 Follow the procedure in “Generating Sequences Based on Arbitrary Events” on page 4-10.
- 2 In the Event-Based Sequence block, set **Vector of output values** to the name of a workspace variable that stores a vector of values. You can change the values in this vector in each simulation run. The size and complexity of this vector must be the same for all simulation runs.

Random Event-Based Signal

- 1 Follow the procedure in “Generating Random Signals Based on Arbitrary Events” on page 4-4.
- 2 In the Event-Based Random Number block, select a type of distribution. Then set one or more remaining parameters to the names of workspace variables. You can change the values of these variables in each simulation run. The array size of each value must be the same for all simulation runs.

Example: Computing an Ensemble Average Using MATLAB Code

This example computes ensemble averages for the waiting times in the Single Server Versus N-Server demo model. In particular, the example:

- Modifies the model for statistical analysis.
- Simulates the modified model multiple times, using a different stream of random numbers each time.
- Analyzes the results.

Alternatively, to achieve the same goal using rapid simulation, see “Example: Computing an Ensemble Average Using Rapid Simulation” on page 15-32.

Prerequisite: Write access to the current folder in the MATLAB session

- 1 Modify the model to record average waiting times:
 - a From the MATLAB Command Window, enter `sedemo_single_v_multi_server` to open the model.
 - b In the current folder, save the model as `server_stats.mdl`.
 - c From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model. In the block dialog box, set these parameters:
 - **Limit data points to last** to 1 because only the final value of the statistic is relevant for this example.
 - **Save format** to Array.
 - d In the top portion of the model, replace the block labeled Average Wait with the Discrete Event Signal to Workspace block.
 - e Drag a second copy of the Discrete Event Signal to Workspace block into the model. In the block dialog box, set these parameters:
 - **Variable name** to `simoutN`.
 - **Limit data points to last** to 1.
 - **Save format** to Array.

- f** In the bottom portion of the model, replace the block labeled Average Wait1 with the second Discrete Event Signal to Workspace block.
 - g** Save the model again, which you can now use for statistical analysis of the waiting time.
- 2** (*Optional*): Remove the Display blocks labeled Number Served and Number Served1.
- 3** In the Command Window, run the simulation repeatedly with different seed values, and capture the statistic from each run:

```
% Set up.
nrns = 10; % Number of simulation runs
w1 = zeros(nrns,1); % Preallocate space for array to build for results.
wN = zeros(nrns,1);
h = waitbar(0,'Running simulations. Please wait...','Name','Running');

% Main simulation loop
for k = 1:nrns
    waitbar(k/nrns,h); % Update progress indicator.
    se_randomizeseeds('server_stats'); % Change random number sequence.
    sim('server_stats',[]); % Run simulation.
    w1(k) = simout; % Build array of empirical values for single server.
    wN(k) = simoutN; % Build array of empirical values for N-server.
end

close(h); % Close progress indicator window.

% Compute ensemble averages.
disp('Ensemble average for single server is:');
disp(mean(w1));
disp('Ensemble average for N-server is:');
disp(mean(wN));
```

Sample output follows.

```
Ensemble average for single server is:
```

```
1.9898
```

```
Ensemble average for N-server is:
```

3.4567

Results vary because the initial seeds for the random number generators are not deterministic. If you want repeatable results instead, specify a global seed when invoking `se_randomize_seeds`, using a different global seed in each iteration of the loop.

Example: Varying the Number of Servers Using MATLAB Code

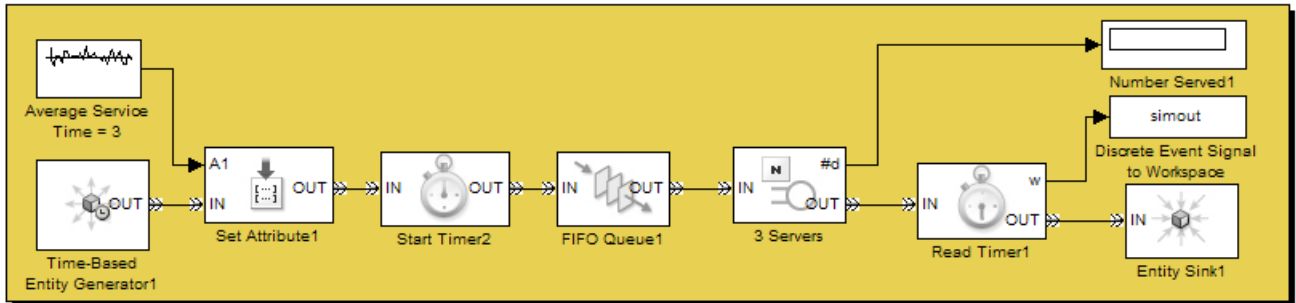
This example simulates a model with different values of N in an N-Server block. In particular, this example:

- Modifies the model for statistical analysis with a varying parameter
- Simulates multiple times with different values of the parameter
- Analyzes results

Prerequisite: Write access to the current folder in the MATLAB session

This example uses the Single Server Versus N-Server demo model.

- 1** Modify the model to record average waiting times:
 - a** In the MATLAB Command Window, enter `sedemo_single_v_multi_server` to open the model .
 - b** In the current folder, save the model as `server_vary_n.mdl` .
 - c** Remove the entire portion of the model labeled “Single Server with Service Time = 1”. The portion of the model labeled “N-Server with N=3, Each with Service Time = 3” remains.
 - d** From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model.
 - e** In the Discrete Event Signal to Workspace block dialog box:
 - Set **Limit data points to last** to 1 because only the final value of the statistic is relevant for this example.
 - Set **Save format** to Array.
 - f** Replace the block labeled Average Wait1 with the Discrete Event Signal to Workspace block. The model looks like the following figure.



- 2 Modify the model to accept varying values of the mean service time and number of servers, from one simulation run to the next:
 - a In the block labeled Average Service Time = 3, open the dialog box and set **Mean** to N. N is a MATLAB variable that you define later.
 - b In the block labeled 3 Servers, open the dialog box and set **Number of servers** to N.
 - c Save the model again, which you can now use for statistical analysis of the waiting time with varying values of N.
- 3 (Optional): Remove the Display block labeled Number Served1.
- 4 In the Command Window, run the simulation with different values of N, and capture the statistic from each run. It takes some time to finish running.

```
% Set up.
nrns = 20; % Number of simulation runs
Nvec = (1:5); % Values of N
nN = length(Nvec); % Number of values in Nvec

% Preallocate space for results.
w = zeros(nrns,1);
wavg = zeros(nN,1);

% Vary the mean arrival rate.
for Nidx = 1:nN
    % N is a parameter in two blocks, so changing N changes the number of
    % servers and the mean service rate in the simulation.
    N = Nvec(Nidx);
```

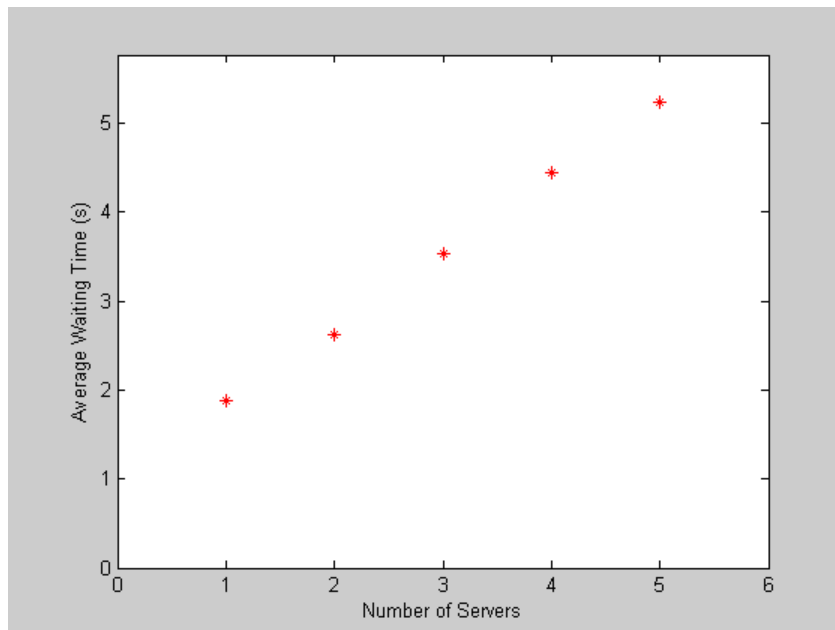
```
disp(['Simulating with number of servers = ' num2str(N)]);

% Compute ensemble average for each value of N.
for k = 1:nruns
    se_randomize seeds('server_vary_n'); % Change random number sequence.
    sim('server_vary_n',1000); % Run simulation.
    w(k) = simout; % Store empirical value of w.
end

wavg(Nidx) = mean(w); % Average for fixed value of N
end

figure; % Create new figure window.
plot(Nvec,wavg,'r*'); % Plot empirical values.
xlabel('Number of Servers')
ylabel('Average Waiting Time (s)')
axis([0 max(Nvec)+1 0 max(wavg)*1.1]);
```

A sample plot shows the average waiting time (averaged over multiple simulations for each fixed value of N) against N.



Example: Computing an Ensemble Average Using Rapid Simulation

In this section...
“Example Scenario” on page 15-32
“Prerequisites” on page 15-32
“Steps in the Example” on page 15-32
“Results” on page 15-35

Example Scenario

This example computes ensemble averages for the waiting times in the Single Server Versus N-Server demo model. In particular, the example:

- Modifies the model for statistical analysis using rapid simulation.
- Builds a rapid simulation executable and a MAT-file containing input data representing seeds for the random number generator.
- Simulates the executable multiple times, using a different stream of random numbers each time.
- Analyzes the results.

Alternatively, to achieve the same goal without using rapid simulation, see “Example: Computing an Ensemble Average Using MATLAB Code” on page 15-26.

Prerequisites

- Real-Time Workshop software license
- Write access to the current folder in the MATLAB session

Steps in the Example

- 1 Modify the model to record average waiting times:

- a** From the MATLAB Command Window, enter `sedemo_single_v_multi_server` to open the model.
 - b** In the current folder, save the model as `server_stats_rsim.mdl`.
 - c** From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model. In the block dialog box, set these parameters:
 - **Limit data points to last** to 1 because only the final value of the statistic is relevant for this example.
 - **Save format** to Array.
 - d** In the top portion of the model, replace the block labeled Average Wait with the Discrete Event Signal to Workspace block.
 - e** Drag a second copy of the Discrete Event Signal to Workspace block into the model. In the block dialog box, set these parameters:
 - **Variable name** to `simoutN`.
 - **Limit data points to last** to 1.
 - **Save format** to Array.
 - f** In the bottom portion of the model, replace the block labeled Average Wait1 with the second Discrete Event Signal to Workspace block.
 - g** Save the model again, which you can now use for statistical analysis of the waiting time.
 - 2** Modify the model to accept varying values of the seeds for the random number generators:
 - a** In the Command Window, define variables called `seed` and `seedN`:

```
seed = 1;  
seedN = 1;
```
 - b** In the dialog box of the block labeled Average Service Time = 1, set **Initial seed** to `seed`.
 - c** In the dialog box of the block labeled Average Service Time = 3, set **Initial seed** to `seedN`.
 - 3** Configure and build the model for rapid simulation:

- a** From the model window, select **Simulation > Configuration Parameters**.
 - b** In the Configuration Parameters dialog box, on the Real-Time Workshop pane, set **System target file** to `rsim.tlc`. Click **Apply**.
 - c** In the Configuration Parameters dialog box, on the Real-Time Workshop RSim Target pane, select **Enable RSim executable to load parameters from a MAT-file**.
 - d** In the Configuration Parameters dialog box, on the Optimization pane, select **Inline parameters**. Click **Configure**.
 - e** In the Model Parameter Configuration dialog box, in the Source list section, select `seed` and `seedN`. Click **Add to table**. Click **OK**.
 - f** In the Configuration Parameters dialog box, click **OK**.
 - g** From the model window, select **Tools > Real-Time Workshop > Build Model**.
- 4** From the Command Window, set up input data by creating and saving a MAT-file. The file contains parameter sets. Each parameter set designates a pair of seed values for one simulation run.

```
% Define the number of simulation runs.
nruns = 10;
% Create vectors of seed values.
seedvalues = floor(rand(nruns,1)*50000);
seedvaluesN = floor(rand(nruns,1)*50000);
% Create a parameter structure to store the seed values.
rtP = rsimgetrtP('server_stats_rsim','AddTunableParamInfo','on');
rtP = rsimsetrtpparam(rtP,nruns);
for k = 1:nruns
    rtP = rsimsetrtpparam(rtP,k,...
        'seed',seedvalues(k),...
        'seedN',seedvaluesN(k));
end
save server_stats_rsim_seeds rtP
```

- 5** From the Command Window, run the executable multiple times, using a different parameter set each time:

```
for k = 1:nruns
```

```

% Create a system command that runs the executable
% file with a particular pair of seeds,
% and stores results in a corresponding MAT-file.
cmd = ['server_stats_rsim -o server_stats_rsim_results' ...
      num2str(k) '.mat -p server_stats_rsim_seeds.mat@' ...
      num2str(k)];
% Execute the command in the operating system.
system(cmd);
end

```

Alternative After setting appropriate environment variables, you can run the executable from a shell window instead of from the Command Window. For more information about environment variables to set, see “Requirements for Running Rapid Simulations” in the Real-Time Workshop documentation.

6 Analyze simulation results. This example computes ensemble averages:

```

% Preallocate space in vectors.
w1 = zeros(nruns,1);
wN = zeros(nruns,1);
for k = 1:nruns
    % Load results from one simulation run.
    load(['server_stats_rsim_results' num2str(k)]);
    % Store results in the vectors.
    w1(k) = rt_simout;
    wN(k) = rt_simoutN;
end

% Compute ensemble averages.
disp('Ensemble average for single server is:');
disp(mean(w1));
disp('Ensemble average for N-server is:');
disp(mean(wN));

```

Results

Sample output follows. Results vary because the initial seeds for the random number generators are not deterministic.

Ensemble average for single server is:
1.9787

Ensemble average for N-server is:
3.4548

Example: Varying Attribute Values Between Runs Using Rapid Simulation

In this section...

“Example Scenario” on page 15-37

“Prerequisites” on page 15-37

“Steps in the Example” on page 15-37

“Results” on page 15-42

Example Scenario

This example illustrates how to change values of attributes between rapid simulation runs. The example:

- Modifies a model to make it possible to change attribute values.
- Builds a rapid simulation executable and a MAT-file containing input data representing different attribute values.
- Simulates the executable multiple times, using a different set of attribute values each time.
- Plots the results.

Prerequisites

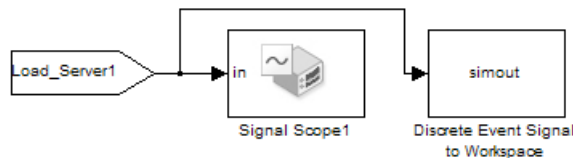
- Real-Time Workshop software license
- Write access to the current folder in the MATLAB session

Steps in the Example

This example uses the Distributed Processing of Multi-Class Jobs demo model.

- 1 Modify the model to record a relevant simulation result. In this case, the result is the load of Service Station 1:
 - a From the MATLAB Command Window, enter `sedemo_star_routing` to open the model.

- b** In the current folder, save the model as `star_routing_rsim.mdl`.
- c** Open the Scopes subsystem by double-clicking it.
- d** From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model.
- e** Branch the line from the Load_Server1 block to the Signal Scope1 block. Connect the new branch to the Discrete Event Signal to Workspace block. The blocks look like this figure:

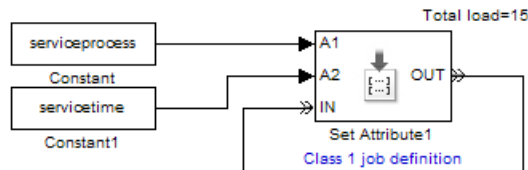


- f** Save the model again.
- 2** Modify the model to accept varying values of attributes that represent job definitions:
- a** In the Command Window, define variables called `serviceprocess` and `servicetime`:


```
serviceprocess = [1 2 4 2 3 5];
servicetime = [2 1 5 3 4 0];
```
 - b** Open the Multiple-Class Job Generation subsystem by double-clicking it.
 - c** Open the dialog box of the Set Attribute block labeled Class 1 job definition. In the **A1** and **A2** rows of the table, set **Value From** to Signal port.
 - d** From the Simulink Sources library, drag two copies of the Constant block into the subsystem.
 - e** Connect the Constant blocks to the **A1** and **A2** signal input ports of the Set Attribute block.
 - f** In the block dialog boxes of the Constant blocks, set **Constant value** to the values in the table.

In the Constant Block that Connects to the...	Set Constant Value to...
A1 signal input port	serviceprocess
A2 signal input port	servicetime

The blocks look like this figure:



- 3 Configure and build the model for rapid simulation:
 - a From the model window, select **Simulation > Configuration Parameters**.
 - b In the Configuration Parameters dialog box, on the Real-Time Workshop pane, set **System target file** to `rsim.tlc`. Click **Apply**.
 - c In the Configuration Parameters dialog box, on the Real-Time Workshop RSim Target pane, select **Enable RSim executable to load parameters from a MAT-file**.
 - d In the Configuration Parameters dialog box, on the Optimization pane, select **Inline parameters**. Click **Configure**.
 - e In the Model Parameter Configuration dialog box, in the Source list section, select `serviceprocess` and `servicetime`. Click **Add to table**. Click **OK**.
 - f In the Configuration Parameters dialog box, click **OK**.
 - g From the model window, select **Tools > Real-Time Workshop > Build Model**.
- 4 From the Command Window, set up input data by creating and saving a MAT-file. The file contains parameter sets. Each parameter set designates a service process vector and a service time vector for one simulation run.

```
% Create a cell array with service process values in the first column,  
% and corresponding service time values in the second column.  
  
service_array = cell(3,2); % Preallocate space.  
% Baseline parameter set  
service_array(1,:) = {[1 2 4 2 3 5], [2 1 5 3 4 0]};  
% Try swapping the roles of stations 1 and 3 in the service process.  
service_array(2,:) = {[3 2 4 2 1 5], [4 1 5 3 2 0]};  
% Try moving the role of station 3 to the beginning of the service process.  
service_array(3,:) = {[3 1 2 4 2 5], [4 2 1 5 3 0]};  
  
% Create a parameter structure to store the values.  
rtP = rsimgetrtP('star_routing_rsim','AddTunableParamInfo','on');  
rtP = rsimsetrtpparam(rtP,3);  
for k = 1:3  
    rtP = rsimsetrtpparam(rtP,k,...  
        'serviceprocess',service_array{k,1},...  
        'servicetime',service_array{k,2});  
end  
save star_routing_rsim_service rtP
```

5 From the Command Window, run the executable multiple times, using a different parameter set each time:

```
% Run the executable with different pairs of service process and  
% service time values. Store results in corresponding MAT-files.  
system(['star_routing_rsim -o star_routing_rsim_results1.mat '...  
    '-p star_routing_rsim_service.mat@1']);  
  
system(['star_routing_rsim -o star_routing_rsim_results2.mat '...  
    '-p star_routing_rsim_service.mat@2']);  
  
system(['star_routing_rsim -o star_routing_rsim_results3.mat '...  
    '-p star_routing_rsim_service.mat@3']);
```

Alternative After setting appropriate environment variables, you can run the executable from a shell window instead of from the Command Window. For more information about environment variables to set, see “Requirements for Running Rapid Simulations” in the Real-Time Workshop documentation.

6 Plot simulation results:

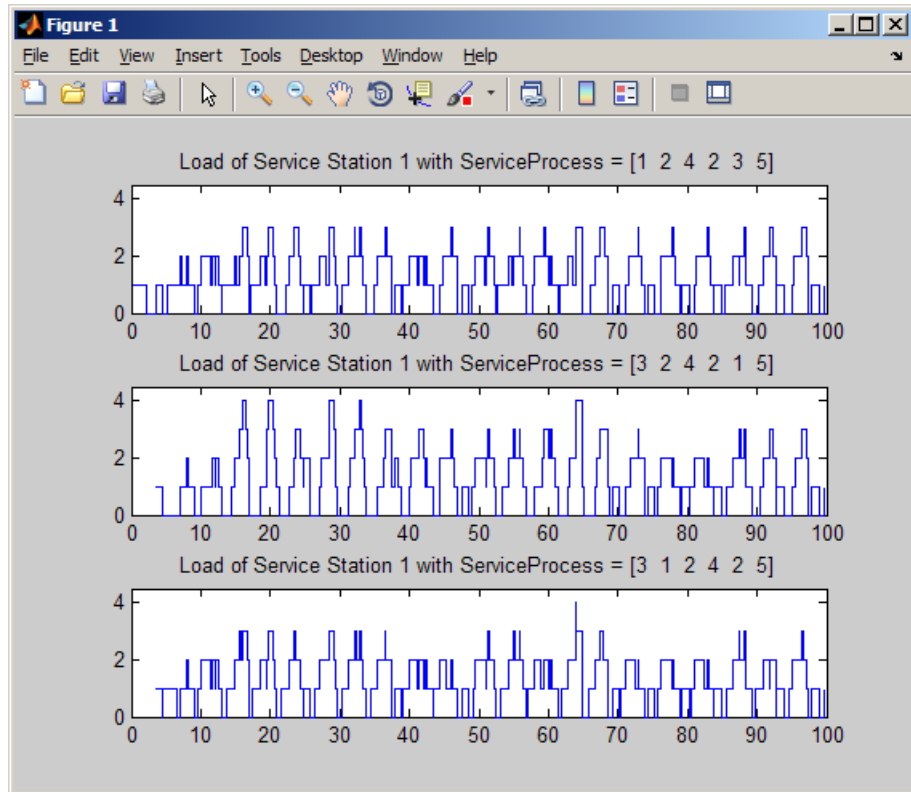
```
h = figure;

subplot(3,1,1);
load star_routing_rsim_results1;
stairs(rt_simout.time,rt_simout.signals.values);
title(['Load of Service Station 1 with ServiceProcess = [' ...
      num2str(service_array{1,1}) ']]);
axis([0 100 0 4.5])

subplot(3,1,2);
load star_routing_rsim_results2;
stairs(rt_simout.time,rt_simout.signals.values);
title(['Load of Service Station 1 with ServiceProcess = [' ...
      num2str(service_array{2,1}) ']]);
axis([0 100 0 4.5])

subplot(3,1,3);
load star_routing_rsim_results3;
stairs(rt_simout.time,rt_simout.signals.values);
title(['Load of Service Station 1 with ServiceProcess = [' ...
      num2str(service_array{3,1}) ']]);
axis([0 100 0 4.5])
```

Results



Example: Varying Queue Capacity Between Runs Using Rapid Simulation

In this section...

“Example Scenario” on page 15-43

“Prerequisites” on page 15-43

“Steps in the Example” on page 15-43

“Results” on page 15-47

Example Scenario

This example explores the effect of changing the capacity of a queue, in a model that discards entities that attempt to arrive at a full queue. The example:

- Modifies the model to make it possible to change queue capacity while using rapid simulation. The modification uses the Variable-Capacity Queue block in the Variable-Capacity Queue demo model.
- Builds a rapid simulation executable and a MAT-file containing input data representing different values of queue capacity.
- Simulates the executable multiple times, using a different queue capacity each time.
- Gathers the results.

Prerequisites

- Real-Time Workshop software license
- Write access to the current folder in the MATLAB session

Steps in the Example

- 1 Modify the model to record a relevant result. In this case, the result is a signal that indicates, for each entity, whether the entity advances from the

switch to the queue. This signal enables you to compute the percentage of entities that cannot enter the queue because it is full.

- a From the MATLAB Command Window, enter `sedemo_flushing_queue` to open the model.
- b In the current folder, save the model as `variable_queue_rsim.mdl`.
- c Open the block dialog box of the Output Switch block. On the **Statistics** tab, set **Last entity departure port, last** to 0n. Click **OK**.

As a result, the block acquires a signal output port whose label is **last**. During the simulation, the value of the signal at this port is:

- 1, if the entity advances to the queue
- 2, if the simulation discards the entity because the queue is full

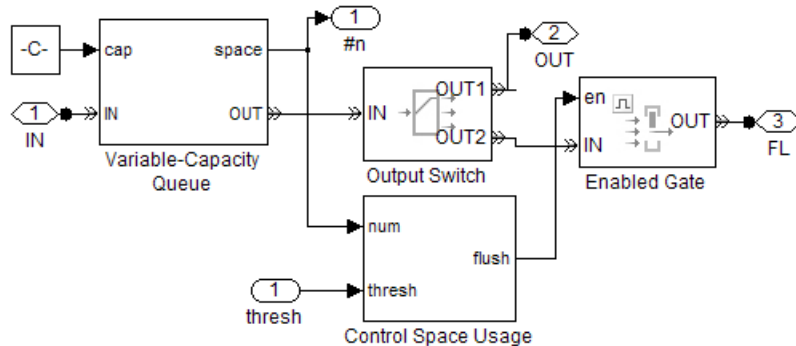
- d From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model.
 - e On the Output Switch block, connect the **last** signal output port to the Discrete Event Signal to Workspace block.
- 2** Modify the model to accept varying values of the queue capacity:

- a From the Command Window, open the Variable-Capacity Queue model by entering `sedemo_variable_capacity_queue`. This model contains a queue block whose capacity comes from a signal.
- b In the Command Window, define a variable called `queuecapacity`:

```
queuecapacity = 10;
```

- c In the `variable_queue_rsim` model, open the Queue with Flushing Threshold subsystem by double-clicking it.
- d Replace the FIFO Queue block in the subsystem with the Variable-Capacity Queue block from the Variable-Capacity Queue model. Then close the Variable-Capacity Queue model.
- e From the Simulink Sources library, drag the Constant block into the subsystem.
- f In the block dialog box of the Constant block, set **Constant value** to `queuecapacity`.

- g** Connect the Constant block to the **cap** signal input port of the Variable-Capacity Queue block. The subsystem now looks like this figure:



- h** At the top level of the `variable_queue_rsim` model, in the dialog box of the Constant block whose value is 9, set **Constant value** to 1000. This example does not use the flushing capability of the queue. Making the flushing threshold exceed the queue capacity is a convenient way to prevent flushing during the simulation.
- i** Save the `variable_queue_rsim` model again. You can now use it for statistical analysis of the entities that attempt to arrive at the queue.
- 3** Configure and build the model for rapid simulation:
- a** From the model window, select **Simulation > Configuration Parameters**.
 - b** In the Configuration Parameters dialog box, on the Real-Time Workshop pane, set **System target file** to `rsim.tlc`. Click **Apply**.
 - c** In the Configuration Parameters dialog box, on the Real-Time Workshop RSim Target pane, select **Enable RSim executable to load parameters from a MAT-file**.
 - d** In the Configuration Parameters dialog box, on the Optimization pane, select **Inline parameters**. Click **Configure**.

- e** In the Model Parameter Configuration dialog box, in the Source list section, select `queuecapacity`. Click **Add to table**. Click **OK**.
 - f** In the Configuration Parameters dialog box, click **OK**.
 - g** From the model window, select **Tools > Real-Time Workshop > Build Model**.
- 4** From the Command Window, set up input data by creating and saving a MAT-file. The file contains parameter sets. Each parameter set designates a queue capacity value for one simulation run.

```
% Define a variable for the name of the system.
mysystem = 'variable_queue_rsim';
% Define the number of simulation runs.
nruns = 10;
% Create a vector of queue capacity values.
queuecapacityvalues = (3:3+nruns-1);
% Create a parameter structure to store the values.
rtP = rsimgetrtP(mysystem, 'AddTunableParamInfo', 'on');
rtP = rsimsetrtpparam(rtP, nruns);
for k = 1:nruns
    rtP = rsimsetrtpparam(rtP, k, ...
        'queuecapacity', queuecapacityvalues(k));
end
save([mysystem '_queuecap'], 'rtP')
```

- 5** From the Command Window, run the executable multiple times, using a different parameter set each time:

```
for k = 1:nruns
    % Create a system command that runs the executable
    % file with a particular queue capacity,
    % and stores the result in a corresponding MAT-file.
    cmd = [mysystem ...
        ' -o ' mysystem '_results' num2str(k) '.mat' ...
        ' -p ' mysystem '_queuecap.mat@' num2str(k)];
    % Execute the command in the operating system.
    system(cmd);
end
```

Alternative After setting appropriate environment variables, you can run the executable from a shell window instead of from the Command Window. For more information about environment variables to set, see “Requirements for Running Rapid Simulations” in the Real-Time Workshop documentation.

6 Gather simulation results:

```
% Preallocate space in a vector.
percentdiscarded = zeros(nruns,1);
for k = 1:nruns
    % Load results from one simulation run.
    load([mysystem '_results' num2str(k)]);
    % Compute the percentage of entities that the model discarded.
    outputport = rt_simout.signals.values;
    total_entities = length(outputport);
    entities_discarded = length(find(outputport == 2));
    % Store result in the vector.
    percentdiscarded(k) = entities_discarded / total_entities;
end
disp('Queue Capacity / Percentage of Entities Discarded')
disp([queuecapacityvalues', percentdiscarded])
```

Results

```
Queue Capacity / Percentage of Entities Discarded
3.0000    0.4242
4.0000    0.3939
5.0000    0.3788
6.0000    0.3636
7.0000    0.3485
8.0000    0.3333
9.0000    0.3182
10.0000   0.3030
11.0000   0.2879
12.0000   0.2727
```

Limitations of Rapid Simulation for Discrete-Event Simulations

SimEvents models do not support certain features of Real-Time Workshop software. The table indicates how to prevent errors that arise from unsupported configurations.

Feature Not Supported	Action Before Building SimEvents Models
Targets other than Rapid Simulation	In the Configuration Parameters dialog box, on the Real-Time Workshop Interface pane, set System target file to <code>rsim.tlc</code> .
On 32-bit Microsoft® Windows® platforms, compilers other than Microsoft compilers	On 32-bit Windows platforms, enter <code>mex -setup</code> . Select a Microsoft compiler that appears in the Real-Time Workshop column of a table on the Supported and Compatible Compilers page on the MathWorks Web site.
C++ code generation	In the Configuration Parameters dialog box, on the Real-Time Workshop pane, set Language to <code>C</code> .
Simulation in External mode	In the Configuration Parameters dialog box, on the Real-Time Workshop Interface pane, set Interface to <code>None</code> .
Fixed-step solvers	In the Configuration Parameters dialog box: <ol style="list-style-type: none"> 1 On the Real-Time Workshop RSim Target pane, set Solver selection to <code>auto</code>. 2 On the Solver pane, set Type to <code>Variable-step</code>.
Automatic suppression of building if you have not changed the model since the last build. Selecting Tools > Real-Time Workshop > Build	No action required.

Feature Not Supported	Action Before Building SimEvents Models
Model always causes the model to build.	

See also “Rapid Simulation Target Limitations” in the Real-Time Workshop documentation.

Learning More About SimEvents Software

Complementing the information in “How Simulink Works” and “Simulating Dynamic Systems” in the Simulink documentation, this section describes some aspects that are different for models that involve event-based processing.

- “Event Sequencing” on page 16-2
- “Choosing How to Resolve Simultaneous Signal Updates” on page 16-7
- “Resolution Sequence for Input Signals” on page 16-8
- “Livelock Prevention” on page 16-16
- “Notifications and Queries Among Blocks” on page 16-18
- “Notifying, Monitoring, and Reactive Ports” on page 16-21
- “Interleaving of Block Operations” on page 16-25
- “Zero-Duration Values and Time-Based Blocks” on page 16-31
- “Update Sequence for Output Signals” on page 16-34
- “Storage and Nonstorage Blocks” on page 16-37

Event Sequencing

In this section...
“Processing Sequence for Simultaneous Events” on page 16-2
“Role of the Event Calendar” on page 16-3
“For Further Information” on page 16-5

Processing Sequence for Simultaneous Events

Even if simultaneous events occur at the same value of the simulation clock, the application processes them sequentially. The processing sequence must reflect causality relationships among events. This table describes the multiple-phase approach the application uses to determine a processing sequence for simultaneous events for which causality considerations alone do not determine a unique correct sequence.

Phase	Events Processed in This Phase	Processing Sequence for Multiple Events in This Phase
1	Events not scheduled on the event calendar	Arbitrary.
2	Events with priority SYS1	Same as the scheduling sequence (FIFO).
3	Events with priority SYS2	Same as the scheduling sequence (FIFO).
4	Events with numerical priority values	Ascending order of priority values. For equal-priority events, the sequence is random or arbitrary, depending on the model’s Execution order parameter.

When the sequence is arbitrary, you should not make any assumptions about the sequence or its repeatability.

The events with priority SYS1 enable the application to detect multiple signal updates before reacting to any of them. The events with priority SYS2 enable entities to advance in response to state changes.

For suggestions on how to use the information in the table when creating models, see “Choosing an Approach for Simultaneous Events” on page 3-7.

Role of the Event Calendar

During a simulation, the application maintains a list, called the *event calendar*, of selected upcoming events that are scheduled for the current simulation time or future times. By referring to the event calendar, the application executes events at the correct simulation time and in an appropriate sequence.

The table below indicates which events are scheduled or might be scheduled on the event calendar. In some cases, you have a choice.

Event Name	Event Type in Debugger	Scheduled On Event Calendar	How to Schedule Event on Event Calendar
Counter reset	CounterReset	Yes	
Delayed restart	DelayedRestart	Yes	
Entity advancement		No	
Entity destruction		No	
Entity generation	EntityGeneration	Yes	
Entity request	EntityRequest	Yes	
Function call	FunctionCall	Maybe	Select Resolve simultaneous signal updates according to event priority , if present, in the dialog box of the block that generates the function call. If the dialog box has no such option, the function call is not scheduled on the event calendar.
Gate (opening or closing)	Gate	Yes	
Memory read	MemoryRead	Maybe	Select Resolve simultaneous signal updates according to event priority on the Read tab of the Signal Latch block's dialog box.
Memory write	MemoryWrite	Maybe	Select Resolve simultaneous signal updates according to event priority on the Write tab of the Signal Latch block's dialog box.
Port selection	PortSelection	Yes	
Preemption		No	
Release	Release	Yes	

Event Name	Event Type in Debugger	Scheduled On Event Calendar	How to Schedule Event on Event Calendar
Sample time hit		No	
Service completion	ServiceCompletion	Yes	
Storage completion	StorageCompletion	Yes	
Subsystem	Subsystem	Maybe	Select Resolve simultaneous signal updates according to event priority in the dialog box of the Discrete Event Inport block that corresponds to the signal that causes the execution.
Timeout	Timeout	Yes	
Trigger		No	
Value change		No	

When you use blocks that offer a **Resolve simultaneous signal updates according to event priority** option, your choice determines whether, or with what priority, particular events are scheduled on the event calendar. For information about this option, see “Resolution Sequence for Input Signals” on page 16-8 and “Choosing How to Resolve Simultaneous Signal Updates” on page 16-7.

For Further Information

- Chapter 3, “Managing Simultaneous Events” — Resolving race conditions in discrete-event simulations
- “Viewing the Event Calendar” on page 14-46 — Displaying event information in the MATLAB Command Window using the SimEvents debugger

- “Resolution Sequence for Input Signals” on page 16-8 — How the application resolves updates in input signals

Choosing How to Resolve Simultaneous Signal Updates

The **Resolve simultaneous signal updates according to event priority** option lets you defer certain operations until the application determines which other operations are supposed to be simultaneous. To use this option appropriately, you should understand your modeling goals, your model's design, and the way the application processes signal updates that are simultaneous with other operations in the simulation. The table indicates sources of relevant information that can help you use the **Resolve simultaneous signal updates according to event priority** option.

To Read About...	Refer to...	Description
Background	“Detection of Signal Updates” on page 16-8 and “Effect of Simultaneous Operations” on page 16-9	What simultaneous signal updates are, and the context in which the option is relevant
Behavior	“Specifying Event Priorities to Resolve Simultaneous Signal Updates” on page 16-10	How the simulation behaves when you select the option
	“Resolving Simultaneous Signal Updates Without Specifying Event Priorities” on page 16-12	How the simulation behaves when you do not select the option
Examples	“Example: Effects of Specifying Event Priorities” on page 3-26	Illustrates the significance of the option
	“Example: Choices of Values for Event Priorities” on page 3-11	Examines the role of event priority values, assuming you have selected the option
Tips	“Choosing an Approach for Simultaneous Events” on page 3-7	Tips to help you decide how to configure your model

For more general information about simultaneous events, information in “Overview of Simultaneous Events” on page 3-2 is also relevant.

Resolution Sequence for Input Signals

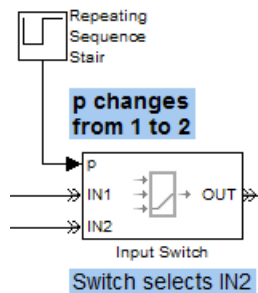
In this section...
“Detection of Signal Updates” on page 16-8
“Effect of Simultaneous Operations” on page 16-9
“Resolving the Set of Operations” on page 16-10
“Specifying Event Priorities to Resolve Simultaneous Signal Updates” on page 16-10
“Resolving Simultaneous Signal Updates Without Specifying Event Priorities” on page 16-12
“For Further Information” on page 16-15

Detection of Signal Updates

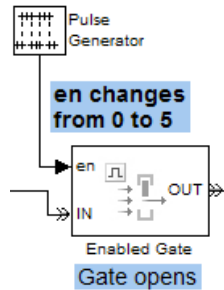
A block that possesses a reactive port listens for relevant updates in the input signal. A relevant update causes the block to react appropriately (for example, by opening a gate or executing a discrete event subsystem).

Example of Signal Updates and Reactions

The schematics below illustrate relevant updates and the blocks’ corresponding reactions.



Signal Update That Causes a Switch to Select a Port



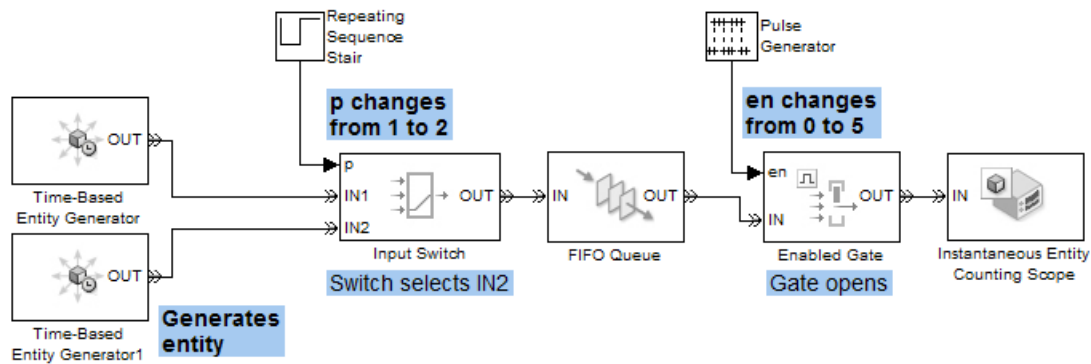
Signal Update That Causes a Gate to Open

Effect of Simultaneous Operations

An update in an input signal is often simultaneous with other operations in the same block or in other blocks in the model. The processing sequence for the set of simultaneous operations can influence the simulation behavior.

Example of Simultaneous Signal Updates

In the model below, two signal updates and one entity-generation event occur simultaneously and independently. The simulation behaves differently depending on the sequence in which it processes these events and their logical consequences (where the **p** signal update is a logical consequence of the update of the **en** signal). Advancement of the newly generated entity is also a potential simultaneous event, but it can occur only if conditions in the switch, queue, and gate blocks permit the entity to advance.



Resolving the Set of Operations

For modeling flexibility, blocks that have reactive ports offer two levels of choices that you can make to refine the simulation's behavior:

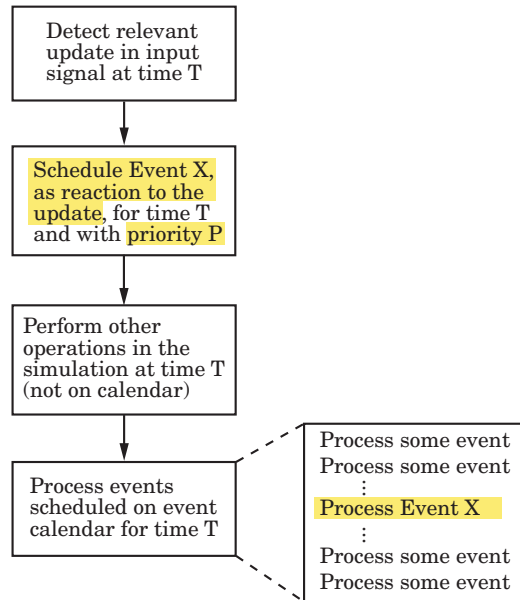
- The **Resolve simultaneous signal updates according to event priority** check box lets you choose the algorithm the application uses to resolve the reactions to signal updates, relative to other simultaneous operations in the simulation.
- If you select **Resolve simultaneous signal updates according to event priority**, the algorithm relies on the relative values of a set of numerical event priorities. You must set the event priority values using parameters in various blocks in the model.

Specifying Event Priorities to Resolve Simultaneous Signal Updates

If you select the **Resolve simultaneous signal updates according to event priority** option in a block that has a reactive port, and if the block detects a relevant update in the input signal that connects to the reactive port, then the application defers reacting to the update until it can determine which other operations are supposed to be simultaneous. Furthermore, the application sequences the reaction to the update using the numerical event priority that you specify.

Schematic Showing Application Processing

The next figure summarizes the steps the application takes when you select **Resolve simultaneous signal updates according to event priority** and the block has a relevant update in its input signal at a given time, T .



Processing for Numerical-Priority Events

Contrast this with the schematics in Processing for System-Priority Events on page 16-13 and Processing for Immediate Updates on page 16-14.

Use of the Event Calendar

To defer reacting to a signal update, the block schedules an event (“Event X” in the schematic) on the event calendar to process the block’s reaction. The scheduled time of the event is the current simulation time, except that the Signal-Based Event to Function-Call Event block lets you specify a time delay. The event priority of the event is the value of the **Event priority** or similarly named parameter in the block dialog box.

After scheduling the event, the application might perform other operations in the model at the current simulation time that are not scheduled on the event calendar. Examples of other other operations can include updating other signals or processing the arrival or departure of entities.

Use of Event Priority Values

When the application begins processing the events that are scheduled on the event calendar for the current simulation time, event priority values influence the processing sequence. For details, see “Processing Sequence for Simultaneous Events” on page 16-2. As a result, the application is resolving the update or change in the input signal (which might be simultaneous with other operations in the same block or in other blocks) according to the relative values of event priorities of all simultaneous events on the event calendar. A particular value of event priority is not significant in isolation; what matters is the ordering in a set of event priorities for a set of simultaneous events.

Resolving Simultaneous Signal Updates Without Specifying Event Priorities

If you do not select the **Resolve simultaneous signal updates according to event priority** option in a block that has a reactive port, and if the block detects a relevant update in the input signal that connects to the reactive port, then the block processes its reaction using one of these approaches:

- Defers reacting to the update until it can determine which other operations are supposed to be simultaneous. Furthermore, the application sequences the reaction to the update using an event priority value called a system priority, denoted SYS1 or SYS2.

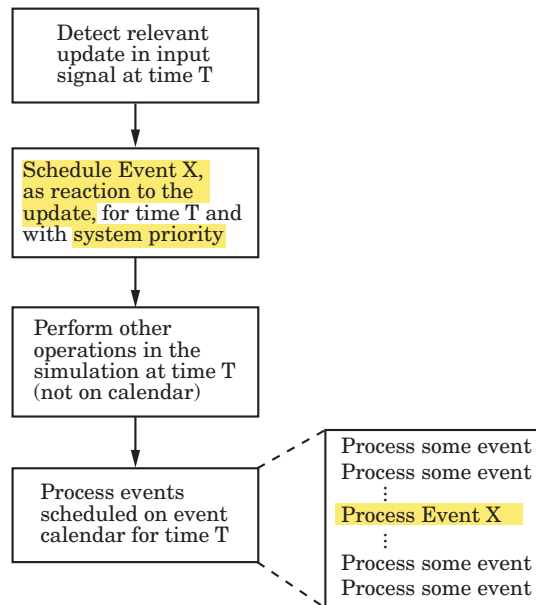
For details, see “System-Priority Events on the Event Calendar” on page 16-12.

- Reacts upon detecting the update (shown as “immediately” in the schematic illustrating this approach). The reaction, such as an execution of a discrete event subsystem, is not deferred, is not scheduled on the event calendar, and has no event priority. As a result, you are not resolving the sequence explicitly.

For details, see “Unprioritized Reactions to Signal Updates” on page 16-14.

System-Priority Events on the Event Calendar

The next figure summarizes the steps the application takes when you choose not to select **Resolve simultaneous signal updates according to event priority** in a block that uses system-priority events and that has a relevant update in its input signal at a given time, T .



Processing for System-Priority Events

Contrast this with the schematics in Processing for Numerical-Priority Events on page 16-11 and Processing for Immediate Updates on page 16-14. The difference between using a system priority and specifying a numerical priority for the same event is that the system priority causes earlier processing; see “Processing Sequence for Simultaneous Events” on page 16-2.

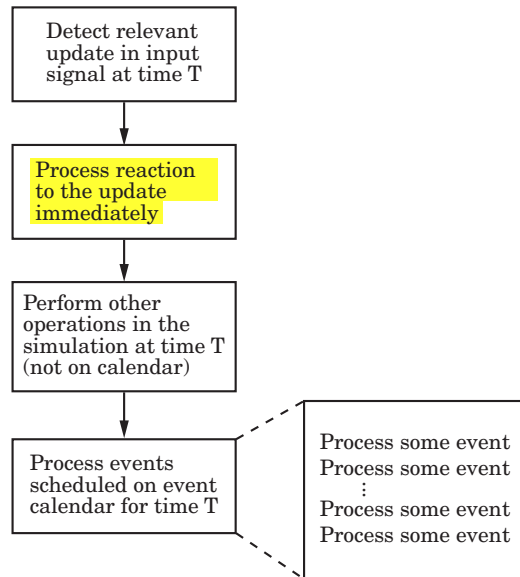
The blocks that can use system-priority events, unlike the blocks that can perform immediate updates, are characterized by the ability to directly change the state of an entity, including its location or attribute:

- Enabled Gate
- Entity Departure Counter
- Event-Based Entity Generator
- Input Switch
- Output Switch
- Path Combiner

- Release Gate

Unprioritized Reactions to Signal Updates

The next figure summarizes the steps the application takes when you do not select **Resolve simultaneous signal updates according to event priority** in a block that performs immediate updates and that has a relevant update in its input signal at a given time, T .



Processing for Immediate Updates

Contrast this with the schematics in Processing for Numerical-Priority Events on page 16-11 and Processing for System-Priority Events on page 16-13.

The blocks that can perform immediate updates, unlike the blocks that can use system-priority events, are characterized by the ability to produce signal outputs as a direct result of signal-based events:

- Discrete Event Inport
- Signal Latch
- Signal-Based Function-Call Event Generator

- Signal-Based Event to Function-Call Event

For Further Information

- Chapter 3, “Managing Simultaneous Events” — Resolving race conditions in discrete-event simulations
- “Reactive Ports” on page 16-23 — What constitutes a relevant update at a reactive port

Livelock Prevention

In this section...

“Overview” on page 16-16

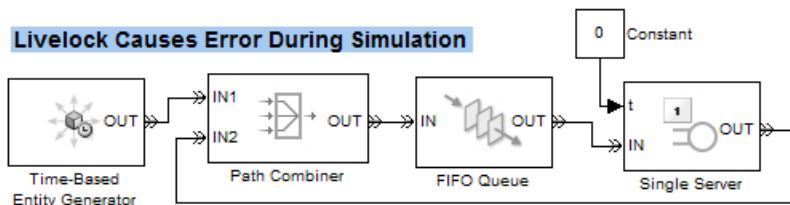
“Permitting Large Finite Numbers of Simultaneous Events” on page 16-16

Overview

SimEvents software includes features to prevent livelock. *Livelock* is a situation in which a block returns to the same state infinitely often at the same time instant. Typical cases include:

- An entity that moves along a looped entity path with no passage of time and no logic to stop the entity for a nonzero period of time
- An intergeneration time of 0 in an entity generator
- A Signal-Based Event to Function-Call Event or Signal-Based Function-Call Event Generator block that calls itself with a time delay of 0

The model below shows an example of livelock. The livelock prevention feature causes the simulation to halt with an error message. Without this error detection, an entity would move endlessly around the looped entity path without the simulation clock advancing.



Permitting Large Finite Numbers of Simultaneous Events

If your simulation creates a large, but not infinite, number of simultaneous events, consider increasing the model's thresholds related to livelock prevention.

For example, if you modify the Preload Queue with Entities demo by setting both the capacity of the queue and the number of iterations of the function-call generator to 2000, then the simulation creates 2000 simultaneous events with no infinite loops. To prevent a spurious error in this situation, increase the model's limit on the number of events per block to at least 2000.

To change the thresholds related to livelock prevention, use this procedure:

- 1** Open the Configuration Parameters dialog box by selecting **Simulation > Configuration Parameters** from the model window's menu bar.
- 2** Navigate to the SimEvents pane of the Configuration Parameters dialog box.
- 3** Change the values of the **Maximum events per block** and **Maximum events per model** parameters.
- 4** Apply the change by clicking **OK** or **Apply**.

Notifications and Queries Among Blocks

In this section...

“Overview of Notifications and Queries” on page 16-18

“Querying Whether a Subsequent Block Can Accept an Entity” on page 16-18

“Notifying Blocks About Status Changes” on page 16-19

Overview of Notifications and Queries

In a variety of situations, a SimEvents block notifies other blocks about changes in its status or queries other blocks about their status. These interactions among blocks occur automatically and are essential to the proper functioning of a discrete-event simulation. Entity request events are one kind of interaction among blocks. Entity request events appear on the event calendar, but other kinds of notifications and queries are not explicitly reported.

Querying Whether a Subsequent Block Can Accept an Entity

Before a SimEvents block outputs an entity, it queries the next block to determine whether that block can accept the entity. For example,

- When an entity arrives at an empty FIFO Queue block, the queue queries the next block. If that block can accept an entity, the queue outputs the entity at the head of the queue; otherwise, the queue holds the entity.
- While a Single Server block is busy serving, it does not query the next block. Upon completion of the service time, the server queries the next block. If that block can accept an entity, the server outputs the entity that has completed its service; otherwise, the server holds the entity.
- When an entity attempts to arrive at a Replicate block, the block queries each of the blocks connected to its entity output ports. If all of them can accept an entity, then the Replicate block copies its arriving entity and outputs the copies; otherwise, the block does not permit the entity to arrive there and the entity must stay in a preceding block.

- After a Time-Based Entity Generator block generates a new entity, it queries the next block. If that block can accept an entity, then the generator outputs the new entity; otherwise, the behavior of the Time-Based Entity Generator block depends on the value of its **Response when blocked** parameter.
- When a block (for example, a Single Server block) attempts to advance an entity to the Input Switch block, the server uses a query to check whether it is connected to the currently selected entity input port of the Input Switch block. If so, the Input Switch queries the next block to determine whether it can accept the entity because the Input Switch block cannot hold an entity for a nonzero duration.
- When an entity attempts to arrive at an Output Switch block, the block must determine which entity output port is selected for departure and whether the block connected to that port can accept the entity. If the **Switching criterion** parameter is set to **First port that is not blocked**, then the Output Switch block might need to query more than one subsequent block to determine whether it can accept the entity. If the **Switching criterion** parameter of the Output Switch block is set to **From attribute**, then the block also requires information about the entity that is attempting to arrive.

Notifying Blocks About Status Changes

When a SimEvents block undergoes certain kinds of status changes, it notifies other blocks of the change. This notification might cause the other blocks to change their behavior or status in some way, depending on the circumstances. For example,

- After an entity departs from a Single Server block, it schedules an entity request event to notify the preceding block that the server's entity input port has changed from unavailable to available.
- After an entity departs from a queue that was full to capacity, the queue schedules an entity request event to notify the preceding block that the queue's entity input port has changed from unavailable to available.
- After an entity departs from a switch or Enabled Gate block, it schedules an entity request event to determine whether another entity can advance from a preceding block. This process repeats until no further entity advancement can occur.

- When a Path Combiner block receives notification that the next block's entity input port has changed from unavailable to available, the Path Combiner block's entity input ports also become available. The block schedules an entity request event to notify preceding blocks that its entity input ports are available.

This case is subtle because the Path Combiner block usually has more than one block to notify, and the sequence of notifications can be significant. See the block's reference page for more information about the options.

- When an entity arrives at a Single Server block that has a **t** signal input port representing the service time, that port notifies the preceding block of the need for a new service time value. If the preceding block is the Event-Based Random Number block, then it responds by generating a new random number that becomes the service time for the arriving entity.

This behavior occurs because the **t** signal input port is a notifying port; see “Notifying, Monitoring, and Reactive Ports” on page 16-21 for details.

Notifying, Monitoring, and Reactive Ports

In this section...

“Overview of Signal Input Ports of SimEvents Blocks” on page 16-21

“Notifying Ports” on page 16-21

“Monitoring Ports” on page 16-22

“Reactive Ports” on page 16-23

Overview of Signal Input Ports of SimEvents Blocks

Signal input ports of SimEvents blocks fall into these categories:

- Notifying ports, which notify the preceding block when a certain event has occurred
- Monitoring ports, which help you observe signal values
- Reactive ports, which listen for updates or changes in the input signal and cause an appropriate reaction in the block possessing the port

The distinctions are relevant when you use the Event-Based Random Number or Event-Based Sequence block. For details, see these topics:

- Event-Based Random Number reference page
- Event-Based Sequence reference page
- “Generating Random Signals” on page 4-4
- “Using Data Sets to Create Event-Based Signals” on page 4-9

Notifying Ports

Notifying ports, listed in the table below, notify the preceding block when a certain event has occurred. When the preceding block is the Event-Based Random Number or Event-Based Sequence block, it responds to the notification by generating a new output value. Other blocks ignore the notification.

List of Notifying Ports

Signal Input Port	Block	Generate New Output Value Upon
A1, A2, A3, etc.	Set Attribute	Entity arrival
in	Signal Latch	Write event
e1, e2	Entity Departure Event to Function-Call Event	Entity arrival
	Signal-Based Event to Function-Call Event	Relevant signal-based event, depending on configuration of block
t	Signal-Based Event to Function-Call Event	Relevant signal-based event, depending on configuration of block
t	Infinite Server	Entity arrival
	N-Server	Entity arrival
	Single Server	Entity arrival
t	Time-Based Entity Generator	Simulation start and subsequent entity departures
ti	Schedule Timeout	Entity arrival
x	X-Y Signal Scope	Sample time hit at in signal input port

Monitoring Ports

Monitoring ports, listed in the table below, help you observe signal values. Optionally, you can use a branch line to connect the Event-Based Random Number or Event-Based Sequence block to one or more monitoring ports. These connections do not cause the block to generate a new output, but merely enable you to observe the signal.

List of Monitoring Ports

Signal Input Port	Block
Unlabeled	Discrete Event Signal to Workspace
in	Signal Scope
	X-Y Signal Scope
ts, tr, vc	Instantaneous Event Counting Scope

Reactive Ports

Reactive ports, listed in the table below, listen for relevant updates in the input signal and cause an appropriate reaction in the block possessing the port. For example, the **p** port on a switch listens for changes in the input signal; the block reacts by selecting a new switch port.

List of Reactive Ports

Signal Input Port	Block	Relevant Update
en	Enabled Gate	Value change from nonpositive to positive, and vice versa
p	Input Switch	Value change
	Output Switch	
	Path Combiner	
ts, tr, vc	Entity Departure Counter	Sample time hit at ts port Appropriate trigger at tr port Appropriate value change at vc port
	Event-Based Entity Generator	
	Release Gate	
	Signal-Based Event to Function-Call Event	
	Signal-Based Function-Call Event Generator	

List of Reactive Ports (Continued)

Signal Input Port	Block	Relevant Update
wts, wtr, wvc, rts, rtr, rvc	Signal Latch	Sample time hit at wts or rts port Appropriate trigger at wtr or rtr port Appropriate value change at wvc or rvc port
Input port corresponding to Discrete Event Inport block in subsystem	Discrete Event Subsystem	Sample time hit at that input port
Unlabeled input port	Initial Value	Sample time hit

For triggers and value changes, “appropriate” refers to the direction you specify in a **Type of change in signal value** or **Trigger type** parameter in the block’s dialog box.

Interleaving of Block Operations

In this section...

“Overview of Interleaving of Block Operations” on page 16-25

“How Interleaving of Block Operations Occurs” on page 16-25

“Example: Sequence of Departures and Statistical Updates” on page 16-26

Overview of Interleaving of Block Operations

During the simulation of a SimEvents model, some sequences of block operations become interleaved when the application processes them. Interleaving can affect the simulation behavior. This section describes and illustrates interleaved block operations to help you understand the processing and make appropriate modeling choices.

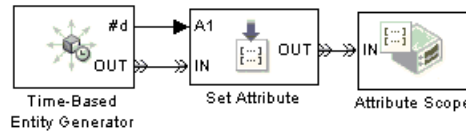
How Interleaving of Block Operations Occurs

At all simulation times from an entity’s generation to destruction, the entity resides in a block (or more than one block, if the entity advances from block to block at a given time instant). Blocks capable of holding an entity for a nonzero duration are called storage blocks. Blocks that permit an entity arrival but must output the entity at the same value of the simulation clock are called nonstorage blocks. During a simulation, whenever an entity departs from a block, the application processes enough queries, departures, arrivals, and other block operations until either a subsequent storage block detects the entity’s arrival or the entity is destroyed. Some block operations, including the updates of statistical output signals that are intended to be updated after the entity’s departure, are deferred until after a subsequent storage block detects the entity’s arrival or the entity is destroyed.

To change the sequence of block operations, you might need to insert storage blocks in key locations along entity paths in your model, as illustrated in “Example: Sequence of Departures and Statistical Updates” on page 16-26. A typical storage block to insert for this purpose is a server whose service time is 0.

Example: Sequence of Departures and Statistical Updates

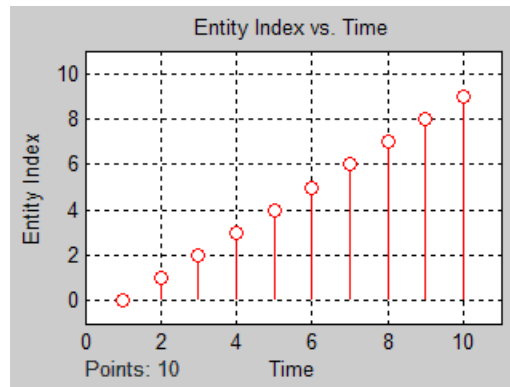
Consider the sequence of operations in the Time-Based Entity Generator, Set Attribute, and Attribute Scope blocks shown below.



At each time $T = 1, 2, 3, \dots, 10$, the application processes the following operations in the order listed:

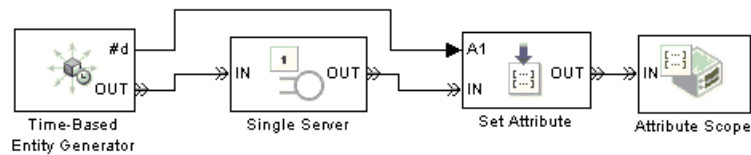
Order	Operation	Block
1	Entity generation	Time-Based Entity Generator
2	Entity advancement to nonstorage block	From Time-Based Entity Generator to Set Attribute
3	Assignment of attribute using value at A1 signal input port	Set Attribute
4	Entity advancement to nonstorage block	From Set Attribute to Attribute Scope
5	Entity destruction	Attribute Scope
6	Update of plot	Attribute Scope
7	Update of signal at #d signal output port	Time-Based Entity Generator

The final operation of the Time-Based Entity Generator block is deliberately processed *after* operations of subsequent blocks in the entity path are processed. This explains why the plot shows a value of 0, not 1, at $T=1$.



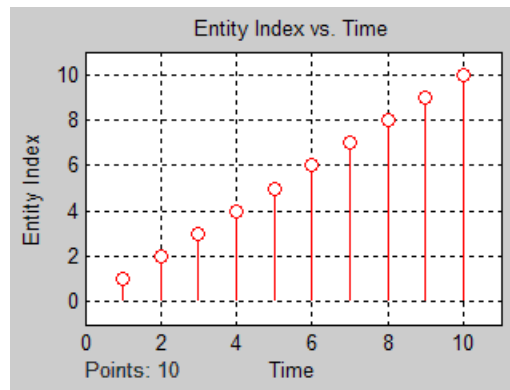
Altering the Processing Sequence

If you want to be sure that the Set Attribute block reads the value at the **A1** signal input port after the Time-Based Entity Generator block has updated its **#d** output signal, then insert a storage block between the two blocks. In this simple model, you can use a Single Server block with a **Service time** parameter of 0. The model, table, and plot are below.



Order	Operation	Block
1	Entity generation	Time-Based Entity Generator
2	Entity advancement to storage block	From Time-Based Entity Generator to Single Server
3	Update of signal at #d signal output port	Time-Based Entity Generator
4	Service completion	Single Server
5	Entity advancement to nonstorage block	From Single Server to Set Attribute

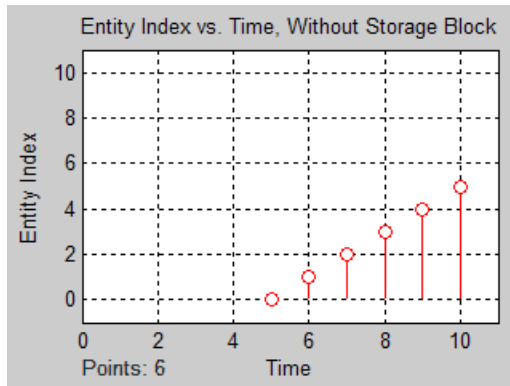
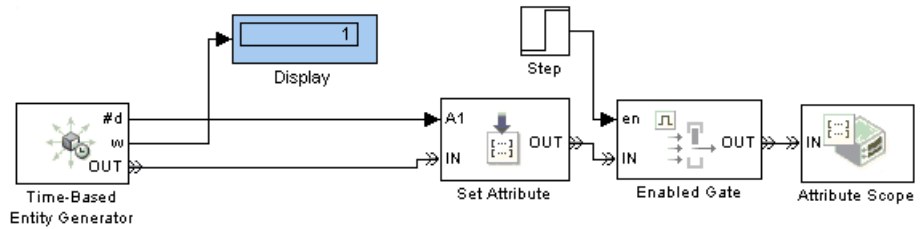
Order	Operation	Block
6	Assignment of attribute using value at A1 signal input port	Set Attribute
9	Entity advancement to nonstorage block	From Set Attribute to Attribute Scope
10	Entity destruction	Attribute Scope
11	Update of plot	Attribute Scope



Consequences of Inserting a Storage Block

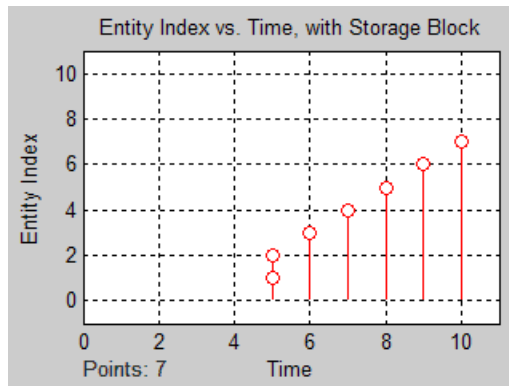
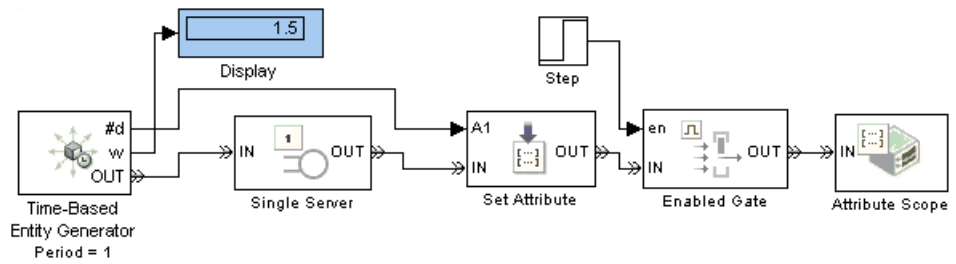
If the storage block you have inserted to alter the processing sequence holds the entity longer than you expect (beyond the zero-duration service time, for example), be aware that your simulation might change in other ways. You should consider the impact of either inserting or not inserting the storage block.

For example, suppose you add a gate block to the preceding example and view the average intergeneration time, w , of the entity generator block. When the gate is closed, a newly generated entity cannot advance immediately to the scope block. Whether this entity stays in the entity generator or a subsequent server block affects the w signal, as shown in the figures below.



Model with Gate and Without Storage Block

When a storage block is present, the first pending entity stays there instead of in the entity generator. The earlier departure of the first entity from the entity generator increases the value of the w signal.



Model with Gate and Storage Block

Zero-Duration Values and Time-Based Blocks

In this section...

“Techniques for Working with Multivalued Signals” on page 16-31

“Example: Using a #n Signal as a Trigger” on page 16-32

Techniques for Working with Multivalued Signals

Because time-based simulations involve signals that assume a unique value at each value of the simulation clock, some blocks designed for time-based simulations recognize only one value of a signal per time instant. Because zero-duration values commonly occur in discrete-event simulations (for example, statistical output signals from SimEvents blocks), you should be aware of techniques for working with zero-duration values. The table below lists examples of time-based blocks that recognize one signal value per time instant, along with similar blocks or techniques that recognize multivalued signals.

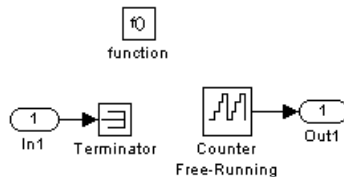
Time-Based Block	Block or Technique for Working with Multivalued Signals
Scope	Signal Scope in the SimEvents Sinks library
To Workspace	Discrete Event Signal to Workspace in the SimEvents Sinks library. Alternatively, put the To Workspace block in a discrete event subsystem.
Triggered Subsystem	Discrete Event Subsystem in the SimEvents Ports and Subsystems library, with the Discrete Event Inport block configured to execute the subsystem upon trigger edges. Alternatively, use the Signal-Based Event to Function-Call Event block in the Event Translation library to convert the trigger signal to a function call, and then use a Function-Call Subsystem instead of a Triggered Subsystem.
Stateflow with a trigger input signal	Use the Signal-Based Event to Function-Call Event block in the Event Translation library to convert the trigger signal to a function call, then call the Stateflow block with a function-call signal instead of a trigger signal.

For an example comparing the Scope viewer with the Signal Scope block, see “Comparison with Time-Based Plotting Tools” on page 11-17.

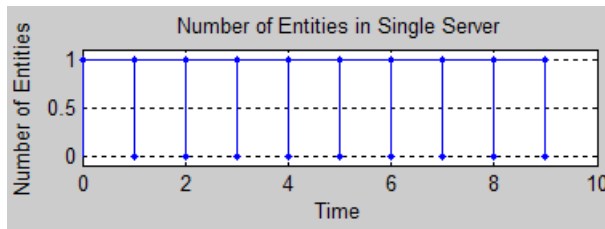
Example: Using a #n Signal as a Trigger

Suppose you want to call a subsystem each time the #n signal from a Single Server block rises from 0 to 1. This signal is 0 when the server is not storing an entity and 1 when the server is storing an entity. It is common for an entity to arrive at a server at the same time instant that the previous entity departed from the server. In this case, the #n signal changes from 1 to 0 and back to 1 in the same time instant. A time-based block that recognizes only one value of a signal per time instant might not recognize a rising edge that occurs in a time interval of length zero.

This example uses a Counter Free-Running block inside a subsystem to count the number of times the subsystem is called. (Be aware that the Counter Free-Running block starts counting from zero, not one.)

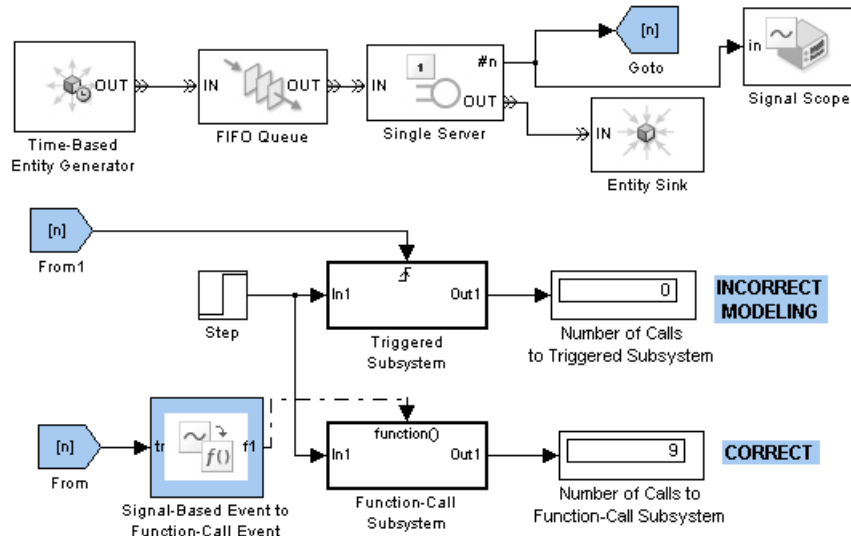


The discrete-event portion of the simulation involves a D/D/1 queuing system in which the server is never idle for a nonzero period of time. As a result, the #n signal exhibits many zero-duration values, shown in the plot below.



The example uses two approaches to try to call the subsystem each time the server's $\#n$ signal rises from 0 to 1:

- The approach using a Triggered Subsystem is unsuitable because it does not count changes that occur in a time interval of length zero. You can see from the Display block that the triggered subsystem is never called.
- The approach using function calls is appropriate because the Signal-Based Event to Function-Call Event block recognizes rising edges of $\#n$ even when they involve zero-duration values. The block converts these rising edges into function calls to which the Function-Call Subsystem responds.



Update Sequence for Output Signals

In this section...

“Determining the Update Sequence” on page 16-34

“Example: Detecting Changes in the Last-Updated Signal” on page 16-35

Determining the Update Sequence

When a block produces more than one output signal in response to events, the simulation behavior might depend on the sequence of signal updates relative to each other. This is especially likely if you use one of the signals to influence a behavior or computation that also depends on another one of the signals, as in “Example: Detecting Changes in the Last-Updated Signal” on page 16-35 and “Example: Detecting Changes from Empty to Nonempty” on page 10-24.

When you turn on more than one output signal from a SimEvents block’s dialog box (typically, from the **Statistics** tab), the block updates each of the signals in a sequence. See the Signal Output Ports table on the block’s reference page to learn about the update order:

- In some cases, a block’s reference page specifies the sequence explicitly using unique numbers in the Order of Update column.

For example, the reference page for the N-Server block indicates that upon entity departures, the **w** signal is updated before the **#n** signal. The Order of Update column in the Signal Output Ports table lists different numbers for the **w** and **#n** signals.

- In some cases, a block’s reference page lists two or more signals without specifying their sequence relative to each other. Such signals are updated in an arbitrary sequence relative to each other and you should not rely on a specific sequence for your simulation results.

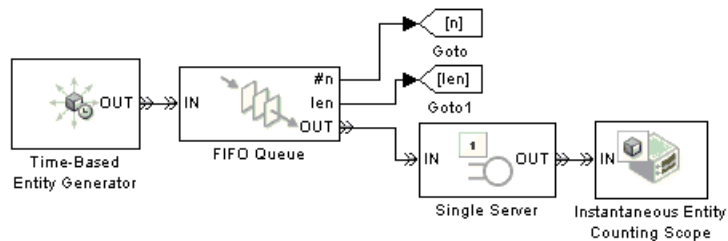
For example, the reference page for the N-Server block indicates that the **w** and **util** signals are updated in an arbitrary sequence relative to each other. The Order of Update column in the Signal Output Ports table lists the same number for both the **w** and **util** signals.

- When a block offers fewer than two signal output ports, the sequence of updates does not need explanation on the block’s reference page. For

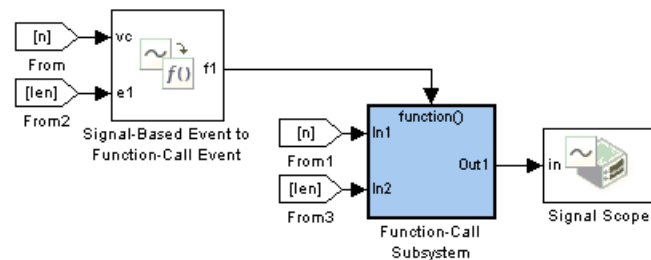
example, the reference page for the Enabled Gate block does not indicate an update sequence because the block can output only one signal.

Example: Detecting Changes in the Last-Updated Signal

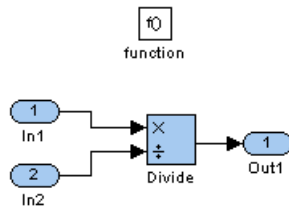
The example below plots the ratio of the queue's current length to the time average of the queue length. The FIFO Queue block produces **#n** and **len** signals representing the current and average lengths, respectively. The computation of the ratio occurs in a function-call subsystem that is called when the Signal-Based Event to Function-Call Event block detects a change in **#n** (as long as **len** is positive, to avoid division-by-zero warnings). Because the FIFO Queue block updates the **len** signal before updating the **#n** signal, both signals are up to date when the value change occurs in the **#n** signal.



Detect Changes in #n Signal



Top-Level Model



Subsystem Contents

If you instead connect the **len** signal to the Signal-Based Event to Function-Call Event block's **vc** input port, then the block issues a function call upon detecting a change in the **len** signal. At that point, the **#n** value is left over from the block's previous arrival or departure, so the computed ratio is incorrect.

Storage and Nonstorage Blocks

In this section...
“Storage Blocks” on page 16-37
“Nonstorage Blocks” on page 16-37

For the significance of the distinction between storage and nonstorage blocks, see “Interleaving of Block Operations” on page 16-25.

Storage Blocks

These blocks are capable of holding an entity for a nonzero duration:

- Blocks in Queues library (However, these can act like nonstorage blocks in some circumstances; see the note below.)
- Blocks in Servers library
- Blocks in Entity Generators library
- Output Switch block with the **Store entity before switching** option selected

Note In the special case of an entity arriving at an empty queue whose entity output port is not blocked, the queue acts like a nonstorage block in that block operations are deferred until the entity’s arrival at a storage block subsequent to the queue.

Nonstorage Blocks

These blocks permit an entity arrival but must output or destroy the entity at the same value of the simulation clock:

- Blocks in Attributes library
- Blocks in Routing library, except the Output Switch block with the **Store entity before switching** option selected
- Blocks in Gates library

- Blocks in the Entity Management library
- Blocks in Timing library
- Blocks in Probes library
- Blocks in SimEvents User-Defined Functions library
- Attribute Scope, X-Y Attribute Scope, and Instantaneous Entity Counting Scope blocks
- Entity Sink block
- Conn block
- Entity Departure Event to Function-Call Event block
- Entity-Based Function-Call Event Generator block

Examples

Use this list to find examples in the documentation.

Attributes of Entities

- “Role of Attributes in SimEvents Models” on page 1-7
- “Example: Setting Attributes” on page 1-10
- “Example: Varying Attribute Values Between Runs Using Rapid Simulation” on page 15-37

Counting Entities

- “Example: Counting Simultaneous Departures from a Server” on page 1-21
- “Example: Resetting a Counter After a Transient Period” on page 1-23

Working with Events

- “Example: Comparing Types of Signal-Based Events” on page 2-5
- “Example: Observing Service Completions” on page 2-22
- “Example: Detecting Collisions by Comparing Events” on page 2-25
- “Example: Opening a Gate Upon Random Events” on page 2-30
- “Example: Counting Events from Multiple Sources” on page 2-35
- “Example: Choices of Values for Event Priorities” on page 3-11
- “Example: Effects of Specifying Event Priorities” on page 3-26

Queuing Systems

- “Example: Event Calendar Usage for a Queue-Server Model” on page 2-10
- “Example: Waiting Time in LIFO Queue” on page 5-2
- “Example: Serving Preferred Customers First” on page 5-8
- “Example: Preemption by High-Priority Entities” on page 5-12
- “Example: M/M/5 Queuing System” on page 5-18
- “Example: Using Servers in Shifts” on page 7-11
- “Example: Varying Queue Capacity Between Runs Using Rapid Simulation” on page 15-43

Working with Signals

“Example: Creating a Random Signal for Switching” on page 4-5

“Example: Resampling a Signal Based on Events” on page 4-13

“Example: Sending Queue Length to the Workspace” on page 4-15

Server States

“Example: Failure and Repair of a Server” on page 5-22

“Example: Adding a Warmup Phase” on page 5-24

Routing Entities

“Example: Cascaded Switches with Skewed Distribution” on page 6-6

“Example: Compound Switching Logic” on page 6-7

“Example: Choosing the Shortest Queue” on page 7-3

Batching

“Example: Varying Fluid Flow Rate Based on Batching Logic” on page 7-6

Gates

“Example: Controlling Joint Availability of Two Servers” on page 8-4

“Example: Synchronizing Service Start Times with the Clock” on page 8-6

“Example: Opening a Gate Upon Entity Departures” on page 8-7

“Example: First Entity as a Special Case” on page 8-11

Timeouts

- “Basic Example Using Timeouts” on page 9-3
- “Defining Entity Paths on Which Timeouts Apply” on page 9-7
- “Example: Dropped and Timed-Out Packets” on page 9-11
- “Example: Rerouting Timed-Out Entities to Expedite Handling” on page 9-12
- “Example: Limiting the Time Until Service Completion” on page 9-14

Discrete Event Subsystems

- “Example: Comparing the Lengths of Two Queues” on page 10-18
- “Example: Normalizing a Statistic to Use for Routing” on page 10-19
- “Example: Ending the Simulation Upon an Event” on page 10-21
- “Example: Sending Unrepeated Data to the MATLAB Workspace” on page 10-22
- “Example: Focusing on Events, Not Values” on page 10-23
- “Example: Detecting Changes from Empty to Nonempty” on page 10-24
- “Example: Logging Data About the First Entity on a Path” on page 10-25
- “Example: Using Entity-Based Timing for Choosing a Port” on page 10-30
- “Example: Performing a Computation on Selected Entity Paths” on page 10-31

Troubleshooting

- “Example: Plotting Entity Departures to Verify Timing” on page 11-11
- “Example: Plotting Event Counts to Check for Simultaneity” on page 11-15
- “Example: Time-Based Addition of Event-Based Signals” on page 14-75
- “Example: Intergeneration Time of Zero at Simulation Start” on page 14-80
- “Example: Absence of Sample Time Hit at Simulation Start” on page 14-81
- “Example: Faulty Logic in Feedback Loop” on page 14-82
- “Example: Deadlock Resulting from Loop in Entity Path” on page 14-83
- “Example: Invalid Connection of Event-Based Random Number Generator” on page 14-86
- “Example: Sequence of Departures and Statistical Updates” on page 16-26

“Example: Using a #n Signal as a Trigger” on page 16-32

Statistics

“Example: Fraction of Dropped Messages” on page 12-9

“Example: Computing a Time Average of a Signal” on page 12-11

“Example: Resetting an Average Periodically” on page 12-13

“Example: Computing an Ensemble Average Using MATLAB Code” on page 15-26

“Example: Varying the Number of Servers Using MATLAB Code” on page 15-29

“Example: Computing an Ensemble Average Using Rapid Simulation” on page 15-32

Timers

“Basic Example Using Timer Blocks” on page 12-21

“Timing Multiple Entity Paths with One Timer” on page 12-23

“Restarting a Timer from Zero” on page 12-24

“Timing Multiple Processes Independently” on page 12-26

Rapid Simulation

“Example: Computing an Ensemble Average Using Rapid Simulation” on page 15-32

“Example: Varying Attribute Values Between Runs Using Rapid Simulation” on page 15-37

“Example: Varying Queue Capacity Between Runs Using Rapid Simulation” on page 15-43

A

- arbitrary event sequence 3-9
 - troubleshooting 14-73
- ARQ
 - Stateflow charts 13-4
- attributes of entities
 - combining 1-25
 - Embedded MATLAB functions 1-13
 - permitted values 1-9
 - plots 11-2
 - reading values 1-18
 - setting values 1-7
 - subsystem for manipulating 1-15
 - usage 1-7
- autoscaling 11-8
- averaging signals
 - over samples 12-13
 - over time 12-11
- axis limits 11-5

B

- block breakpoints 14-56
- block identifiers 14-18
 - obtaining 14-38
- block-to-block interactions 16-18
- breakpoint identifiers 14-18
- breakpoints
 - clearing (removing) 14-63
 - disabling 14-63
 - discrete-event simulation 14-56
 - enabling 14-64
 - listing 14-60
 - running simulation until 14-62

C

- caching 11-6
- cancellation messages in debugger 14-13
- cascading switch blocks

- random 6-6
- combining entities 1-25
- combining events 2-34
- conditional events 2-39
- copying entities 1-33
- counter reset events 2-2
- counting entities
 - cumulative 1-20
 - instantaneous 1-20
 - reset 1-22
 - storing in attribute 1-24
- counting events 11-2
 - simultaneity 11-15

D

- data history 11-6
- data types 4-3
- debugger
 - discrete-event simulations 14-3
 - identifiers 14-18
 - indentation 14-20
 - sedb package 14-7
 - sedebg prompt 14-7
 - simulation log 14-8
 - starting SimEvents 14-5
 - startup commands 14-54
 - stepping in discrete-event simulation 14-26
 - stopping 14-24
- debugging
 - discrete-event simulations 14-2
- delayed restart events 2-2
- delays
 - signal updates 14-90
- dependent operations 14-20
- detail settings 14-47
- detection messages in debugger 14-14
- Discrete Event Subsystem block
 - building subsystems 10-11
- discrete event subsystem execution events 2-2

- discrete event subsystems 10-7
 - blocks inside 10-10
 - building
 - entity departures 10-29
 - function calls 10-34
 - signal-based events 10-11
 - combinations of events 10-33
 - entity departures 10-27
 - events in terminated signals 10-23
 - function calls 10-33
 - multiple inputs 10-15
 - need for 10-2
 - sequence of events 10-8
 - signal-based events 10-14
- discrete state plots 11-4
- discrete-event plots 11-4
 - compared to time-based plots 11-17
 - customizing 11-8
 - exporting 11-9
 - troubleshooting using 11-10
- dropped messages 12-9
 - timeouts 9-11
- E**
- Embedded MATLAB functions
 - attributes of entities 1-13
- Embedded MATLAB® Function blocks 7-3
- enabled gates 8-4
- entities
 - combining 1-25
 - counting 1-20
 - event-based generation 1-2
 - replicating 1-33
 - synchronizing 1-25
 - timeouts 9-1
- entity advancement events 2-2
- entity collisions 2-25
- entity data
 - combining 1-25
 - Embedded MATLAB functions 1-13
 - permitted values 1-9
 - plots 11-2
 - reading values 1-18
 - setting values 1-7
 - subsystem for manipulating 1-15
 - usage 1-7
- entity destruction events 2-2
- entity generation
 - event-based 1-2
 - vector of times 1-5
- entity generation events 2-2
- entity identifiers 14-18
 - obtaining 14-38
- entity messages in debugger 14-16
- entity paths
 - timeouts 9-7
- entity request events 2-2
- entity-departure subsystems 10-27
- equal event priorities 3-9
 - troubleshooting 14-73
- event breakpoints 14-56
- event calendar 16-2
 - displaying 14-46
 - displaying in simulation log 14-10
 - events on 16-3
 - events on/off 3-26
 - example 2-10
 - numerical/system-level priority 3-26
- event identifiers 14-18
 - obtaining 14-38
- event-based sequences 4-9
- event-based signals
 - data sets 4-9
 - deferring reactions 16-10
 - description 4-2
 - feedback loops 14-80
 - initial values 4-12
 - latency 14-76
 - manipulating 4-12

- MATLAB® workspace 4-15
- random 4-4
- resampling 4-13
- resolving updates 16-7
- troubleshooting 14-74
- unrepeated values to workspace 10-22
- update sequence 16-34

events

- conditionalizing 2-39
- generating 2-28
- manipulating 2-32
- numerical/system-level priority 3-26
- observing 2-18
- on/off event calendar 3-26
- priorities 3-8
- reacting to signal updates 16-10
- resolving signal updates 16-7
- sequence 16-2
 - modeling approaches 3-7
- simultaneous 3-2
- supported types 2-2
- translating 2-37
- troubleshooting 14-73
- union 2-34

execution messages in debugger 14-13

F

- failure modeling
 - conditional events 2-40
 - gates 5-20
 - Stateflow 5-21
- feedback entity paths
 - troubleshooting 14-83
- feedback loops
 - troubleshooting 14-80
- first-order-hold plots 11-4
- function call events
 - discrete-event simulation 2-2
- function calls 2-8

- generating 2-28
- function-call subsystems
 - discrete event 10-33

G

- gate events 2-2
- gates 8-1
 - combinations 8-9
 - enabled 8-4
 - entity departures 8-7
 - release 8-6
 - role in modeling 8-2
 - types 8-3

I

- identifiers in debugger 14-18
 - obtaining 14-38
- indentation
 - simulation log 14-20
- independent operations 14-20
- independent replications 12-28
- initial conditions 4-12
 - feedback loops 14-80
- initial port selection
 - switching based on signal 6-2
- initial seeds 12-28
- input signals
 - deferring reactions to updates 16-10
 - resolving updates 16-7
- inspecting states
 - blocks 14-36
 - current point 14-31
 - entities 14-34
 - events 14-38
- instantaneous gate openings 8-6
- intergeneration times
 - event generation 2-30
- interleaved operations 16-25

L

- latency
 - interleaved operations 16-25
 - signal updates 14-90
 - switching based on signal 6-2
 - troubleshooting 14-76
- LIFO queues 5-2
- livelock prevention 16-16
- logic
 - block diagrams 7-10
 - code 7-3
 - usage in discrete-event simulations 7-2
- loops in entity paths 14-83

M

- M/M/5 queuing systems 5-18
- maximum number of events 16-16
- memory read events 2-2
- memory write events 2-2
- monitoring block messages in debugger 14-17
- monitoring ports 16-22

N

- n-servers 5-18
- nonstorage blocks 16-37
- notifying ports 16-21

O

- Output Switch block
 - signal-based routing 6-2

P

- plots 11-1
 - customizing 11-8
 - troubleshooting using 11-10
 - zero-duration values 4-20
- port selection events 2-2

ports

- monitoring 16-22
 - notifying 16-21
 - reactive 16-23
- preemption events 2-2
- preemption in servers 5-11
- priorities, entity
 - priority queues with preemptive servers 5-12
 - queue sequence 5-5
 - server preemption 5-11
- priorities, event 3-8
 - reacting to signal updates 16-10
 - resolving signal updates 16-7
 - troubleshooting 14-73

Q

- queues
 - choosing shortest using logic blocks 7-14
 - choosing shortest using MATLAB code 7-3
 - LIFO vs. FIFO 5-2
 - preemptive servers 5-12
 - priority 5-5
- queuing systems
 - M/M/5 5-18

R

- race conditions 3-11
- random
 - signals 4-4
- random event sequence 3-9
 - troubleshooting 14-73
- random numbers
 - event-based 4-4
 - switch selection 4-5
 - time-based 4-6
- reactive ports 16-23
- release events 2-2
- release gates 8-6

- reneging in queuing 9-3
- repeating simulations 15-13
- replication of entities 1-33
- replications 12-28
- resampling signals 4-13
- resetting averages 12-13
- residual service time 5-12
- resolving simultaneous updates of signals 16-7
- running simulations
 - repeatedly 15-13
 - varying parameters 15-29

S

- sample means 12-13
- sample time 4-2
- sample time hit events 2-2
- scatter plots 11-4
- scheduling messages in debugger 14-13
- scope blocks 11-1
 - zero-duration values 4-20
- sedb package 14-7
- sedebg prompt 14-7
- seed of random number generator
 - varying results 12-28
- server states 5-20
- servers
 - failure states 5-20
 - multiple 5-18
 - preemption 5-11
- service completion events 2-2
- signal-based events
 - comparison 2-5
 - definition 2-4
- signals
 - deferring reactions to updates 16-10
 - event-based 4-2
 - event-based data 4-9
 - random 4-4
 - resolving updates 16-7

- simulation log 14-8
 - filtering 14-47
 - indentation 14-20
- simulation parameters
 - varying in repeated runs 15-29
- simultaneous events 3-2
 - discrete event subsystems 10-8
 - event priorities 3-8
 - input signal updates 16-7
 - interleaved operations 16-25
 - modeling approaches 3-7
 - numerical/system-level priority 3-26
 - on/off event calendar 3-26
 - output signal updates 4-19
 - sequential processing 16-2
 - troubleshooting 14-73
 - unexpected 14-72
- splitting entities 1-25
- stack 5-2
- stairstep plots 11-4
- state inspection
 - blocks 14-36
 - current point 14-31
 - entities 14-34
 - events 14-38
- Stateflow and SimEvents® 5-21
- Stateflow® 13-3
- statistics 12-2
 - accessing from blocks 12-5
 - custom 12-8
 - discrete event subsystems 10-3
 - interleaved updates 16-25
 - latency 14-90
- Statistics tab 12-5
- stem plots 11-4
- stepping in discrete-event simulation 14-26
- stop time 12-35
 - event-based timing 10-2
- storage blocks 16-37
 - changing processing sequence 16-27

- looped entity paths 14-83
- storage completion events 2-2
- switching entity paths
 - based on signal 6-2
 - initial port selection 6-2
 - random with cascaded blocks 6-6
 - repeating sequence 10-4
 - storing entity 6-2
- synchronizing entities 8-6

T

- time averages 12-11
- time-based blocks
 - zero-duration values 16-31
- timed breakpoints 14-56
- timed-out entities 9-10
 - routing 9-12
- timeout events 2-2
- timeout intervals 9-4
- timeout paths 9-7
- timeout tags 9-4
- timeouts of entities 9-1
 - role in modeling 9-2
- timer tags 12-22
- timers 12-20
 - combining 1-27
 - independent 12-26

- multiple entity paths 12-23
- restarting 12-24
- trigger events
 - discrete-event simulation 2-2
- triggers
 - zero-duration values 16-32

V

- value change events 2-2
- visualization 11-1
 - zero-duration values 4-20

W

- workspace 4-15
 - unrepeated signal values 10-22

Z

- zero-duration values
 - definition 4-19
 - MATLAB workspace 4-22
 - time-based blocks 16-31
 - visualization 4-20
- zero-order hold
 - plots 11-4