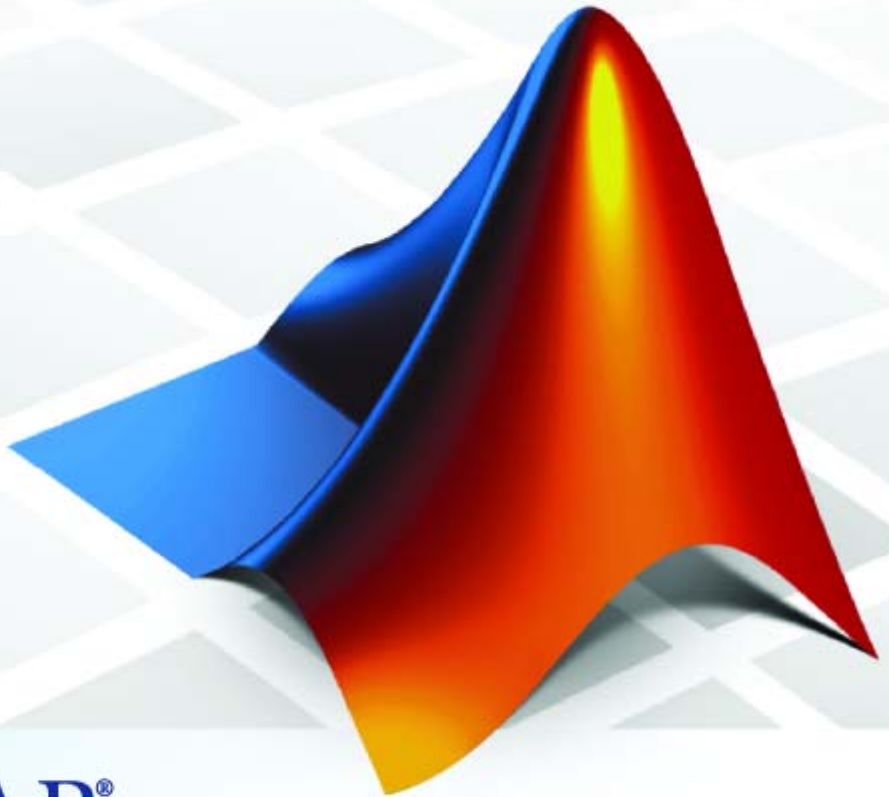


SimEvents™ 2

User's Guide



MATLAB®
& **SIMULINK®**

How to Contact The MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
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 2005–2007 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

Other product or brand names are 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
March 2006 Online only
September 2006 Online only
March 2007 Online only

New for Version 1.0 (Release 14SP3+)
Revised for Version 1.1 (Release 2006a)
Revised for Version 1.2 (Release 2006b)
Revised for Version 2.0 (Release 2007a)

Working with Entities

1

Generating Entities When Events Occur	1-2
Detecting Sample Time Hits	1-2
Detecting Changes in Signal Values	1-4
Detecting Edges in Trigger Signals	1-5
Detecting Function Calls	1-7
Using Generation Times from a Vector	1-11
Configuring the Block to Generate Entities at Specified Times	1-11
Sample Use Cases	1-12
Setting Attributes of Entities	1-13
Example: Setting Attributes	1-14
When to Use Attributes	1-16
Accessing Attributes of Entities	1-19
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	1-23
Combining Entities	1-24
Overview of the Entity-Combining Operation	1-24
Example: Waiting to Combine Entities	1-24
Example: Copying Timers When Combining Entities	1-26
Example: Managing Data in Composite Entities	1-27
Replicating Entities on Multiple Paths	1-30
Departure Port Precedence	1-30

Supported Events in SimEvents Models	2-2
Types of Supported Events	2-2
Signal-Based Events	2-3
Function Calls	2-7
Event Processing in SimEvents	2-9
Role of the Event Calendar	2-9
Processing Sequence for Simultaneous Events	2-11
Livelock Detection	2-12
Working with Simultaneous Events	2-14
Choosing an Approach for Simultaneous Events	2-14
Setting Event Priorities	2-15
Example: Event Calendar for a Queue-Server Model	2-17
Example: Race Conditions at a Switch	2-25
Events On and Off the Event Calendar	2-31
Observing Events	2-36
Example: Observing Service Completions	2-38
Example: Detecting Collisions by Comparing Events	2-40
Generating Function-Call Events	2-43
Generating Events When Other Events Occur	2-43
Generating Events Using Intergeneration Times	2-45
Manipulating Events	2-46
Blocks for Manipulating Events	2-47
Creating a Union of Multiple Events	2-47
Translating Events to Control the Processing Sequence ..	2-50
Conditionalizing Events	2-52

Role of Event-Based Signals in SimEvents Models	3-2
Comparison with Time-Based Signals	3-2
Generating Random Signals	3-4
Generating Random Event-Based Signals	3-4
Examples of Random Event-Based Signals	3-5
Generating Random Time-Based Signals	3-6
Using Data Sets to Create Event-Based Signals	3-9
Generating Sequences Based on Arbitrary Events	3-9
Understanding the Resolution Sequence for Input	
Signals	3-11
Detection of Signal Updates	3-11
Effect of Simultaneous Operations	3-12
Resolving the Set of Operations	3-13
Using Event Priorities to Resolve Simultaneous Signal	
Updates	3-13
Resolving Simultaneous Signal Updates Without Using	
Event Priorities	3-15
Choosing How to Resolve Simultaneous Signal	
Updates	3-17
Update Sequence for Output Signals	3-18
Example: Detecting Changes in the Last-Updated	
Signal	3-18
Multiple Simultaneous Updates of an Output Signal ..	3-21
Zero-Duration Values of Signals	3-21
Importance of Zero-Duration Values	3-22
Detecting Zero-Duration Values	3-22
Latency in Signal Updates	3-25
Manipulating Signals	3-27

Specifying Initial Conditions for Event-Based Signals	3-27
Example: Resampling a Signal Based on Events	3-28
Sending Data to the MATLAB Workspace	3-31
Example: Sending Queue Length to the Workspace	3-31
Using the To Workspace Block with Event-Based Signals	3-34

Modeling Queues and Servers

4

Using a LIFO Queuing Discipline	4-2
Example: Waiting Time in LIFO Queue	4-2
Sorting by Priority	4-4
Example: Serving Preferred Customers First	4-7
Preempting an Entity in a Server	4-10
Criteria for Preemption	4-10
Residual Service Time	4-10
Queuing Disciplines for Preemptive Servers	4-11
Example: Preemption by High-Priority Entities	4-11
Modeling Multiple Servers	4-13
Example: M/M/5 Queuing System	4-13
Modeling the Failure of a Server	4-15
Server States	4-15
Using a Gate to Implement a Failure State	4-15
Using Stateflow to Implement a Failure State	4-16

Routing Techniques

5

Output Switching Based on a Signal	5-2
---	------------

Specifying an Initial Port Selection	5-2
Using the Storage Option to Prevent Latency Problems ..	5-2
Example: Cascaded Switches with Skewed Distribution	5-6
Example: Compound Switching Logic	5-7

Using Logic

6

Role of Logic in SimEvents Models	6-2
Using Embedded MATLAB Function Blocks for Logic	6-3
Example: Choosing the Shortest Queue	6-3
Example: Varying Fluid Flow Rate Based on Batching Logic	6-6
Using Logic Blocks	6-10
Example: Using Servers in Shifts	6-11
Example: Choosing the Shortest Queue Using Logic Blocks	6-16

Regulating Arrivals Using Gates

7

Role of Gates in SimEvents Models	7-2
Accessing Gate Blocks	7-3
Types of Gates	7-3
Keeping a Gate Open Over a Time Interval	7-4
Example: Controlling Joint Availability of Two Servers ..	7-4

Opening a Gate Instantaneously	7-6
Example: Synchronizing Service Start Times with the Clock	7-6
Example: Opening a Gate Upon Entity Departures	7-7
Using Logical Combinations of Gates	7-9
Example: First Entity as a Special Case	7-10

Forcing Departures Using Timeouts

8

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

Timing Issues in SimEvents Models	9-2
Timing for the End of the Simulation	9-2
Timing for a Statistical Computation	9-3
Timing for Choosing a Port Using a Sequence	9-4
Role of Discrete Event Subsystems in SimEvents	
Models	9-7
Purpose of Discrete Event Subsystems	9-8
Processing Sequence for Events in Discrete Event Subsystems	9-8
Blocks Inside Discrete Event Subsystems	9-10
Working with Discrete Event Subsystem Blocks	9-11
Setting Up Signal-Based Discrete Event Subsystems	9-11
Signal-Based Events That Control Discrete Event Subsystems	9-14
Examples Using Discrete Event Subsystem Blocks	9-17
Example: Comparing the Lengths of Two Queues	9-17
Example: Normalizing a Statistic to Use for Routing	9-18
Example: Using Event-Based Timing for a Statistical Computation	9-20
Example: Ending the Simulation Upon an Event	9-21
Example: Sending Unrepeated Data to the MATLAB Workspace	9-22
Example: Focusing on Events, Not Values	9-23
Example: Detecting Changes from Empty to Nonempty ..	9-24
Example: Logging Data About the First Entity on a Path	9-25
Creating Entity-Departure Subsystems	9-27
Accessing Blocks for Entity-Departure Subsystems	9-28
Setting Up Entity-Departure Subsystems	9-29
Examples Using Entity-Departure Subsystems	9-30

Example: Using Entity-Based Timing for Choosing a Port	9-30
Example: Performing a Computation on Selected Entity Paths	9-32
Using Function-Call Subsystems	9-33
Use Cases for Function-Call Subsystems	9-33
Setting Up Function-Call Subsystems in SimEvents Models	9-34

Plotting Data

10

Choosing and Configuring Plotting Blocks	10-2
Sources of Data for Plotting	10-2
Inserting and Connecting Scope Blocks	10-3
Connections Among Points in Plots	10-4
Varying Axis Limits Automatically	10-5
Caching Data in Scopes	10-6
Examples Using Scope Blocks	10-6
 Features of Plot Window	 10-8
 Using Plots for Troubleshooting	 10-9
 Example: Plotting Entity Departures to Verify Timing	 10-10
Model Exhibiting Correct Timing	10-10
Model Exhibiting Latency	10-11
 Example: Plotting Event Counts to Check for Simultaneity	 10-14
 Comparison with Time-Based Plotting Tools	 10-16

Role of Statistics in Discrete-Event Simulation	11-2
Statistics for Data Analysis	11-2
Statistics for Run-Time Control	11-3
Accessing Statistics from SimEvents Blocks	11-4
Accessing Statistics Throughout the Simulation	11-4
Accessing Statistics When Stopping or Pausing Simulation	11-6
Deriving Custom Statistics	11-7
Graphical Block-Diagram Approach	11-7
Coded Approach	11-8
Post-Simulation Analysis	11-8
Example: Fraction of Dropped Messages	11-8
Example: Computing a Time Average of a Signal	11-10
Example: Resetting an Average Periodically	11-12
Using Timers	11-19
Basic Example Using Timer Blocks	11-19
Basic Procedure for Using Timer Blocks	11-21
Timing Multiple Entity Paths with One Timer	11-21
Restarting a Timer from Zero	11-23
Timing Multiple Processes Independently	11-24
Running a Series of Simulations	11-26
Creating Independent Replications	11-26
Running Simulations from MATLAB	11-28
Regulating the Simulation Length	11-33

Using Stateflow with SimEvents

Role of Stateflow in SimEvents Models	12-2
--	-------------

Guidelines for Using Stateflow and SimEvents Blocks	12-3
Examples Using Stateflow and SimEvents Blocks	12-5
Failure State of Server	12-5
Go-Back-N ARQ Model	12-5

Troubleshooting Discrete-Event Simulations

13

Viewing the Event Calendar	13-2
Turning Event Logging On	13-2
Logging the Processing of Events	13-3
Logging the Scheduling of Events	13-4
Logging the List of Events	13-5
Example: Event Logging	13-6
Viewing Entity Locations	13-9
Turning Entity Logging On	13-9
Interpreting Entity Logging Messages	13-9
Example: Entity Logging	13-10
Common Problems in SimEvents Models	13-13
Unexpectedly Simultaneous Events	13-13
Unexpectedly Nonsimultaneous Events	13-14
Unexpected Processing Sequence for Simultaneous Events	13-14
Time-Based Block Not Recognizing Certain Trigger Edges	13-15
Incorrect Timing of Signals	13-15
Unexpected Use of Old Value of Signal	13-17
Effect of Initial Condition on Signal Loops	13-21
Loops in Entity Paths Without Storage Blocks	13-23
Unexpected Timing of Random Signal	13-24
Unexpected Correlation of Random Processes	13-26
Configuration Parameters for SimEvents Models	13-28

Notifications and Queries Among Blocks 14-2

 Querying Whether a Subsequent Block Can Accept an Entity 14-2

 Notifying Blocks About Status Changes 14-3

Notifying, Monitoring, and Reactive Ports 14-4

 Notifying Ports 14-4

 Monitoring Ports 14-5

 Reactive Ports 14-6

Interleaving of Block Operations 14-8

 How Interleaving of Block Operations Occurs 14-8

 Storage and Nonstorage Blocks 14-9

 Example: Sequence of Departures and Statistical Updates 14-10

 Example: Using the Event Calendar to Prevent Interleaving 14-14

Zero-Duration Values and Time-Based Blocks 14-17

 Example: Using a #n Signal as a Trigger 14-18

Examples

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

Index

Working with Entities

Generating Entities When Events Occur (p. 1-2)

Using events to determine when to generate an entity

Using Generation Times from a Vector (p. 1-11)

Generating entities at explicit values of time

Setting Attributes of Entities (p. 1-13)

Attaching data to entities

Accessing Attributes of Entities (p. 1-19)

Reading and using data attached to entities

Counting Entities (p. 1-20)

Counting entities per time instant and across the simulation

Combining Entities (p. 1-24)

Synchronizing, combining, and splitting entities

Replicating Entities on Multiple Paths (p. 1-30)

Creating copies of entities

Generating Entities When Events Occur

The Event-Based Entity Generator block enables you to generate entities in response to events that occur during the simulation. Event times and the time intervals between pairs of successive entities are not necessarily predictable in advance. This section describes the events that can cause entity generation, in these topics:

- “Detecting Sample Time Hits” on page 1-2
- “Detecting Changes in Signal Values” on page 1-4
- “Detecting Edges in Trigger Signals” on page 1-5
- “Detecting Function Calls” on page 1-7

Generating entities when events occur might be appropriate if you want the dynamics of your model to determine when to generate entities. For example, if you want to generate an entity every time a Stateflow® chart transitions from state A to state B, then you configure the Stateflow chart to output a function call upon such a transition and configure the Event-Based Entity Generator block to react to each function call by generating an entity. As another example, if you want to generate an entity every time the length of a queue changes, then you configure the queue to output a signal indicating the queue length and configure the Event-Based Entity Generator block to react to changes in that signal’s value by generating an entity.

Note To specify intergeneration times between pairs of successive entities, use the Time-Based Entity Generator block as described in “Creating Entities Using Intergeneration Times” in the getting started documentation.

Detecting Sample Time Hits

You can configure the Event-Based Entity Generator block so that it generates entities in response to updates in a signal. More explicitly, whenever the block producing that signal recomputes and outputs the signal value, the Event-Based Entity Generator block generates an entity. The actual value of the signal and the question of whether recomputing the value yields a different result compared to the previous time step are irrelevant to the entity generation process; only the time of the output is relevant.

Configuring the Block to Detect Sample Time Hits

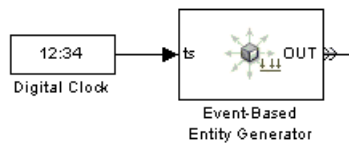
To use this method for generating entities, set the Event-Based Entity Generator block's **Generate entities upon** parameter to **Sample time hit** from port **ts**. This causes the block to have an input signal port labeled **ts**. During the simulation, the Event-Based Entity Generator block generates entities in response to updates in the signal connected to this **ts** port.

Sample Use Cases

Here are a few scenarios that illustrate this method for generating entities:

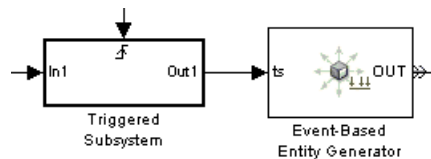
- The **ts** signal is the output of a block with an explicit **Sample time** parameter. Regardless of the value of the **ts** signal, the Event-Based Entity Generator block generates an entity periodically, according to the sample time of the driving block.

Compared to using a Time-Based Entity Generator block with **Distribution** set to **Constant**, this event-based approach is a more direct way to synchronize entity generation events with sample time hits and avoid possible roundoff errors. Below is an example.



For other examples that use this entity-generation method to effect desired simultaneity of events, see “Example: Race Conditions at a Switch” on page 2-25.

- The **ts** signal is the output of a triggered subsystem whose **Propagate execution context across subsystem boundary** parameter is selected. Whenever Simulink® calls the subsystem and recomputes the output, the Event-Based Entity Generator block generates an entity.



Note Do not put the Event-Based Entity Generator block inside a triggered subsystem. Like other blocks that possess entity ports, the Event-Based Entity Generator block is not valid inside a triggered subsystem. See also “Detecting Edges in Trigger Signals” on page 1-5.

- The **ts** signal is a statistical output signal from a SimEvents™ block. Whenever the block recomputes and outputs the statistic, the Event-Based Entity Generator block generates an entity.

Detecting Changes in Signal Values

You can configure the Event-Based Entity Generator block so that it generates entities in response to numerical changes in a signal. This can be useful if the changes in the signal’s value have some significance in your simulation; for example, a signal representing the length of a queue changes whenever an entity arrives at or departs from the queue.

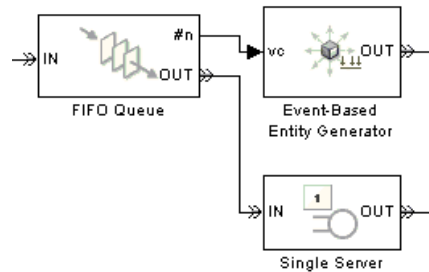
Configuring the Block to Detect Value Changes

To use this method for generating entities, set the Event-Based Entity Generator block’s **Generate entities upon** parameter to Change in signal from port **vc**. This causes the block to have an input signal port labeled **vc**. Also, set the **Type of value change** parameter to indicate whether the block should generate an entity whenever the signal connected to this **vc** port increases, decreases, or exhibits either type of change.

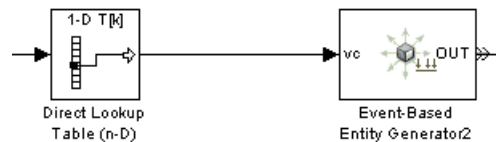
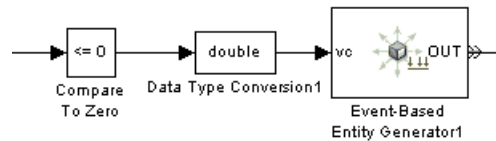
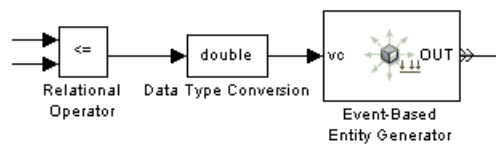
Sample Use Cases

Typically, the **vc** signal is one that you expect to change values at only a discrete set of times during the simulation. In some applications, the signal is discrete valued. Here are a few scenarios that illustrate this method for generating entities:

- The **vc** signal is an integer-valued statistical output signal from a SimEvents block. For example, the statistic could be the number of entities in a queue (shown below) or the number of entities that have departed from the block. Whenever the statistic changes values, the Event-Based Entity Generator block generates an entity.



- The **vc** signal is the output of a discrete-valued block, such as Relational Operator, Compare to Zero, or Direct Lookup Table (n-D). Whenever the logical value or the lookup table value changes, the Event-Based Entity Generator block generates an entity.



Detecting Edges in Trigger Signals

You can configure the Event-Based Entity Generator block so that it generates entities in response to rising or falling edges in a signal. This can be useful if the signal's zero crossings have some significance in your simulation; for example, a signal representing acceleration crosses zero whenever the velocity reverses direction. A signal whose rising and falling edges are used to invoke a behavior during the simulation is called a trigger signal.

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).

Configuring the Block to Detect Edges

To use this method for generating entities, set the Event-Based Entity Generator block's **Generate entities upon** parameter to **Trigger** from port **tr**. This causes the block to have an input signal port labeled **tr**. Also, set the **Trigger type** parameter to indicate whether the block should generate an entity whenever the signal connected to this **tr** port has a rising edge, a falling edge, or either type of edge.

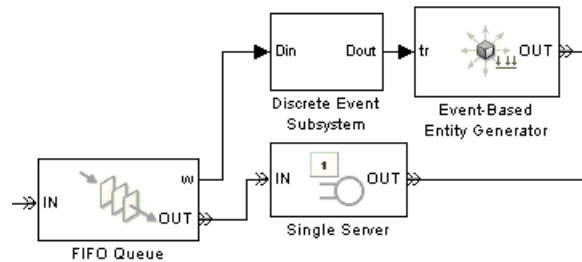
Note Do not put the Event-Based Entity Generator block inside a triggered subsystem, but rather attach the trigger signal directly to the block's **tr** port. Like other blocks that possess entity ports, the Event-Based Entity Generator block is not valid inside a triggered subsystem.

Sample Use Cases

Here are a few scenarios that illustrate this method for generating entities:

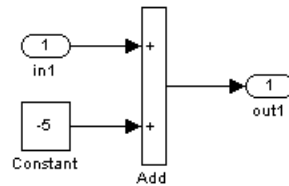
- The **tr** signal arises from the time-driven dynamics of your system. Whenever the signal crosses zero, the Event-Based Entity Generator block generates an entity.
- The **tr** signal is a real-valued statistical output signal from a SimEvents block, plus a negative constant. Whenever the statistic crosses a threshold that is the absolute value of the constant, the sum crosses zero and the Event-Based Entity Generator block generates an entity.

In the figure below, the Event-Based Entity Generator block generates a new entity each time the queue's average waiting time signal crosses the threshold of 5 seconds.



Top-Level Model

The subsystem between the queue and the entity generator, shown below, adds -5 to the average waiting time value to translate the threshold from 5 to 0. To understand why this computation occurs in a subsystem rather than at the top level of the model hierarchy, see “Timing Issues in SimEvents Models” on page 9-2.



Subsystem Contents

Detecting Function Calls

A function-call signal is a special type of signal that directly defines a time instant and whose typical purpose is to call a subsystem or other functional operation at that instant. A function-call signal can come from an Entity Departure Event to Function-Call Event block, a Signal-Based Event to Function-Call Event block, a Function-Call Generator block, or a Stateflow block.

You can configure the Event-Based Entity Generator block so that it generates entities in response to a function call. This can be useful for generating new entities based on the behavior of existing entities in the simulation,

generating multiple new entities simultaneously, or incorporating Stateflow dynamics into your SimEvents model.

Configuring the Block to Detect Function Calls

To use this method for generating entities, set the Event-Based Entity Generator block's **Generate entities upon** parameter to Function call from port **fcn**. This causes the block to have an input signal port labeled **fcn**. During the simulation, the Event-Based Entity Generator block generates entities in response to function calls in the signal connected to this **fcn** port.

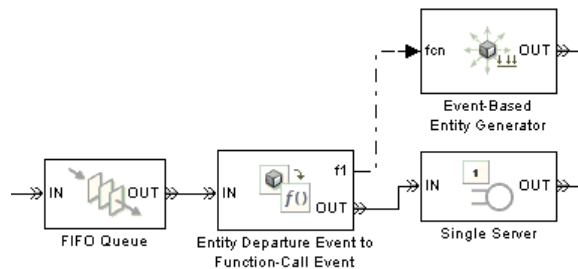
Note Do not put the Event-Based Entity Generator block inside a function-call subsystem, but rather attach the function-call signal directly to the block's **fcn** port. Like other blocks that possess entity ports, the Event-Based Entity Generator block is not valid inside a function-call subsystem.

Sample Use Cases

Here are a few scenarios that illustrate this method for generating entities:

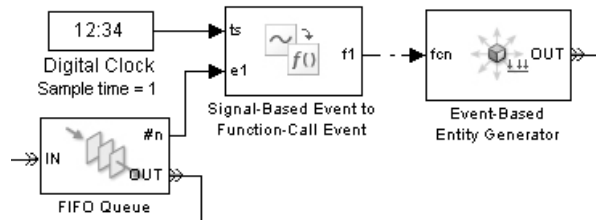
- An Entity Departure Event to Function-Call Event block issues a function call whenever an entity departs from it. Whenever this occurs, the Event-Based Entity Generator block generates an entity.

In the figure below, the Event-Based Entity Generator block generates a new entity each time an entity departs from the queue (or, equivalently, each time an entity arrives at the server).



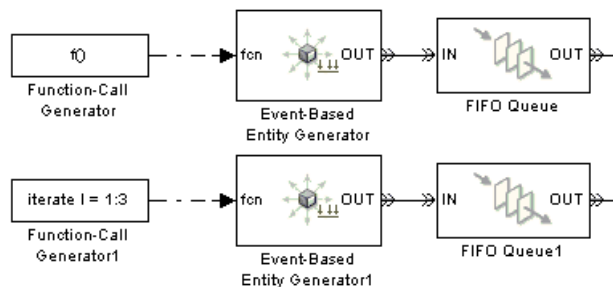
- A Signal-Based Event to Function-Call Event block with **ts** and **e1** input ports issues a function call whenever the **ts** signal is updated while the **e1** signal is positive. During time intervals when **e1** is positive, the Event-Based Entity Generator block can generate entities, where the specific times depend on updates of the **ts** signal. When **e1** is zero or negative, the Event-Based Entity Generator block generates no entities, even if the **ts** signal has an update. The **e1** signal provides a way to enable and disable entity generation.

In the figure below, the Event-Based Entity Generator block generates entities at integer-valued times when the queue is nonempty.



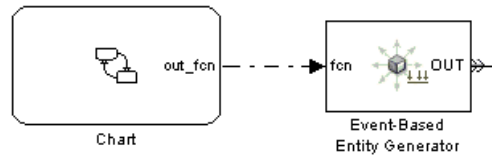
- A Function-Call Generator block issues one or more function calls periodically during the simulation. Each function call causes the Event-Based Entity Generator block to generate an entity.

In the figure below, the two Function-Call Generator blocks share the same sample time of 1 second, but the top block generates one function call at a time, while the bottom one generates three simultaneous function calls at a time. The icon changes to reflect the multiplicity of function calls. As a result, the top Event-Based Entity Generator block generates one entity each second, while the bottom one generates three entities *simultaneously* each second.



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 you might want to insert a queue between the generator and the server so that entities (in particular, the second and third entities) have a place to wait for the server to become available.

- A Stateflow block issues a function-call output whenever the state takes a particular transition, as defined in the chart. Each function call causes the Event-Based Entity Generator block to generate an entity.



Using Generation Times from a Vector

If you have an explicit list of unique times at which you want to generate entities, you can configure the Time-Based Entity Generator block so that it generates entities at these times. To do this, create a vector of intergeneration times. Intergeneration times are the differences between pairs of successive time values in your list.

These topics provide instructions and motivation:

- “Configuring the Block to Generate Entities at Specified Times” on page 1-11
- “Sample Use Cases” on page 1-12

Configuring the Block to Generate Entities at Specified Times

To generate entities at specified times, follow this procedure:

- 1** Set the Time-Based Entity Generator block’s **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 in the Time-Based Entity Generator block.
- 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 by entering the definition in the MATLAB® Command Window, by loading a MAT-file that you previously created, or by manipulating a variable that a To Workspace or Discrete Event Signal to Workspace block previously created.

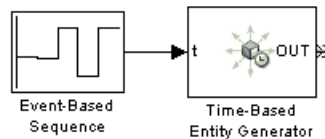
An example of a column vector listing generation times is below.

```
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 dialog box of 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` to halt the generation process if the simulation time exceeds your maximum generation time.

Sample Use Cases

Using explicit entity-generation times might be appropriate if you want to

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

Setting Attributes of Entities

- “Example: Setting Attributes” on page 1-14
- “When to Use Attributes” on page 1-16

You can attach data to an entity using one or more *attributes* of the entity. Each attribute has a name and a numeric value. For example, if your entities represent a message that you are transmitting across a communication network, you might assign data called *length* that indicates the length of each particular message. You can read or change the values of attributes during the simulation.

The Set Attribute block assigns attributes on each arriving entity. Assignments can create new attributes or change the values of existing attributes. Attribute values can come from information you enter in the block’s dialog box or from signals.

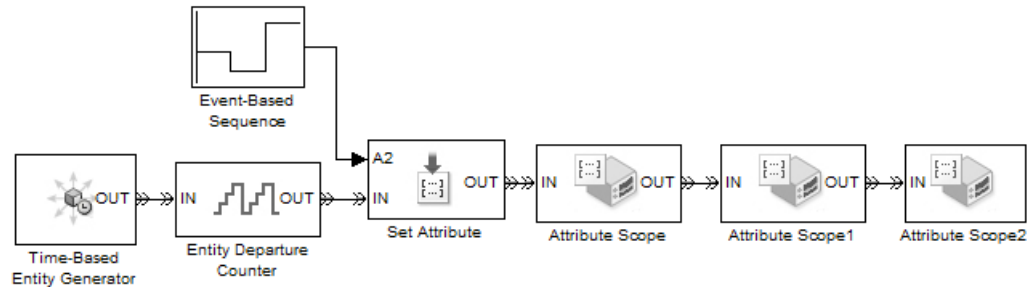
Other blocks can set particular kinds of attributes:

- The Entity Departure Counter block can set an attribute whose value is the entity count.
- The Single Server block can set an attribute whose value is the residual service time for a preempted entity. For more information, see “Preempting an Entity in a Server” on page 4-10.

To learn how to query entities for attribute values, see “Accessing Attributes of Entities” on page 1-19. To learn how to aggregate attributes from distinct entities, see “Combining Entities” on page 1-24.

Example: Setting Attributes

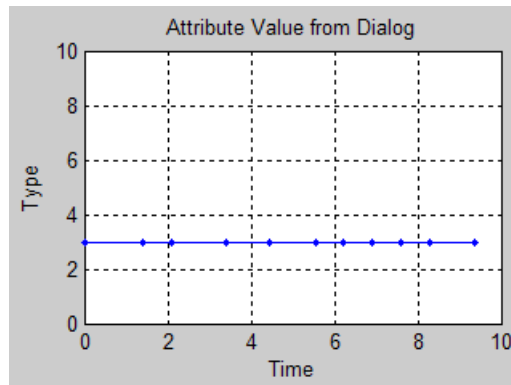
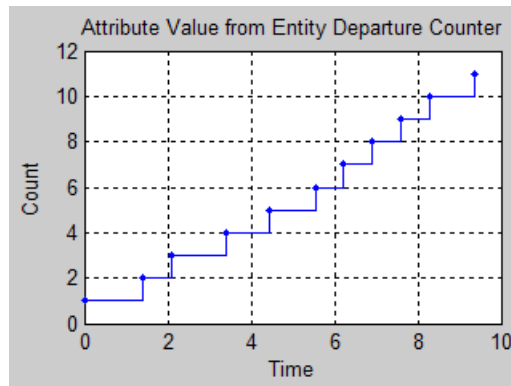
The example below illustrates different ways of assigning attribute values to entities.

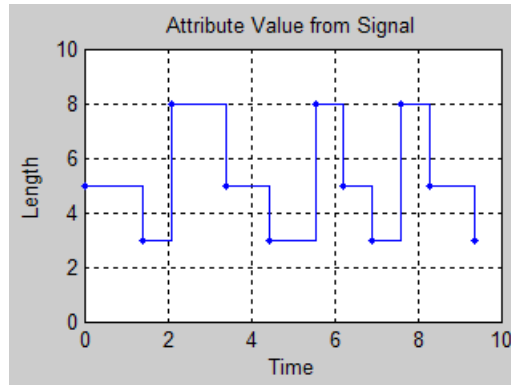


After each entity departs from the Set Attribute block, it possesses the attributes listed in the table.

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 = 0n 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
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

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.





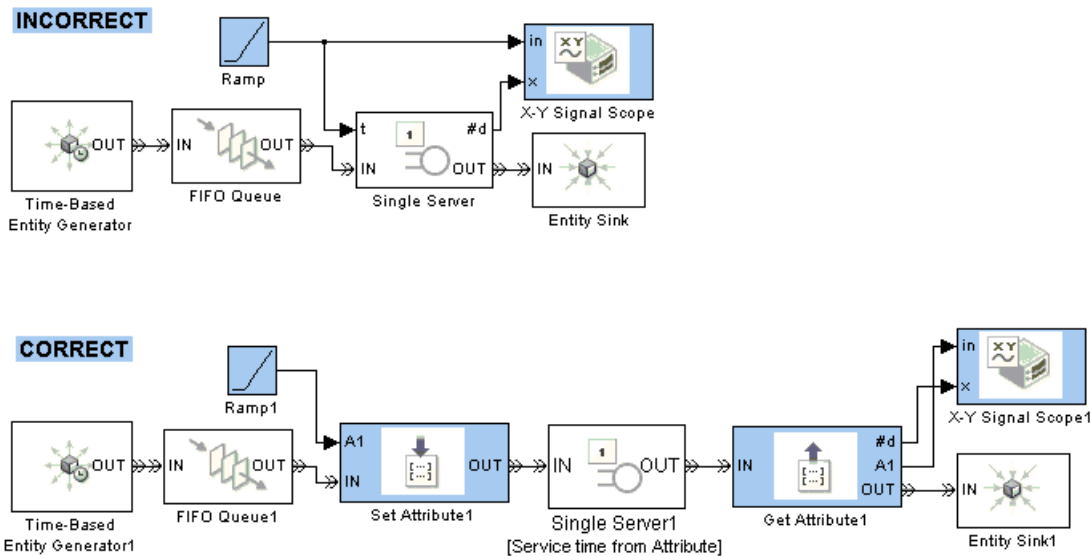
When to Use Attributes

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. This section discusses the importance of considering not only the topology of your block diagrams, but also the timing of data signals appearing in SimEvents models.

Example: Reusing Data

Consider a queue-server example with varying service times, where you want to plot the service time against entity count for each entity that departs 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 because of 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.

A correct way to implement such an example involves attaching the service time to each entity using an attribute and retrieving the attribute value from each entity upon its departure from the server. 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.

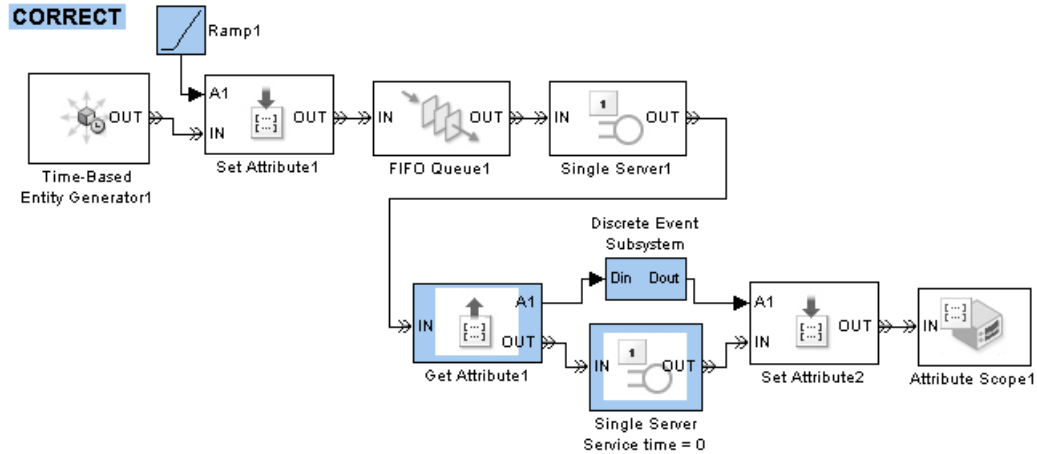
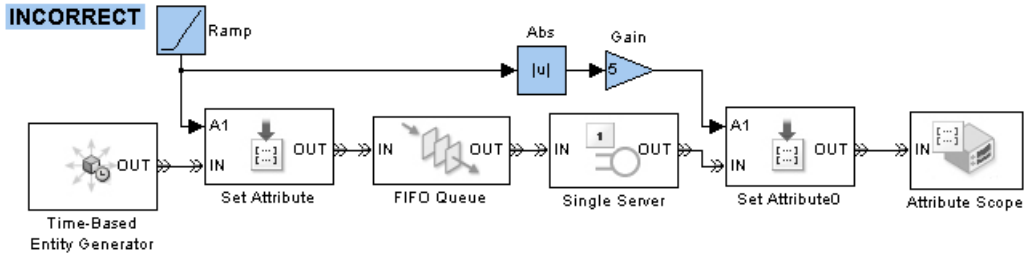


Example: Manipulating Data

To manipulate the value of an attribute that you originally set using a signal, follow these rules:

- Query the value and then manipulate it, instead of creating a branch line from the signal and manipulating that. To query an entity for the value of an attribute, use the Get Attribute block.
- Perform the manipulation in a discrete event subsystem, as described in Chapter 9, “Controlling Timing with Subsystems”. This ensures correct timing of the manipulation of the event-based signal that represents the attribute value.
- Insert a storage block with a delay of zero after the Get Attribute block, if you use the manipulated attribute value in a subsequent block on the same entity path. This ensures that the subsequent block reads the up-to-date results of the manipulation upon the entity’s arrival. For details, see “Interleaving of Block Operations” on page 14-8.

The example below illustrates the use of the Get Attribute block to query an entity for its attribute value, the use of the Discrete Event Subsystem block to contain the manipulation of the attribute value, and the insertion of a Single Server block between the Get Attribute and Set Attribute blocks.



Accessing Attributes of Entities

The section “Setting Attributes of Entities” on page 1-13 described how to use the Set Attribute block to attach data to entities near the time that you generate the entities. You can also use that block to change the value of an attribute at any point in an entity path.

To access data that has been attached to an entity, use one of these methods:

- Send the entity to a Get Attribute block. The Get Attribute block retrieves attributes on each arriving entity and creates signals with the attribute values.

For example, see the subsystem of the model described in “Adding Event-Based Behavior” in the getting started documentation.

- Send the entity to an Attribute Scope block and set the block’s **Y attribute name** parameter to the name of the attribute. Alternatively, send the entity to an X-Y Attribute Scope block and set the block’s **X attribute name** and **Y attribute name** parameters to the names of two attributes.

For example, see Chapter 10, “Plotting Data”.

- Name the attribute in the dialog box of a block that supports the use of attribute values for block parameters. For example, you can use an attribute value to specify the service time in the Single Server block or the selected entity output port in the Output Switch block.

For example, see “Example: Using an Attribute to Select an Output Port” in the getting started documentation.

Tip If your entity possesses an attribute containing a desired service time, switching criterion, or other quantity that a block can obtain from either an attribute or signal, it is usually better to use the attribute directly than to create a signal with the attribute’s value and ensure 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 13-19.

Counting Entities

Counting entities can be useful for statistical measures and for understanding a simulation. This section describes these techniques for counting entities in different ways:

- “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 1-23

For troubleshooting purposes, see also “Viewing Entity Locations” on page 13-9.

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 and when their 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 getting started documentation
- “Example: First Entity as a Special Case” on page 7-10
- “Stopping Based on Entity Count” on page 11-34

Counting Departures per Time Instant

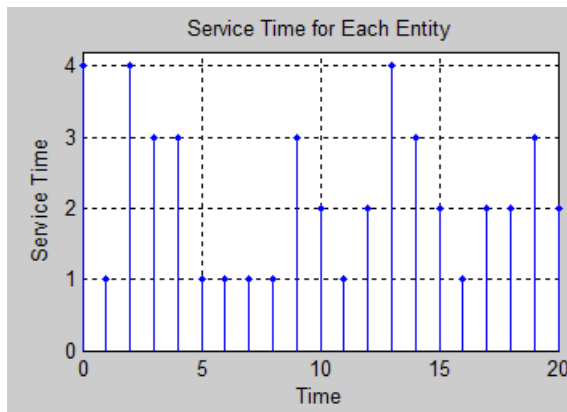
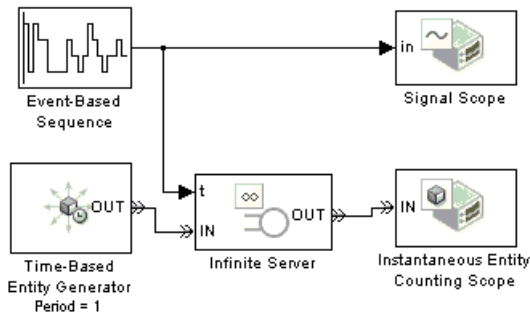
In some cases, you want to visualize how many entities have departed from a particular block and when their departures occurred, but you want to restart the counter at each time instant. This can be useful for

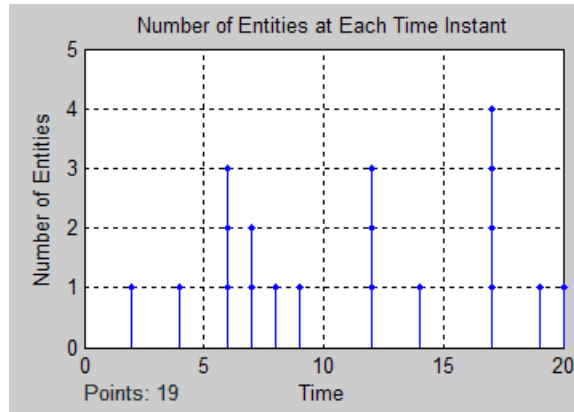
- Detecting simultaneous departures
- Focusing on the departure times without needing to accommodate large counts (for example, in a plot with a large range of axis values)

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 the example below, 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.



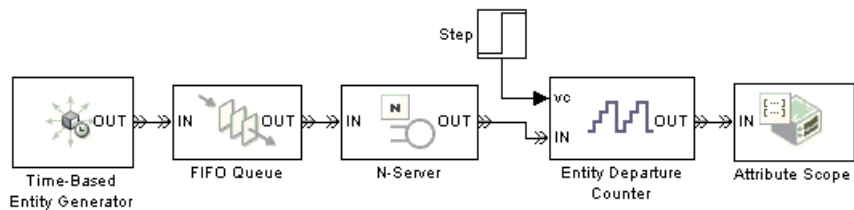


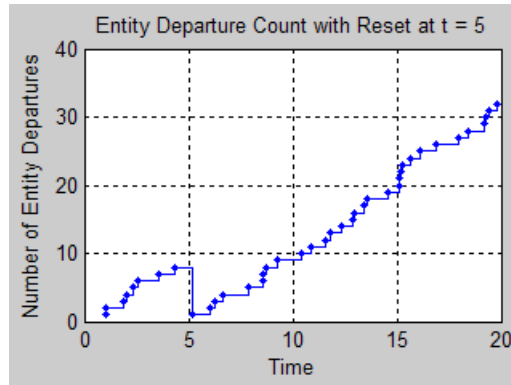
Resetting a Counter Upon an Event

Use the Entity Departure Counter block with **Reset counter upon** set to **Change in signal** from port **vc** or **Trigger** from port **tr** to count entity departures via a resettable counter. For details on this feature, see the reference page for the Entity Departure Counter block.

Example: Resetting a Counter After a Transient Period

The example below 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 an entity count with the entities that use a particular entity 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 1-14.

For an example that illustrates when using the Entity Departure Counter block is more straightforward than storing the #d output signal in an attribute using the Set Attribute block, see “Example: Sequence of Departures and Statistical Updates” on page 14-10.

Combining Entities

- “Overview of the Entity-Combining Operation” on page 1-24
- “Example: Waiting to Combine Entities” on page 1-24
- “Example: Copying Timers When Combining Entities” on page 1-26
- “Example: Managing Data in Composite Entities” on page 1-27

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.

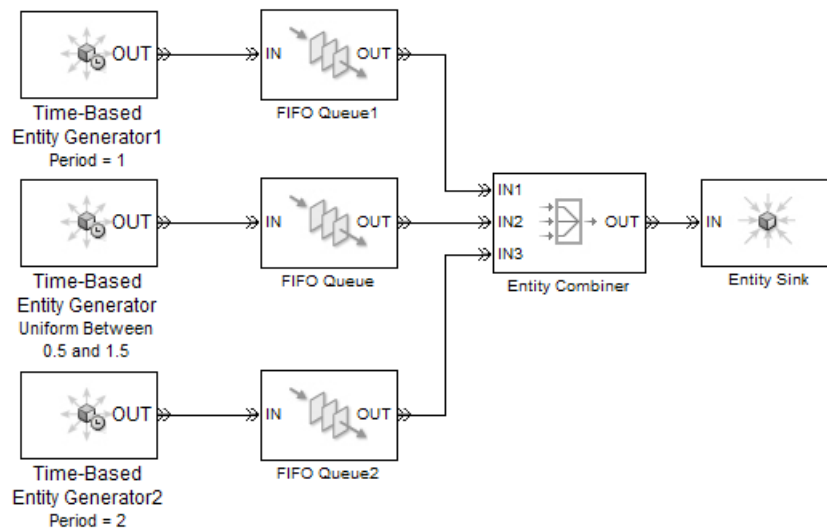
The Entity Combiner block and its surrounding blocks automatically detect when all necessary component entities are present and the result of the combining operation would be able to advance to a storage block. “Example: Waiting to Combine Entities” on page 1-24 illustrates these automatic interactions 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 1-27. 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 Entity Sink block is available, which is true throughout the simulation.

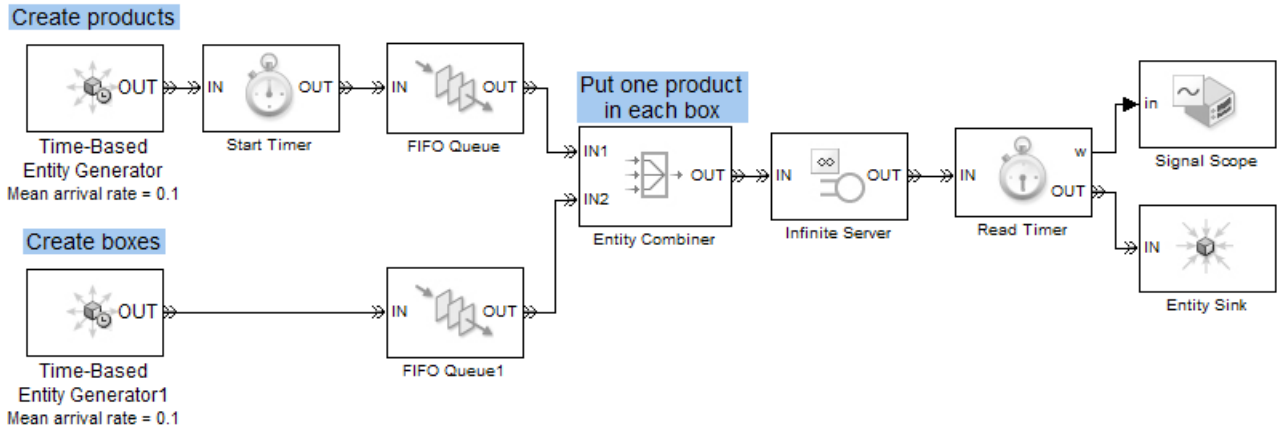
The bottom entity generator has the largest intergeneration time among the three entity generators, and is the limiting factor that determines when the Entity Combiner block accepts one entity from each queue. The top and middle queues store pending entities while waiting for the bottom entity generator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

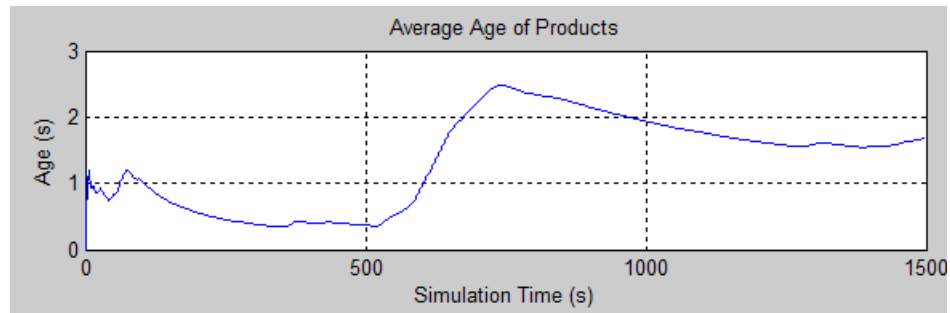
While you could alternatively synchronize the departures from the three queues using appropriately configured gates, it is simpler and more intuitive to use the Entity Combiner block as shown.

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.

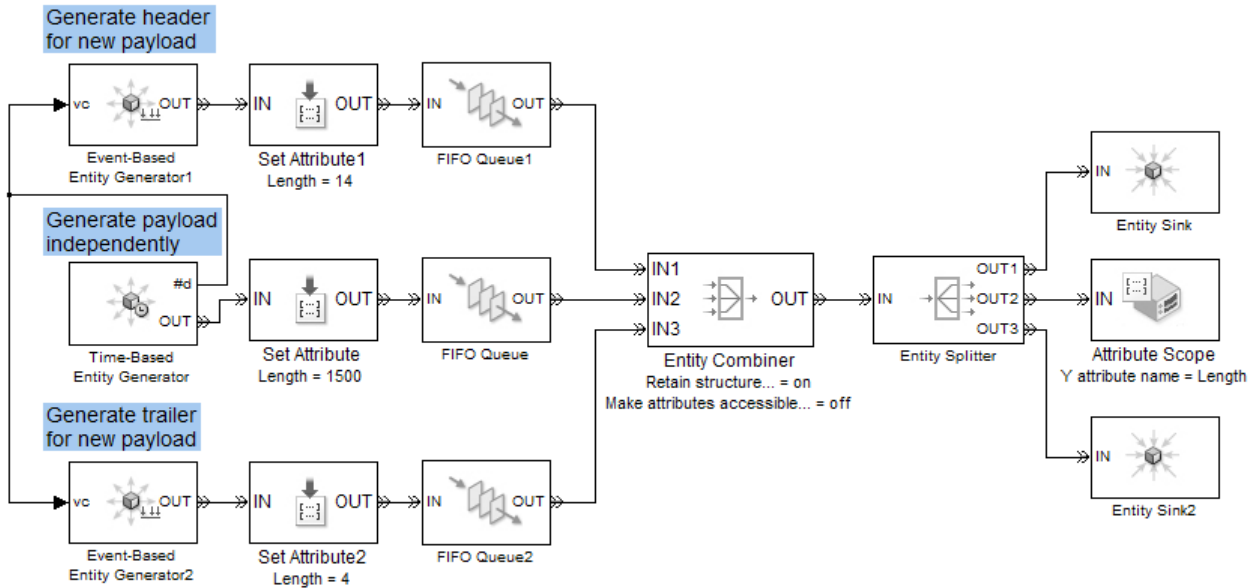


Example: Managing Data in Composite Entities

This section illustrates the choice 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.

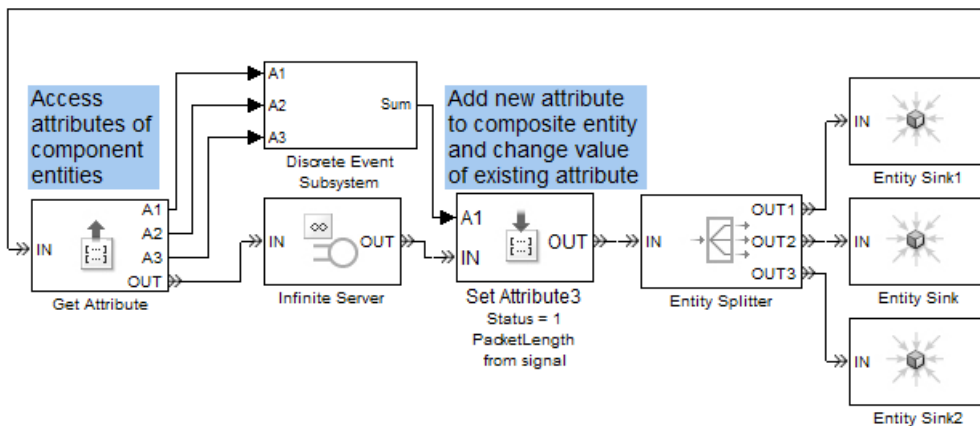
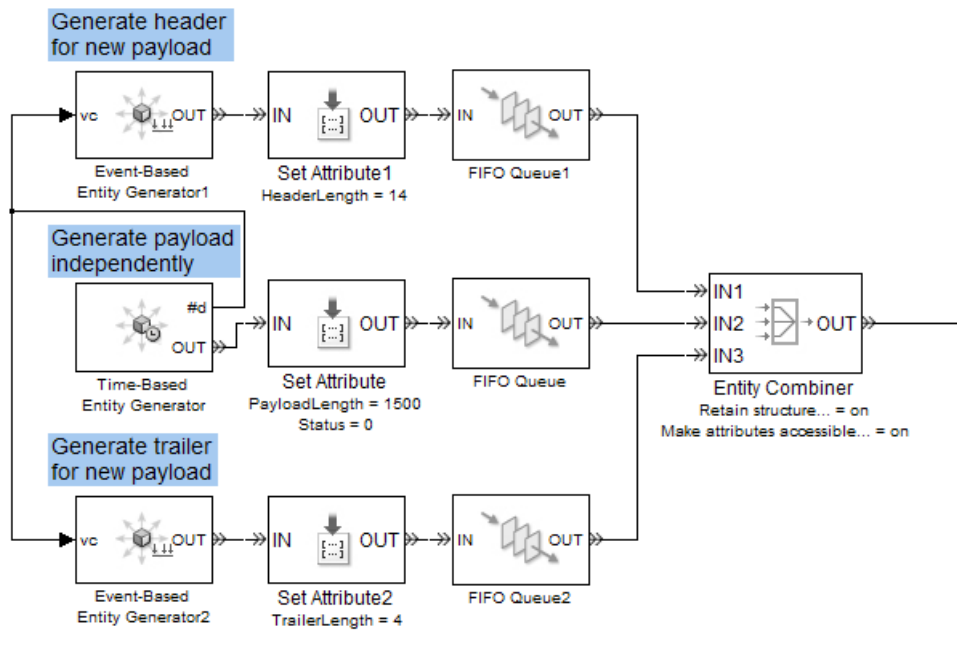


Attribute Names Unique and Accessible in Composite Entity

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 data sources.

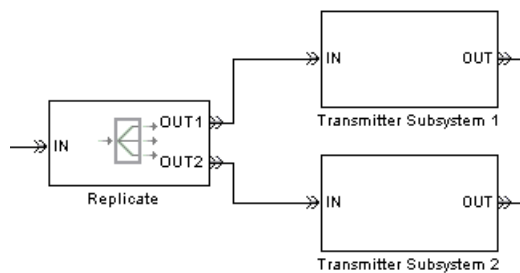
The model illustrates these ways of accessing attributes via the composite entity:

- Reading the lengths using the Get Attribute block.
- Changing the value of the Status attribute using the Set Attribute block. The new value persists when the Entity Splitter block divides the packet into its component entities.
- Defining a new attribute called PacketLength. This is an attribute of the composite entity that is not associated with any of the component entities, so it does not persist beyond the Entity Splitter block.



Replicating Entities on Multiple Paths

You can distribute copies of an entity on multiple entity paths using the Replicate block. Replicating entities might be a requirement of the situation you are modeling or it might be merely a convenient modeling construct. One scenario in which you might replicate entities is when copies of messages in a multicasting communication system advance to multiple transmitters or multiple recipients, as shown in the fragment below.



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

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.

Departure Port Precedence

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 (p. 2-2)

List and discussion of event types

Event Processing in SimEvents
(p. 2-9)

The event calendar and handling of
simultaneous events

Working with Simultaneous Events
(p. 2-14)

Modeling tips, examples, and
procedures for working with
simultaneous events

Observing Events (p. 2-36)

Techniques for determining when
events occur

Generating Function-Call Events
(p. 2-43)

Generating events in an event-based
or time-based manner

Manipulating Events (p. 2-46)

Translating, combining, prioritizing,
and delaying events

Supported Events in SimEvents Models

An event is an instantaneous discrete incident that changes a state variable, an output, and/or the occurrence of other events. This section lists the supported events in SimEvents models and discusses some types of events in greater detail. The topics are

- “Types of Supported Events” on page 2-2
- “Signal-Based Events” on page 2-3
- “Function Calls” on page 2-7

Types of Supported Events

SimEvents supports the events listed below.

Event	Description
Sample time hit	Update in the value of a signal that is connected to a block configured to react to signal updates
Value change	Change in the value of a signal connected to a block that is configured to react to relevant value changes
Trigger	Rising or falling edge of a signal connected to a block that is configured to react to relevant trigger edges
Function call	Discrete invocation request carried from block to block by a special signal called a function-call signal
Entity generation	Creation of an entity
Entity destruction	Arrival of an entity at a block that has no entity output port
Entity advancement	Departure of an entity from one block and arrival at another block
Service completion	Completion of service on an entity in a server
Preemption	Replacement of an entity in a server by a higher priority entity

Event	Description
Timeout	Departure of an entity that has exceeded a previously established time limit
Counter reset	Reinitialization of the counter in the Entity Departure Counter block
Gate opening or closing	Change in the state of the gate represented by the Enabled Gate or Release Gate block
Port selection	Selection of an entity input port in the Input Switch block or an entity output port in the Output Switch block
Memory writing	Writing of memory in the Signal Latch block
Memory reading	Reading of memory in the Signal Latch block

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.

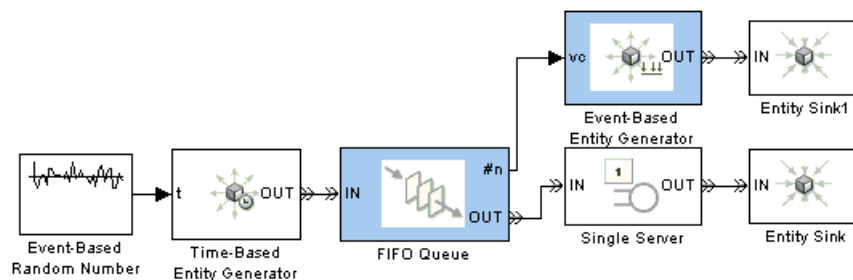
When comparing the types of signal-based events, note that

- 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.

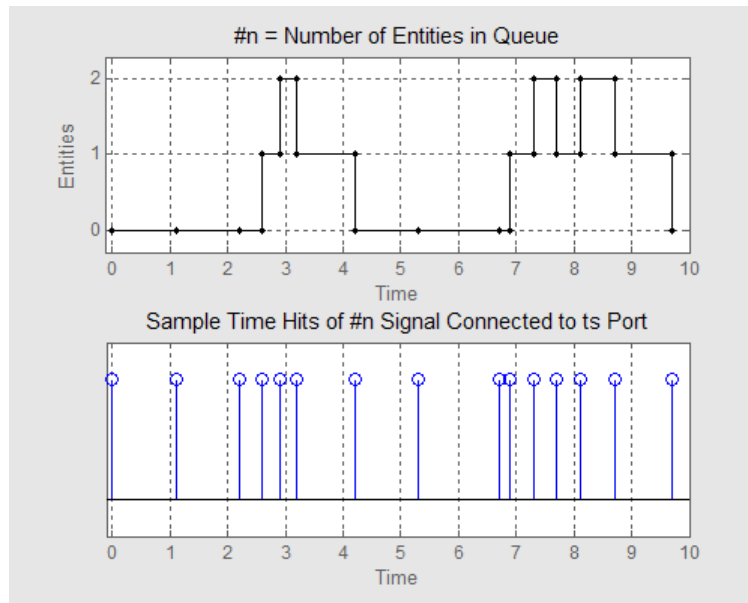
- Triggers and value changes can be rising or falling. You configure a block to determine whether the block considers rising, falling, or either type to be a relevant occurrence.
- Blocks in the Simulink libraries are more restrictive than blocks in the SimEvents libraries regarding trigger edges that 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”. 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

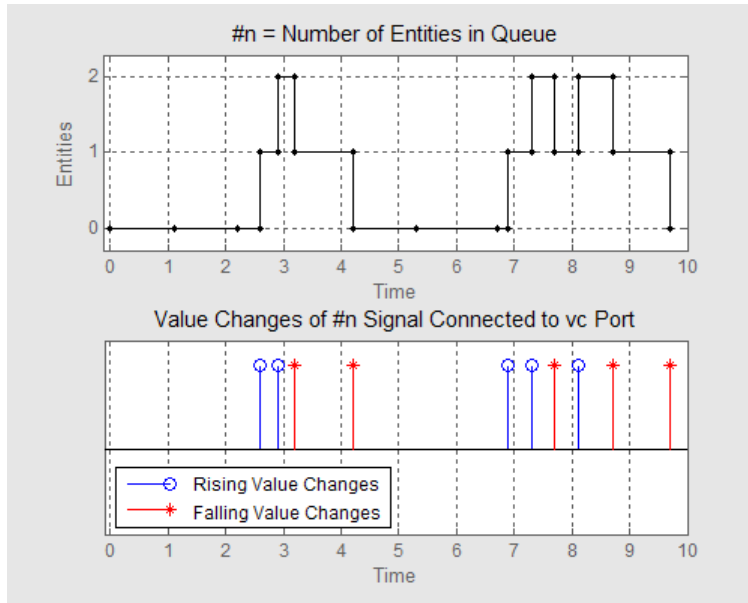
Consider the signal representing the number of entities stored in a FIFO queue. This is the **#n** output signal from the FIFO Queue block in the model below. The **#n** signal is connected to the Event-Based Entity Generator block, which reacts to different types of signal-based events. Parameters in 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.



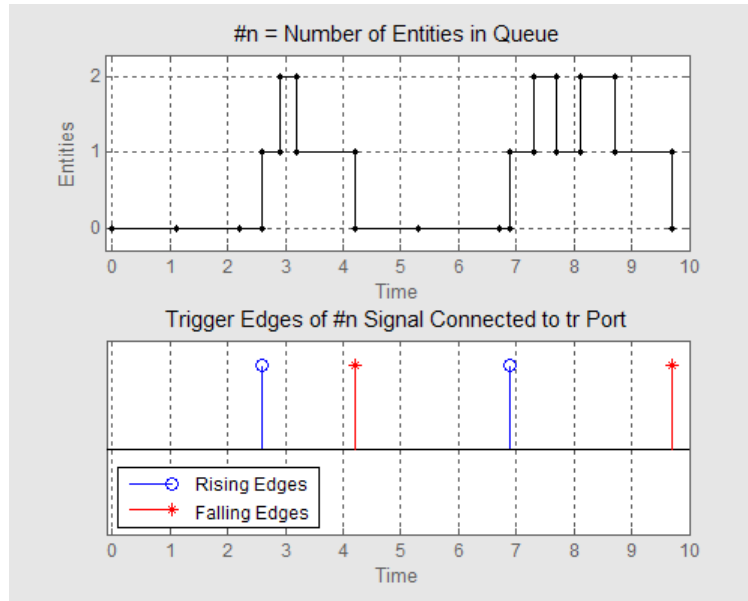
The following figures use a staircase plot to show the values of the **#n** signal, starting from the first entity’s arrival at $T=0$. The figures use stem plots to indicate when signal-based events occur at different signal input ports (**ts**, **vc**, or **tr**) of the Event-Based Entity Generator block.



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 recommended way to make Stateflow blocks and blocks in the Simulink libraries respond to asynchronous state changes.

Function calls and signal-based events are often interchangeable in their ability to elicit reactions from various SimEvents blocks, such as the Event-Based Entity Generator block and the Signal-Based Event to Function-Call Event block.

Function-call signals can be combined, as described in “Creating a Union of Multiple Events” on page 2-47.

Event Processing in SimEvents

During a simulation, multiple events can occur at the same value of the simulation clock, whether or not due to causality. SimEvents maintains a list, called the *event calendar*, of selected upcoming events that are scheduled for the current simulation time or future times. This section indicates which events appear on the event calendar and how the application handles simultaneous events. The topics are

- “Role of the Event Calendar” on page 2-9
- “Processing Sequence for Simultaneous Events” on page 2-11
- “Livelock Detection” on page 2-12

For suggestions and examples related to event processing, see “Working with Simultaneous Events” on page 2-14. To learn how to display event information in the MATLAB Command Window, see “Viewing the Event Calendar” on page 13-2.

Role of the Event Calendar

The table below indicates which events appear or might appear on the event calendar. In some cases, you have a choice.

Event	On Event Calendar	How to Cause Event to Appear on Event Calendar
Sample time hit	Never	
Value change		
Trigger		
Function call	Sometimes	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, then the function call is not on the event calendar.
Entity generation	Sometimes	Use the Time-Based Entity Generator block, or select Resolve simultaneous signal updates according to event priority in the Event-Based Entity Generator block's dialog box.
Entity destruction	Never	
Entity advancement	Never	
Service completion	Always	
Preemption	Never	
Timeout	Always	
Counter reset	Sometimes	Select Resolve simultaneous signal updates according to event priority , if present, in the block's dialog box.
Gate opening or closing		
Port selection		
Memory writing	Sometimes	Select Resolve simultaneous signal updates according to event priority on the Write tab of the block's dialog box.
Memory reading	Sometimes	Select Resolve simultaneous signal updates according to event priority on the Read tab of the block's dialog box.

If an event does not appear on the event calendar, then the application arbitrarily decides when to process the event relative to other events occurring

at the same value of the simulation clock that are also not on the event calendar. When multiple unrelated events occur simultaneously, causality considerations alone do not necessarily determine a unique correct sequence.

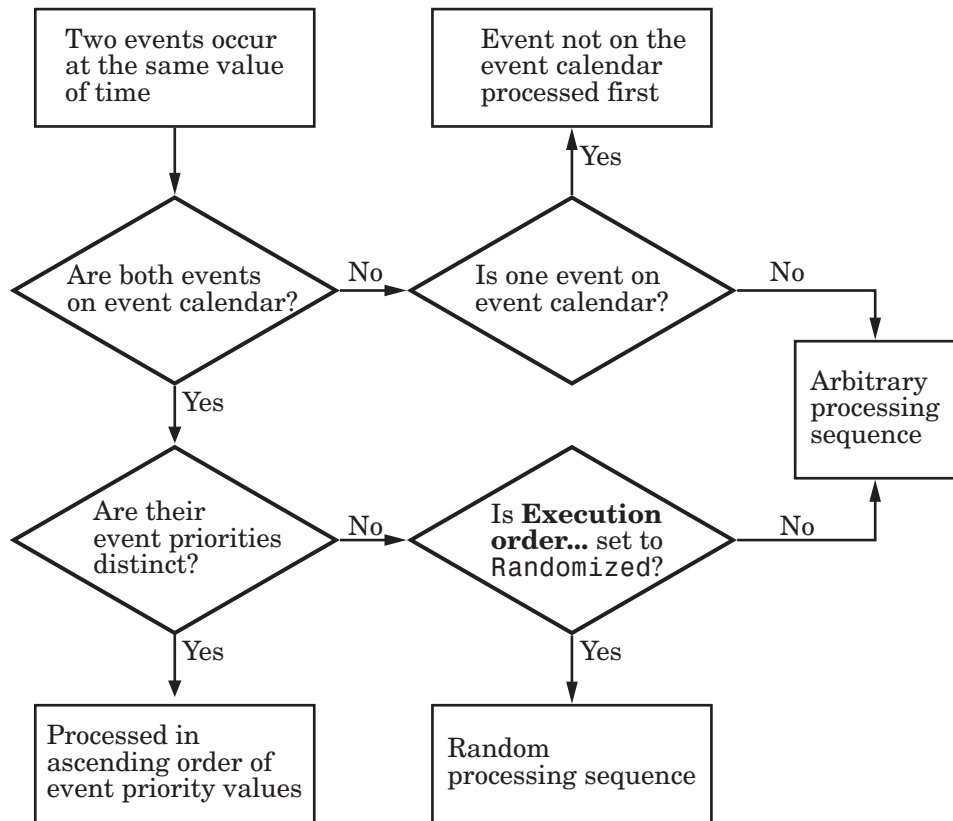
When you use blocks that offer a **Resolve simultaneous signal updates according to event priority** option, your choice determines whether particular events appear on the event calendar. For information about this option, see “Understanding the Resolution Sequence for Input Signals” on page 3-11. For issues you should consider when deciding whether to select this option, see “Choosing an Approach for Simultaneous Events” on page 2-14.

Processing Sequence for Simultaneous Events

Even if simultaneous events occur at the same value of the simulation clock, the application processes them sequentially. If one event causes another event directly or indirectly, the processing sequence must reflect the causality. This section describes the processing sequence for two simultaneous events in the case when neither causes the other. Depending on the event types and on how you have designed your model, the application might determine the processing sequence

- Explicitly according to modeling choices you make. Simultaneous events having distinct event priorities are processed in ascending order of the event priority values. Events not on the event calendar are processed before simultaneous events on the event calendar.
- Randomly, where your choice of a seed for the random number generator provides repeatability.
- Arbitrarily, using an internal algorithm.

The figure below indicates when each kind of determination is in effect. The figure assumes that two events, neither of which causes the other, occur at the same value of the simulation clock. The abbreviation **Execution order...** refers to the **Execution order of simultaneous events** parameter on the **SimEvents** tab of the model’s Configuration Parameters dialog box. See “Events with Equal Priorities” on page 2-16 for details.



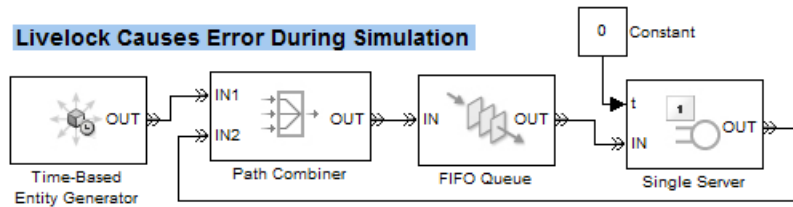
When the sequence is arbitrary, you should not make any assumptions about the sequence or its repeatability.

For suggestions on how to use the information in the figure when creating models, see “Choosing an Approach for Simultaneous Events” on page 2-14.

Livelock Detection

SimEvents includes features to detect livelock and other infinite loops. Livelock is a situation in which an entity moves along a looped entity path with no passage of time and no logic to stop the entity for a nonzero period of time.

The model below shows an example of livelock. The livelock detection 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.



SimEvents also detects causality loops that involve recursion beyond a fixed limit.

Working with Simultaneous Events

Building on the descriptions in “Event Processing in SimEvents” on page 2-9, this section provides modeling suggestions and examples. The topics are

- “Choosing an Approach for Simultaneous Events” on page 2-14
- “Setting Event Priorities” on page 2-15
- “Example: Event Calendar for a Queue-Server Model” on page 2-17
- “Example: Race Conditions at a Switch” on page 2-25
- “Events On and Off the Event Calendar” on page 2-31

Choosing an Approach for Simultaneous Events

Building on the logic depicted in “Processing Sequence for Simultaneous Events” on page 2-11, here are some examples of situations in which you might want to use different modeling approaches for working with simultaneous events and the event calendar:

- You want the simulation to run as fast as possible and you know that the specific processing sequence for unrelated simultaneous events has little or no effect on your simulation results. In this case, you leave **Resolve simultaneous signal updates according to event priority** parameters at their default values.
- You need to make a block react to relevant updates in its input signal upon detecting the update, without necessarily waiting for other block operations. In this case, you turn off the block’s **Resolve simultaneous signal updates according to event priority** option.
- You need to make a block defer the reaction to relevant updates in its input signal until after other operations at that time have been processed. In this case, you select the block’s **Resolve simultaneous signal updates according to event priority** option. Upon detecting the update or change in the block’s input signal, the application schedules an event on the event calendar for the current simulation time.

For examples, see “Events On and Off the Event Calendar” on page 2-31.

Note As a special case, consider a Stateflow chart having both a function-call output and a signal output, connected to SimEvents blocks that use the Stateflow outputs at the same time. Selecting **Resolve simultaneous signal updates according to event priority**, if present, in the SimEvents block that reacts to the function call might prevent latency issues. To learn more about using SimEvents and Stateflow blocks together in a model, see Chapter 12, “Using Stateflow with SimEvents”.

- Unrelated simultaneous events seem likely to occur and their sequence can alter your simulation results. In this case, you select **Resolve simultaneous signal updates according to event priority** in most or all of the blocks involved in the collection of simultaneous events. Then use event priorities as described in “Setting Event Priorities” on page 2-15 to determine a sequence that meets your simulation needs. Simultaneous events having distinct event priorities are processed in ascending order of the event priority values.

For examples that show the effect of changing event priorities, see “Example: Race Conditions at a Switch” on page 2-25 and the Event Priorities demo.

Setting Event Priorities

If the event calendar contains two or more events that are scheduled for times that are equal or sufficiently close, then you can assign event priorities to influence the processing sequence of the events. Simultaneous events having distinct event priorities are processed in ascending order of the event priority values. To assign event priorities, 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**, then 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. When choosing an integer, remember that 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.

Events that are not on the event calendar have no event priority. Such events are processed before unrelated events scheduled on the event calendar for the same time. However, you cannot vary, or necessarily predict, the processing sequence of multiple unrelated simultaneous events that are not on the event calendar.

For examples that show the effect of changing event priorities, see “Example: Race Conditions at a Switch” on page 2-25 and the Event Priorities demo.

Events with Equal Priorities

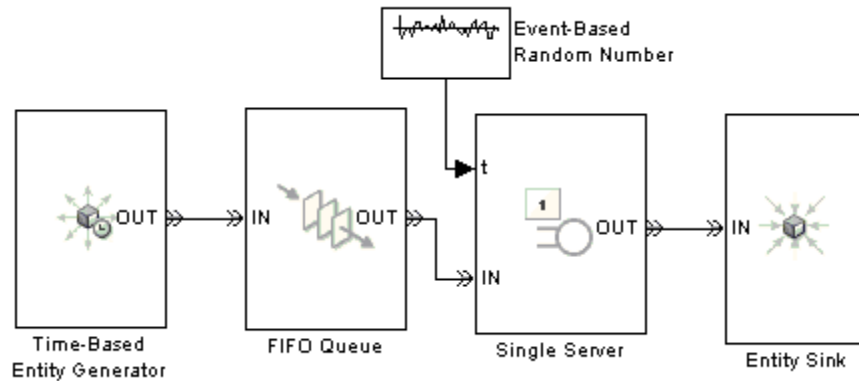
If simultaneous events on the event calendar have equal event priorities, then the application arbitrarily or randomly determines the processing sequence, depending on a modelwide configuration parameter. To access this parameter, use this procedure:

- 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 of simultaneous events** 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: Event Calendar for a Queue-Server Model

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 random service times that are uniformly distributed between 0.5 and 2.5 seconds.

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.

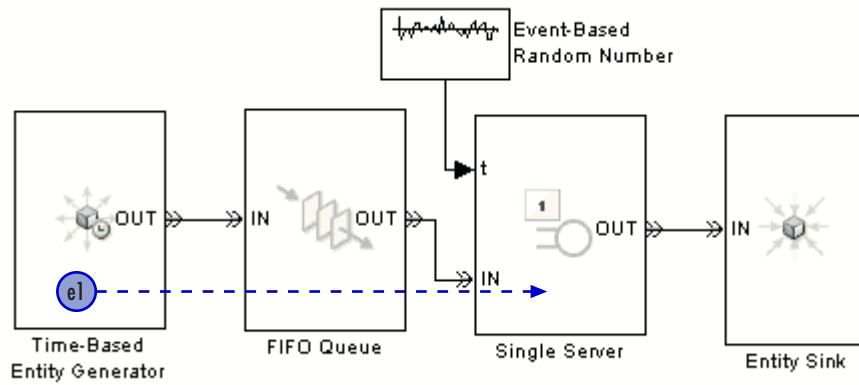
Time of Event (s)	Type of Event
0.9	Time-Based Entity Generator block generates an entity.

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. Suppose the (stochastic) service time turns out to be 1.3 in this case. This means that the duration of service 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.



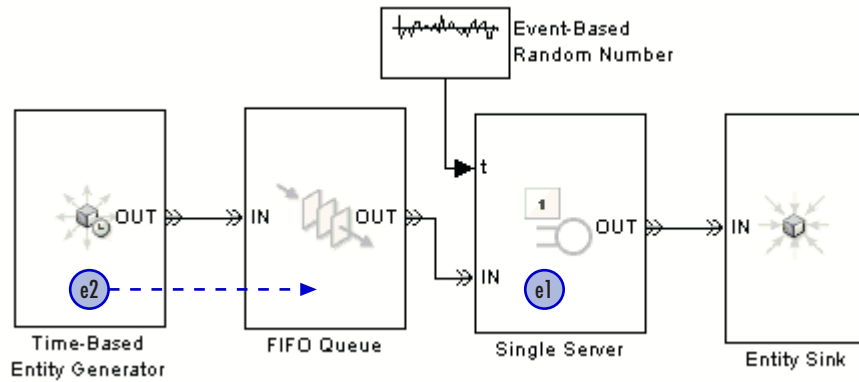
The event calendar looks like this.

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 $T = 3.8$.



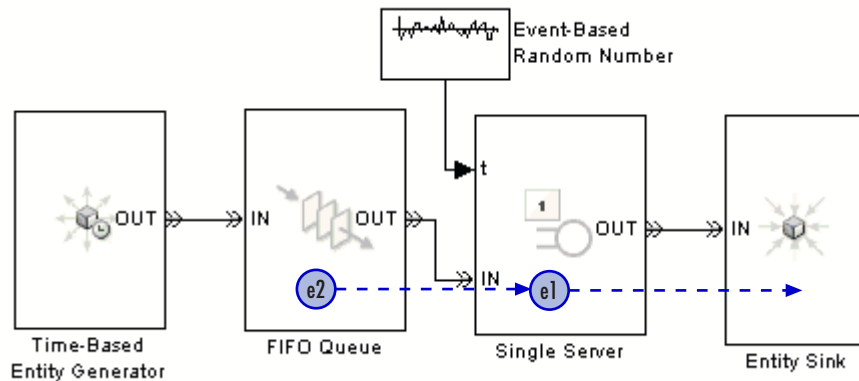
Time of Event (s)	Event Description
2.2	Single Server block completes service on the first entity.
3.8	Time-Based Entity Generator block generates the third entity.

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 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. Suppose the service time turns out to be 2.0 in this case.

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.



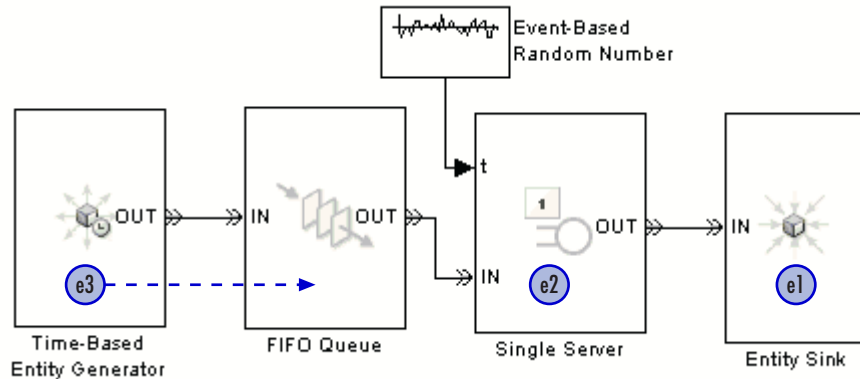
Time of Event (s)	Event Description
3.8	Time-Based Entity Generator block generates the third entity.
4.2	Single Server block completes service on the second entity.

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 $T = 3.9$.

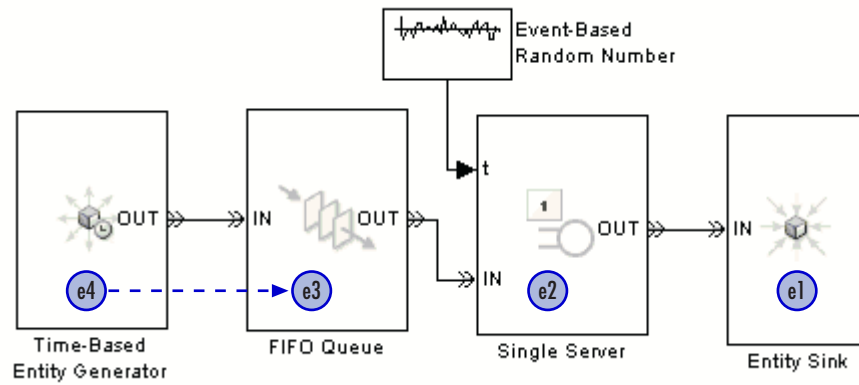


Time of Event (s)	Event Description
3.9	Time-Based Entity Generator block generates the fourth entity.
4.2	Single Server block completes service on the second entity.

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.

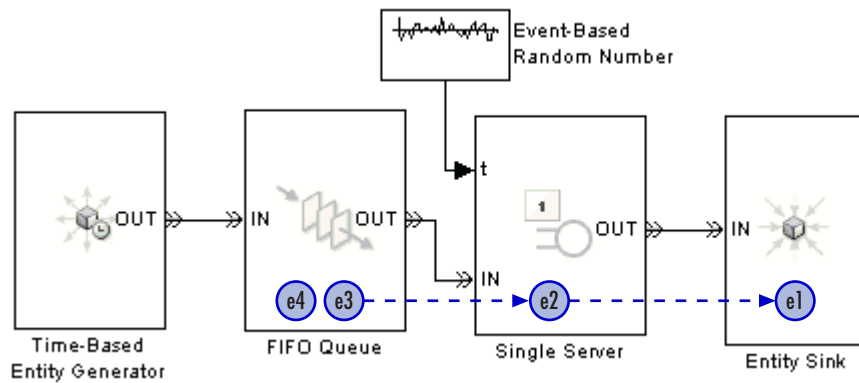
Completion of Service Time

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 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. Suppose the service time turns out to be 0.7 in this case.

- 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.

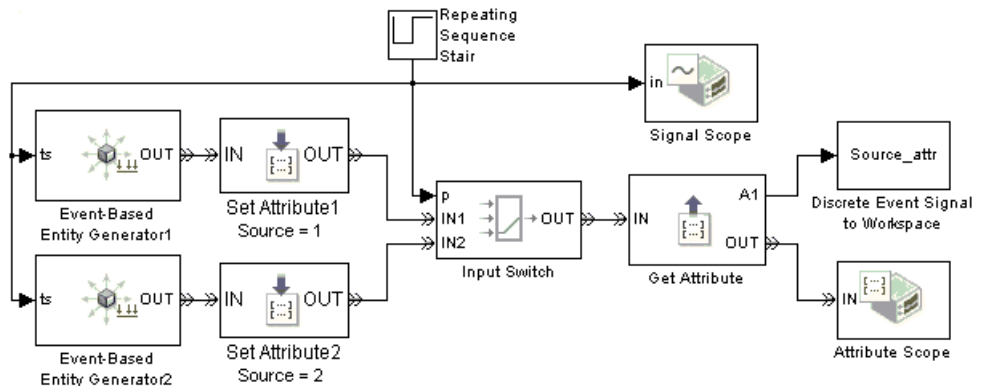
Note The queue's entity output port started this time instant in the blocked state, became unblocked (when it sensed that the server's entity input port became available), and then became blocked once again (when the server began serving the third entity).



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

Example: Race Conditions at a Switch

This example shows how you can vary the processing sequence for simultaneous events by varying their event priorities.



At $T=1, 2, 3, \dots$ Repeating Sequence Stair block changes its value from 1 to 2 or vice versa. 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, where 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 predict the sequence in which the blocks react 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 model is configured so that 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	200

At T=1,

- 1 The switch selects its **IN2** entity input port.
- 2 The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.
- 3 The bottom entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block.

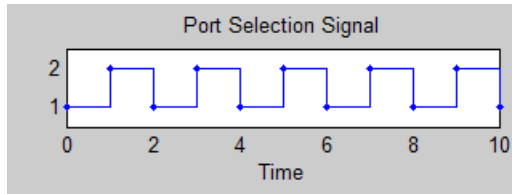
At $T=2$,

- 1** 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.
- 2** 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.
- 3** The bottom entity generator generates an entity, which cannot depart because the switch's **IN2** entity input port is unavailable.

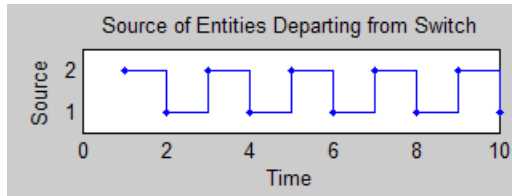
At $T=3$,

- 1** 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.
- 2** The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.
- 3** 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.

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 model is configured so that 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.

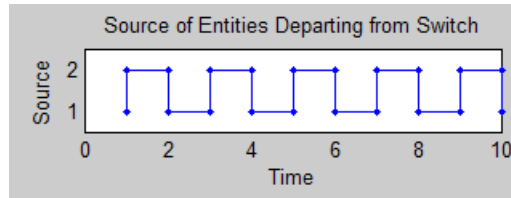
At T=1,

- 1** The top entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block.
- 2** The bottom entity generator generates an entity, which cannot depart because the switch's **IN2** entity input port is unavailable.
- 3** 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.

At T=2,

- 1** The top entity generator generates an entity, which cannot depart because the switch's **IN1** entity input port is unavailable.
- 2** The bottom entity generator generates an entity, which advances from block to block until it reaches the Attribute Scope block.
- 3** 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.

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 =
```

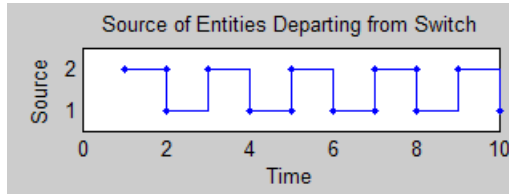
1	1
1	2
2	2
2	1
3	1
3	2
4	2
4	1
5	1
5	2
6	2
6	1
7	1
7	2
8	2
8	1
9	1
9	2
10	2
10	1

Randomly Selecting a Sequence

Suppose the model is configured so that 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 of simultaneous events** 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

Events On and Off the Event Calendar

This section presents these examples to illustrate when the decision to put an event on the event calendar — that is, your decision regarding the **Resolve**

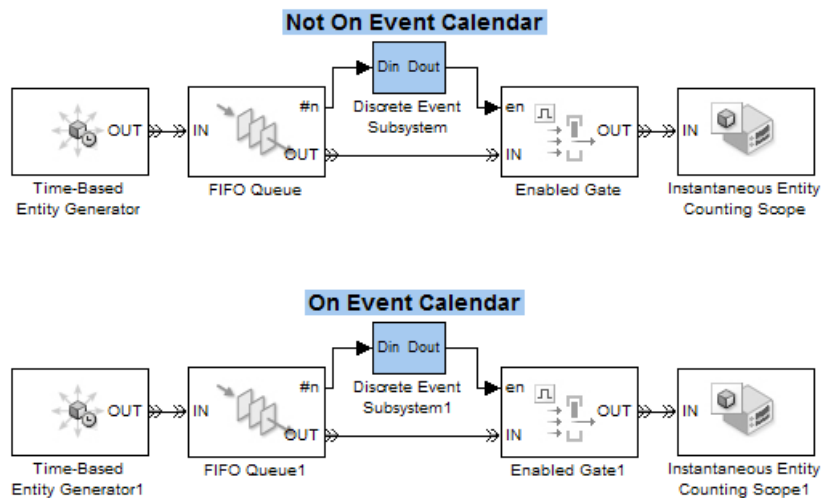
simultaneous signal updates according to event priority option — affects the simulation significantly:

- “Example: Queue Thresholds” on page 2-32
- “Example: Simultaneous Events in Aggregation Demo” on page 2-34

Another relevant example is “Example: Using the Event Calendar to Prevent Interleaving” on page 14-14.

Example: Queue Thresholds

The example below illustrates how deferring the reaction to a signal update until after other simultaneous block operations have been considered can change the way a gate lets entities out of a queue.



In the top row of blocks, the subsystem contains a Discrete Event Inport block with the **Resolve simultaneous signal updates according to event priority** option not selected. In the bottom row of blocks, the subsystem contains a Discrete Event Inport block with the **Resolve simultaneous signal updates according to event priority** option selected. In both cases, the subsystem returns 1 when the queue length is greater than or equal to 5. During the simulation, each queue accumulates entities until it updates the queue length signal, **#n**, to 5. At that point, the behaviors of the two

portions of the model diverge because the top portion of the model reevaluates the threshold condition as soon as it detects the change in **#n**, while the bottom portion of the model schedules an event to reevaluate the threshold condition. The sequences of relevant operations in the two portions of the model are as follows:

Operations in Top Portion of Model

- 1** The **#n** signal becomes 5.
- 2** The subsystem executes immediately and finds that the queue length is at the threshold.
- 3** The gate opens.
- 4** One entity departs from the queue.
- 5** The queue length decreases.
- 6** The subsystem executes immediately and finds that the queue length is beneath the threshold.
- 7** The gate closes.

Operations in Bottom Portion of Model

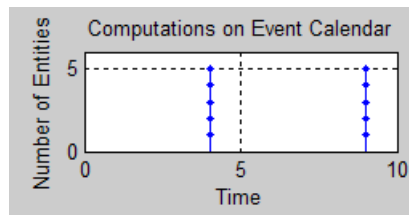
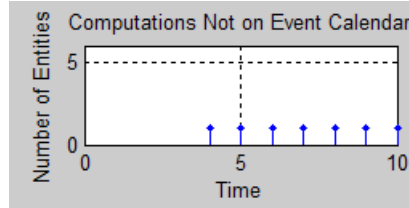
- 1** The **#n** signal becomes 5.
- 2** The subsystem schedules an execution event for the current time.
- 3** The event calendar causes the subsystem to execute. The subsystem finds that the queue length is at the threshold.
- 4** The gate opens.
- 5** One entity departs from the queue.
- 6** The queue length decreases.
- 7** The subsystem schedules an execution event for the current time.
- 8** Steps One entity departs from the queue.

on page 33 through The subsystem schedules an execution event for the current time.

on page 33 repeat until the queue is empty. The gate remains open during this period. This step shows the difference in simulation behavior between scheduling the execution event and processing it immediately upon detecting the decrease in queue length.

- 9 The event calendar causes the subsystem to execute. The subsystem finds that the queue length is beneath the threshold.
- 10 The gate closes.
- 11 The event calendar causes the subsystem to execute additional times, but the subsystem output is the same.

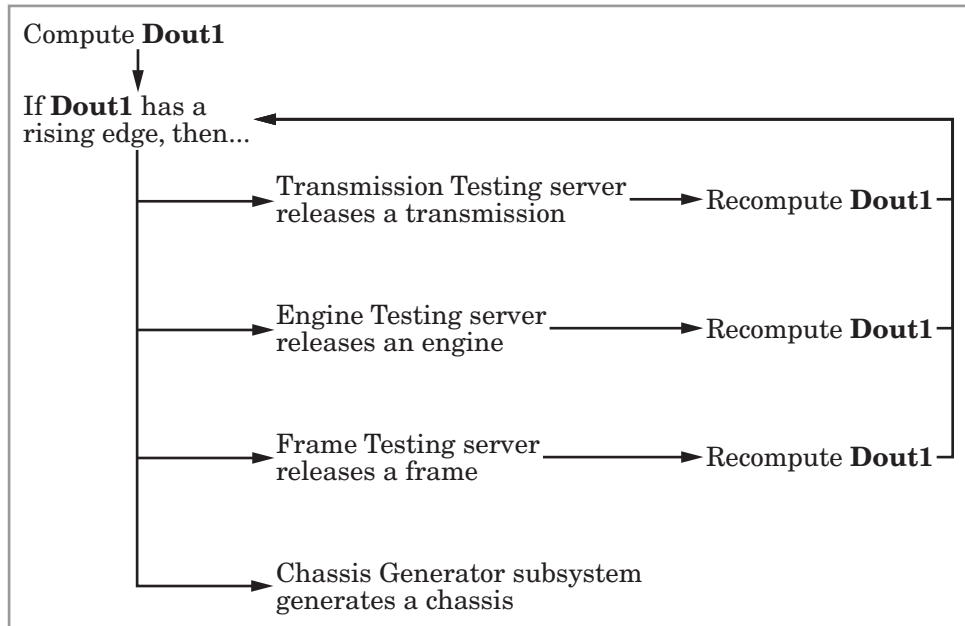
In summary, when the queue length reaches the threshold, the top portion of the model releases one entity so that the queue length is just beneath the threshold, while the bottom portion of the model empties the queue. The plots of departures from the gates reflect this behavior.



Example: Simultaneous Events in Aggregation Demo

The Aggregation: Assembling a Vehicle Chassis demo illustrates why the event calendar can be useful for deferring the processing of an event until other operations at that time have completed their processing. The next figure

shows some events in the model that simultaneously cause other events. A rising edge of the **Dout1** signal causes four simultaneous events, three of which are release events that in turn cause a simultaneous recomputation of **Dout1**.



It is important to defer the recomputation of **Dout1** until after the three release events are complete, or else a recomputation of **Dout1** might eliminate the rising edge and prevent the processing of one of the release events. To defer the recomputation of **Dout1** until after the three release events are complete, the model puts the subsystem execution event on the event calendar while keeping the release events off the event calendar. Specifically, the model selects the **Resolve simultaneous signal updates according to event priority** option in the Din, Din1, and Din2 blocks within the Chassis Assembly Control subsystem and does not select the **Resolve simultaneous signal updates according to event priority** option in the Release Frame, Release Engine, and Release Transmission blocks. As a result, whenever a testing server changes its output signal, the Chassis Assembly Control schedules the computation of **Dout1** on the event calendar (for the current time instant) instead of proceeding immediately with the computation.

Observing Events

- “Example: Observing Service Completions” on page 2-38
- “Example: Detecting Collisions by Comparing Events” on page 2-40

The event logging feature described in “Viewing the Event Calendar” on page 13-2 can help you observe events that appear on the event calendar. The table below suggests some ways to observe events that do not appear on the event calendar. 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 9-23.

Note Do not select **Resolve simultaneous signal updates according to event priority** options in block dialog boxes for the sole purpose of using the event logging feature to observe those events. When you have a choice about whether an event appears on the event calendar, your decision might affect the processing sequence of simultaneous events and hence the simulation behavior.

Event	Observation Technique
Sample time hit	Use a branch line to connect the signal to an Instantaneous Event Counting Scope block.
Value change	
Trigger	
Function call	If the block issuing the function call provides a #f1 output signal, then 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 <code>fcn</code> ; then insert this block between the block issuing the function call and the block reacting to the function call.

Event	Observation Technique
Entity generation	Observe values of the entity generator's pe output signal. Upon an entity generation, pe repeats a previous value of 0 (if the generated entity departs immediately) or increases from 0 to 1 (if the entity cannot depart).
Entity destruction	Observe increases in the #a output signal. The Instantaneous Entity Counting Scope block provides a plot in place of a #a signal.
Entity advancement	Observe increases in the #d output signal of the block from which the entity departs. Alternatively, use the entity logging feature described in "Viewing Entity Locations" on page 13-9.
Service completion	Observe values of the server's #pe (or pe in the case of the Single Server block) output signal. Upon a service completion, the signal repeats a previous value of 0 (if the entity departs immediately) or increases (if the entity cannot depart). Because simultaneous repeated values would not be easy to observe on a plot, it is best to observe the signal value in the MATLAB workspace if a departure and a new service completion might occur simultaneously in your model.
Preemption	Observe increases in the #p output signal of the server block.
Timeout	Observe increases in the #to output signal of the storage block from which the entity times out.
Counter reset	Observe falling edges in the counter block's #d output signal. Alternatively, use a branch line to connect the counter block's input signal to an Instantaneous Event Counting Scope block.
Gate opening or closing	Use a branch line to connect the gate block's input signal to an Instantaneous Event Counting Scope block. In the case of an enabled gate, rising trigger edges of the input signal indicate gate-opening events, while falling trigger edges of the input signal indicate gate-closing events.

Event	Observation Technique
Port selection	If the block has a p input signal, use a branch line to connect the p signal to an Instantaneous Event Counting Scope block, configured to plot value changes. Otherwise, observe the block's last output signal.
Memory writing	Observe sample time hits in the Signal Latch block's mem output signal.
Memory reading	Observe sample time hits in the Signal Latch block's out output signal.

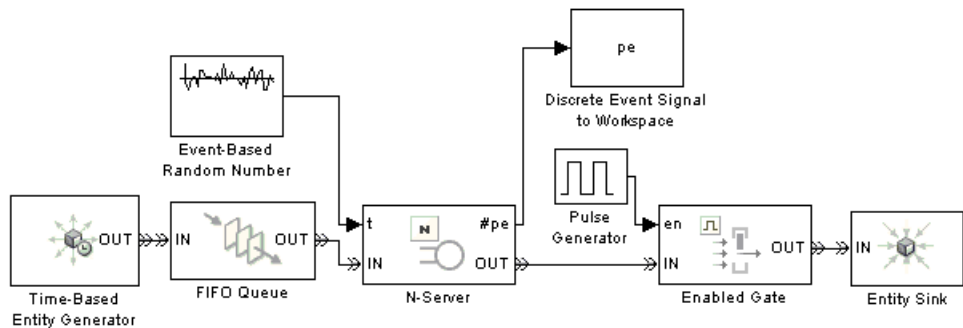
For examples that use one or more of these techniques, see

- “Example: Plotting Event Counts to Check for Simultaneity” on page 10-14
- “Example: Entity Logging” on page 13-10
- “Example: Observing Service Completions” on page 2-38
- “Example: Focusing on Events, Not Values” on page 9-23

Also, “Example: Detecting Collisions by Comparing Events” on page 2-40 shows how to use a Signal Latch block to observe which of two types of events occurred more recently.

Example: Observing Service Completions

The example below writes an N-Server block's **#pe** signal to a structure called `pe` in the MATLAB workspace. In the structure, `pe.time` indicates when each value is attained and `pe.signals.values` indicates the corresponding value.



An entity completes its service at precisely those times when the **#pe** signal repeats its previous value or increases to a larger value. After the simulation is over, you can form a vector of service completion times using the code below.

```
% Output #pe times and values.
pe_matrix = [pe.time, pe.signals.values]

% Determine when #pe changes its value.
dpe = [0; diff(pe.signals.values)];

% Service completions occur when #pe does not decrease.
t_svcpc = pe.time(dpe >= 0)
```

Sample output, which depends on the entity generation, service completion, gate opening, and gate closing times in the model, is below. Notice the rows of `pe_matrix` in which the first column is 2.0000 and the second column has decreasing values. These correspond to decreases in the **#pe** signal value at $T=2$, which result from departures of entities that previously completed their service. The value $T=2$ does not appear in the `t_svcpc` vector because this is not a time at which service completion events occur.

```
pe_matrix =

    0.9077    1.0000
    1.0849    2.0000
    1.8895    3.0000
    2.0000    2.0000
    2.0000    1.0000
```

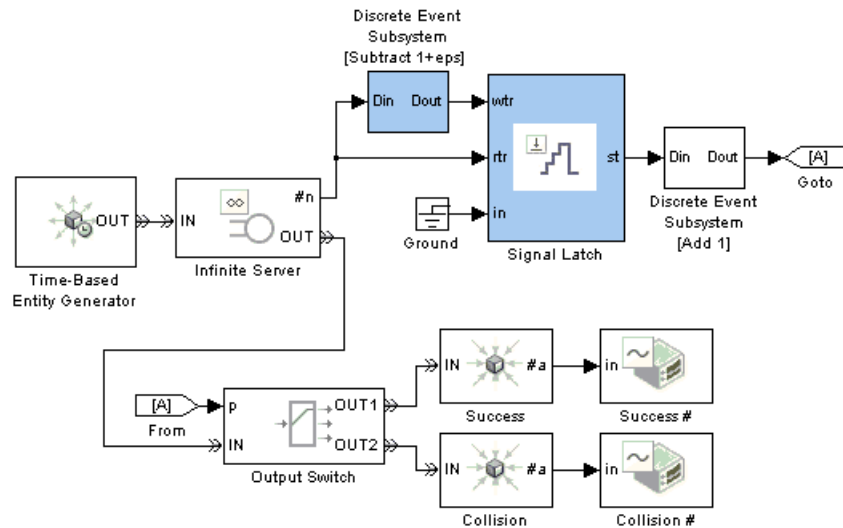
2.0000	0
2.4487	0
3.0707	0
3.1958	0
3.4068	0
3.5856	0
3.7906	0
4.3674	0
4.8672	1.0000
4.9545	2.0000

t_svcp =

0.9077
1.0849
1.8895
2.4487
3.0707
3.1958
3.4068
3.5856
3.7906
4.3674
4.8672
4.9545

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

You can generate an event and use it to

- Invoke a discrete event subsystem or a 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 and Stateflow blocks upon trigger edges, this method has limitations in discrete-event simulations. In particular, you should use function calls instead of trigger edges 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.

These topics describe how to generate function calls in an event-based or time-based manner:

- “Generating Events When Other Events Occur” on page 2-43
- “Generating Events Using Intergeneration Times” on page 2-45

Generating Events When Other Events Occur

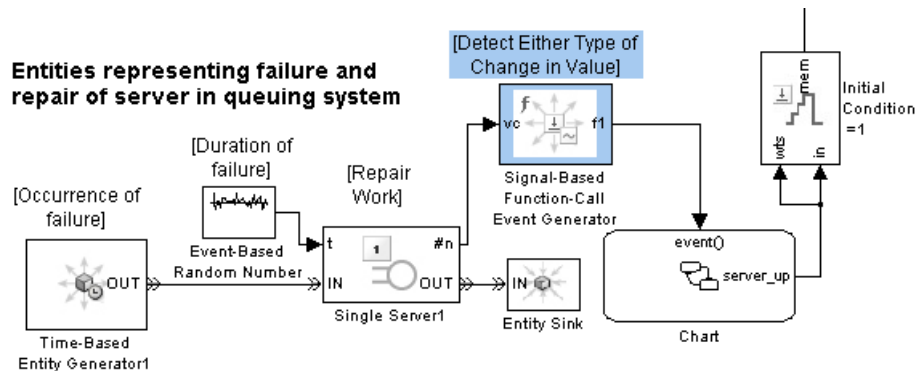
The table below indicates which blocks generate function calls when other events occur.

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 to Implement a Failure State” on page 4-16, 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 trigger signals 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:

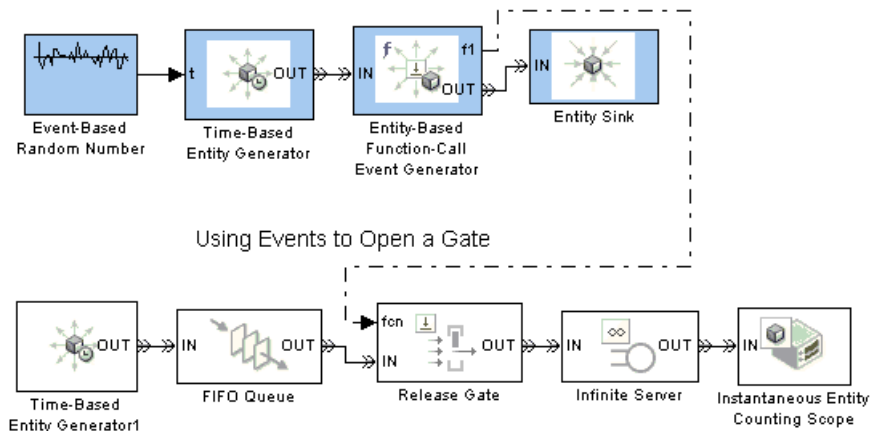
- 1 Use the signal or statistical distribution with the Time-Based Entity Generator block to generate entities.
- 2 Use the Entity-Based Function-Call Event Generator block to generate an event associated with each entity.
- 3 Terminate the entity path with an Entity Sink block.

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



Manipulating Events

- “Blocks for Manipulating Events” on page 2-47
- “Creating a Union of Multiple Events” on page 2-47
- “Translating Events to Control the Processing Sequence” on page 2-50
- “Conditionalizing Events” on page 2-52

You can manipulate an event to accomplish any of these goals:

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

Note You can invoke triggered subsystems 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 2-47.
- To prioritize the reaction to an event relative to simultaneous events. See “Translating Events to Control the Processing Sequence” on page 2-50.
- To delay the reaction to an event. See the **Function-call time delay** parameter on the Signal-Based Event to Function-Call Event block’s reference page.
- To conditionalize the reaction to an event. See “Conditionalizing Events” on page 2-52.

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 9-32 and below.

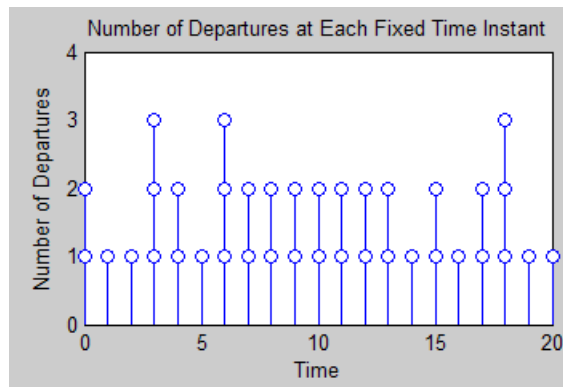
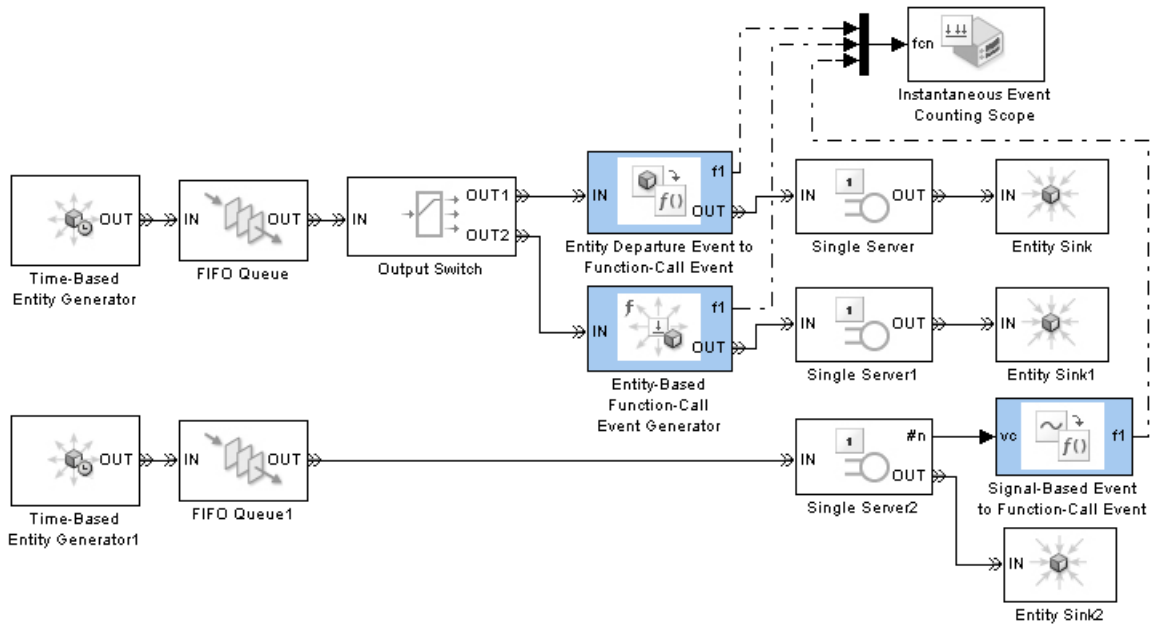
Example: Counting Events from Multiple Sources

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:

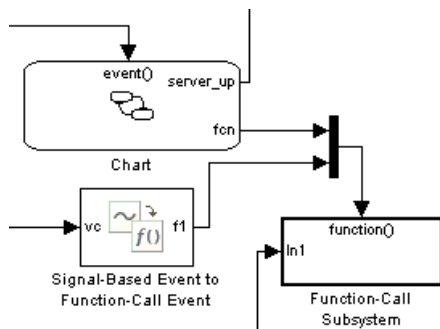
- One path uses the Entity Departure Event to Function-Call Event block, treating the problem as one of event translation.
- One path uses the Entity-Based Event Generator block, treating the problem as one of event generation. This is similar to the approach above.
- One path uses the Signal-Based Event 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 Simulink uses to solve a model. Blocks are made up of multiple methods. For details, see “Block Methods” in the Simulink documentation.



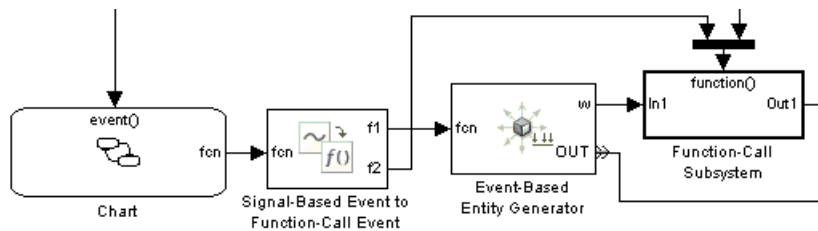
Another similar example is in “Example: Performing a Computation on Selected Entity Paths” on page 9-32.

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.

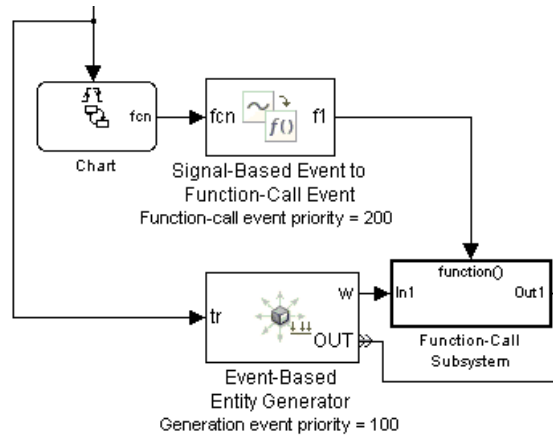
Example: Issuing Two Function Calls in Sequence

In the model below, 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.



Example: Generating a Function Call with an Event Priority

The model below 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 on the event calendar, with a new function call that appears on the event calendar with a priority of 200. The Event-Based Entity Generator block creates 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.

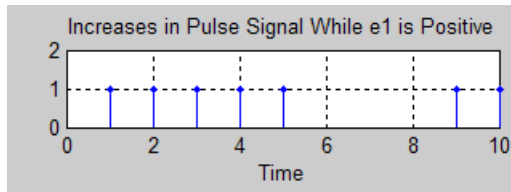
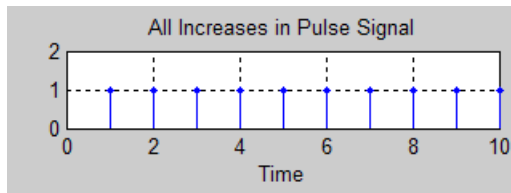
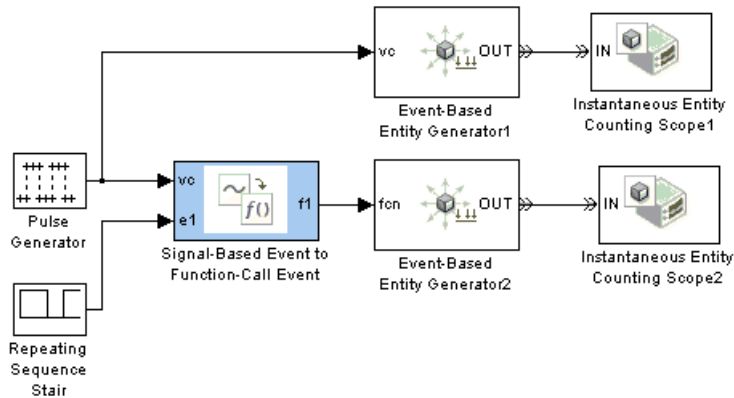
Conditionalizing Events

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 3-18 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.



Working with Signals

Role of Event-Based Signals in SimEvents Models (p. 3-2)	Overview of event-based signals and issues involving them
Generating Random Signals (p. 3-4)	Producing random numbers in an event-based or time-based manner
Using Data Sets to Create Event-Based Signals (p. 3-9)	Generating event-based signals using sequences of data you provide
Understanding the Resolution Sequence for Input Signals (p. 3-11)	How the application resolves updates in input signals, especially when simultaneous with other operations
Choosing How to Resolve Simultaneous Signal Updates (p. 3-17)	Links to information that can help you use the resolution options effectively
Update Sequence for Output Signals (p. 3-18)	Signal updates relative to each other
Multiple Simultaneous Updates of an Output Signal (p. 3-21)	Working with zero-duration values
Latency in Signal Updates (p. 3-25)	Delays in signal updates or responses to updates
Manipulating Signals (p. 3-27)	Using the Signal Latch block to delay or resample signals
Sending Data to the MATLAB Workspace (p. 3-31)	Collecting data from event-based signals for manipulation in MATLAB

Role of Event-Based Signals in SimEvents Models

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.

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

Discrete-event simulations often use random numbers for entity intergeneration times, service times, routing, and other purposes. An important block for generating random signals is the Event-Based Random Number block. These topics describe how to use this block to produce random signals:

- “Generating Random Event-Based Signals” on page 3-4
- “Examples of Random Event-Based Signals” on page 3-5
- “Generating Random Time-Based Signals” on page 3-6

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 14-4. 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 **wfcn**) 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 getting started documentation
- “Example: A Packet Switch” in the getting started documentation
- “Example: Using Random Service Times in a Queuing System” in the getting started documentation
- “Example: Event Calendar for a Queue-Server Model” on page 2-17
- “Example: M/M/5 Queuing System” on page 4-13
- “Example: Compound Switching Logic” on page 5-7

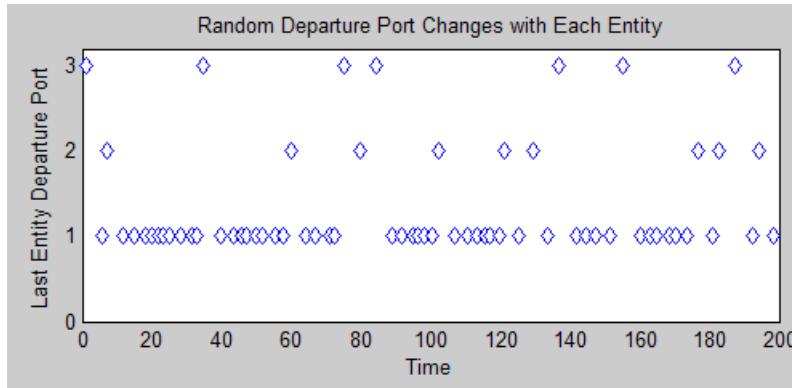
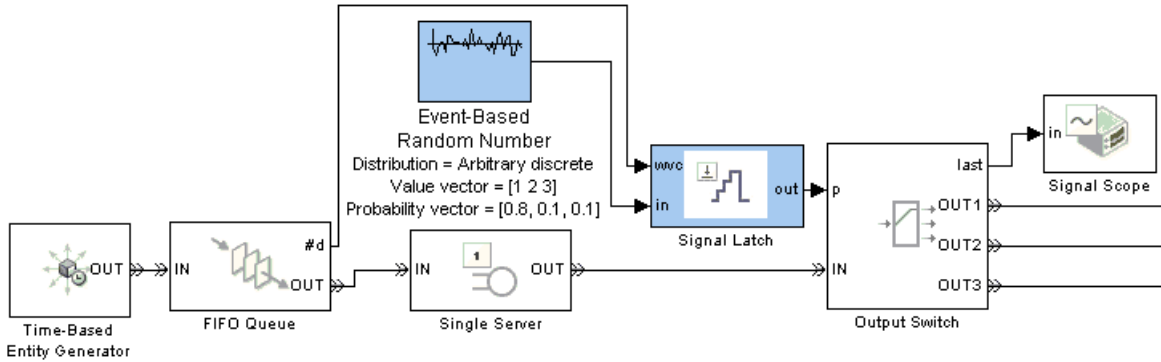
The model in “Example: Compound Switching Logic” on page 5-7 also illustrates how to use the Signal Latch block as described in “Generating Random Signals Based on Arbitrary Events” on page 3-4, to generate a random number upon each departure from an Input Switch block.

The models in “Example: Invalid Connection of Event-Based Random Number Generator” on page 13-25 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 9-30, 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

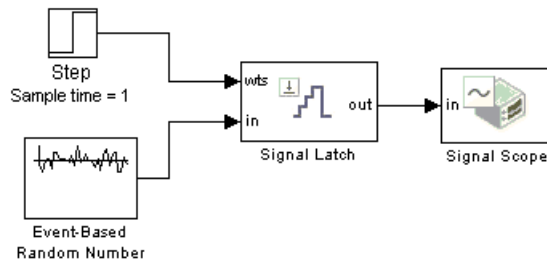
time-based random signals using the Event-Based Random Number block, use this procedure:

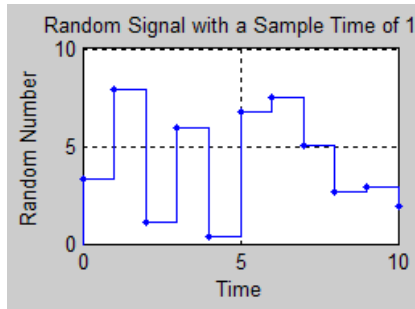
- 1** Insert an Event-Based Random Number block into your model and configure it to indicate the distribution and parameters you want to use.
- 2** Insert and configure a Signal Latch block:
 - a** Set **Write to memory upon** to Sample time hit from port **wts**.
 - b** Set **Read from memory upon**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

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 14-4.

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

For examples using this block, see these sections:

- “Using Generation Times from a Vector” on page 1-11
- “Example: Counting Simultaneous Departures from a Server” on page 1-21
- “Example: Setting Attributes” on page 1-14

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:

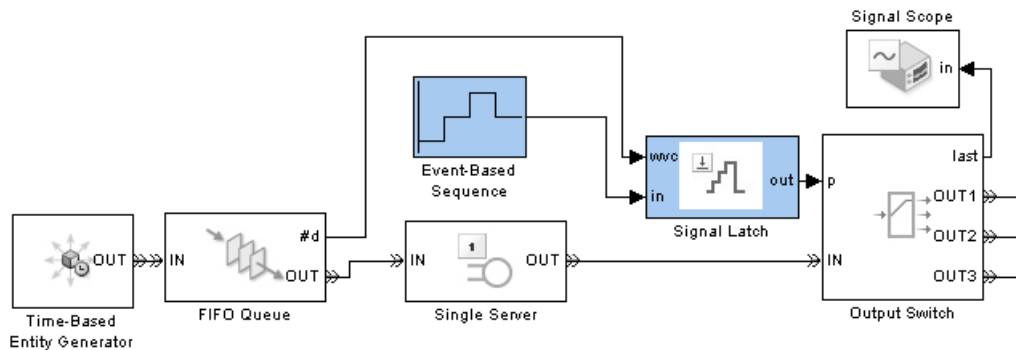
- 1 Insert an Event-Based Sequence block into your model and configure it to indicate the data you want to use.
- 2 Insert a Signal Latch block and set **Read from memory upon** to Write to memory event. The block no longer has an **rvc** signal input port.

- 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 3-5 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** to Holding final value, then the switch selects ports 1, 2, 3, 2, 2, 2, 2,... instead.

Understanding the Resolution Sequence for Input Signals

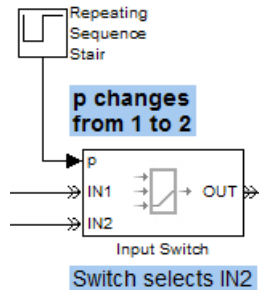
- “Detection of Signal Updates” on page 3-11
- “Effect of Simultaneous Operations” on page 3-12
- “Resolving the Set of Operations” on page 3-13
- “Using Event Priorities to Resolve Simultaneous Signal Updates” on page 3-13
- “Resolving Simultaneous Signal Updates Without Using Event Priorities” on page 3-15

Detection of Signal Updates

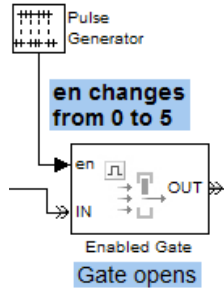
A block that possesses a reactive port listens for relevant updates in the input signal. Upon detecting a relevant update, the block reacts appropriately (for example, by opening a gate or executing a discrete event subsystem).

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



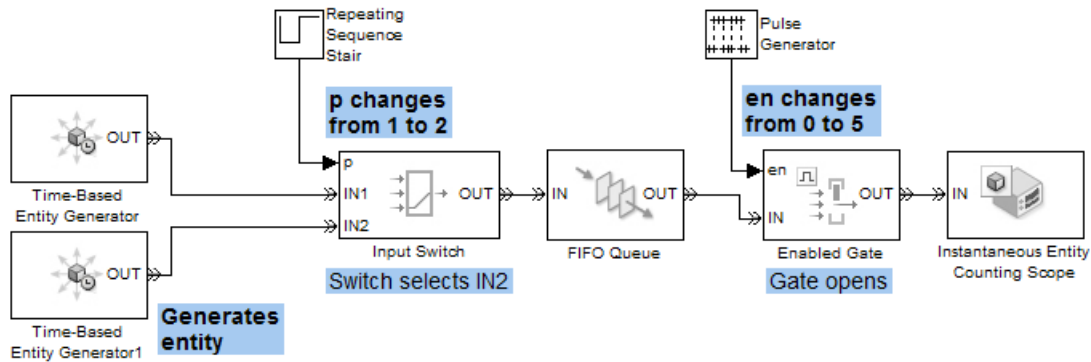
Signal Update That Causes a Gate to Open

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 port selection event is a logical consequence of the update of the **p** signal and the gate opening 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:

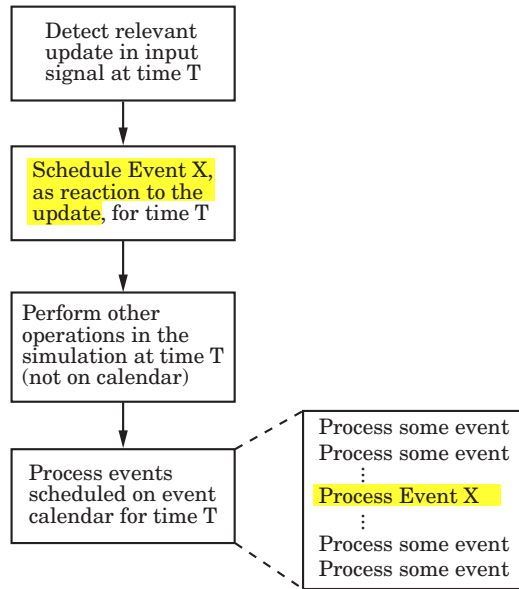
Dialog Box Choice	Description
<input type="checkbox"/> Resolve simultaneous signal updates according to event priority	This option lets you choose between two algorithms for the application to resolve the reactions to signal updates, relative to other simultaneous operations in the simulation.
<input checked="" type="checkbox"/> Resolve simultaneous signal updates according to event priority Event priority: <input type="text"/>	If you select this option, the algorithm relies on the relative values of a set of event priorities. You must set the event priority values in various blocks in the model.

Using 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.

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 When Using Event Priority

Contrast this with the schematic in Processing When Not Using Event Priority on page 3-16.

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 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. Simultaneous events having distinct event priorities are processed in ascending order of the event priority values. 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.

Learning More About Event Priorities and the Event Calendar

- For an example that examines the role of event priority values, assuming you have selected **Resolve simultaneous signal updates according to event priority**, see “Example: Race Conditions at a Switch” on page 2-25.
- To learn how to assign event priority values, see “Setting Event Priorities” on page 2-15.
- To learn how event priority values influence the processing sequence, see “Processing Sequence for Simultaneous Events” on page 2-11.
- To learn about the event calendar, see “Event Processing in SimEvents” on page 2-9.
- To learn what constitutes a relevant update at a reactive port, see “Reactive Ports” on page 14-6.

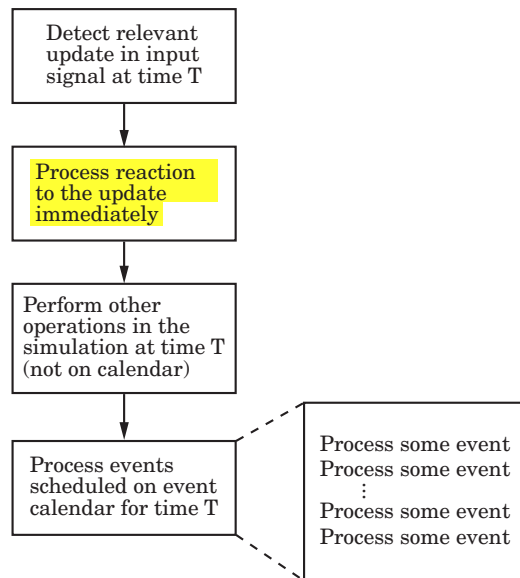
Resolving Simultaneous Signal Updates Without Using Event Priorities

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 upon detecting the update (shown as “immediately” in the schematic below). The reaction, such as a gate opening or an execution of a discrete event subsystem, is not deferred, does not appear on the event calendar, and has no event priority. As a result, you are not resolving the sequence explicitly.

To learn how the application processes simultaneous events that may or may not be on the event calendar, see “Processing Sequence for Simultaneous Events” on page 2-11.

Schematic Showing Application Processing

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 has a relevant update in its input signal at a given time, T.



Processing When Not Using Event Priority

Contrast this with the schematic in Processing When Using Event Priority on page 3-14.

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 3-11 and “Effect of Simultaneous Operations” on page 3-12	What simultaneous signal updates are, and the context in which the option is relevant
Behavior	“Using Event Priorities to Resolve Simultaneous Signal Updates” on page 3-13	How the simulation behaves when you select the option
	“Resolving Simultaneous Signal Updates Without Using Event Priorities” on page 3-15	How the simulation behaves when you do not select the option
Examples	“Events On and Off the Event Calendar” on page 2-31 and “Example: Using the Event Calendar to Prevent Interleaving” on page 14-14	Examples that illustrate the significance of the option
	“Example: Race Conditions at a Switch” on page 2-25	An example that examines the role of event priority values, assuming you have selected the option
Tips	“Choosing an Approach for Simultaneous Events” on page 2-14	Tips to help you decide how to configure your model

Update Sequence for Output Signals

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 3-18 and “Example: Detecting Changes from Empty to Nonempty” on page 9-24.

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.

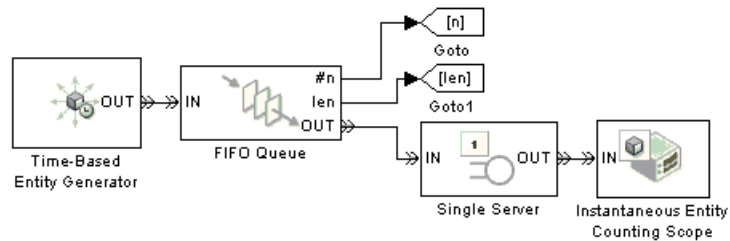
For example, the reference page for the N-Server block indicates that the **w** and **util** signals are updated in an arbitrary sequence relative to each other. The Order of Update column in the Signal Output Ports table lists the same number for both the **w** and **util** signals.

- 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.

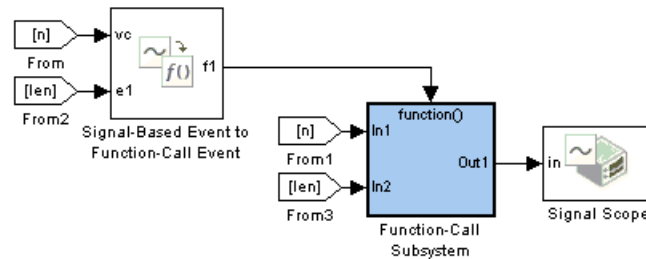
Example: Detecting Changes in the Last-Updated Signal

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**

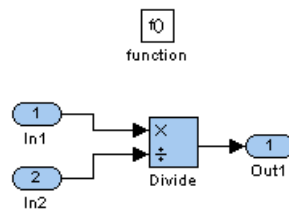
signals representing the current and average lengths, respectively. The computation of the ratio occurs in a function-call subsystem that is called when the Signal-Based Event to Function-Call Event block detects a change in **#n** (as long as **len** is positive, to avoid division-by-zero warnings). 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.

Multiple Simultaneous Updates of an Output Signal

Simultaneous events, which might be causally related to each other, are common in discrete-event simulation. This section describes how they affect output signals from SimEvents blocks. Common scenarios involving simultaneous events include the following:

- An entity completes its service and departs from a server, which permits an entity to arrive at the same time instant from a queue that precedes the server.
- An entity arrives at an empty queue, finds that the subsequent server block is also empty, and advances immediately to the server.
- An Enabled Gate block between a queue and an Entity Sink block changes from the closed state to the open state, which permits all entities in the queue to depart simultaneously.

This section defines zero-duration values and illustrates how you can detect them in your simulation. The topics are as follows:

- “Zero-Duration Values of Signals” on page 3-21
- “Importance of Zero-Duration Values” on page 3-22
- “Detecting Zero-Duration Values” on page 3-22

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 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: Status of Pending Entities in a Queue

Another example of zero-duration values is in “Plotting the Pending-Entity Signal”, which discusses a signal that indicates when the entity at the head of a queue is unable to depart. This signal becomes 0 if the entity at the head of the queue, previously unable to depart, finally departs. If the queue is left with other entities that cannot depart at this time, then the signal becomes 1 again. That is, the value of 0 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 3-21, 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

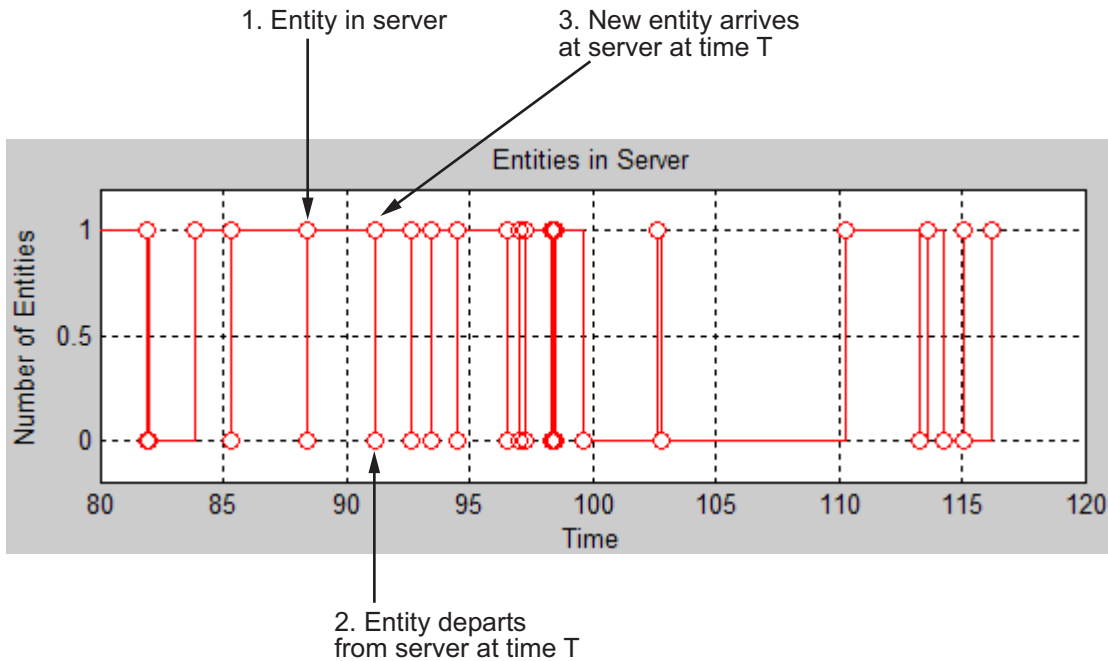
These topics describe ways to detect and examine zero-duration values:

- “Plotting Signals that Exhibit Zero-Duration Values” on page 3-22
- “Plotting the Number of Signal Changes Per Time Instant” on page 3-24
- “Viewing Zero-Duration Values in the MATLAB Workspace” on page 3-24

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 3-21.



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.

By contrast, some of the vertical lines have exactly one marker, meaning that the signal value at that time instant is unique. For example, near time 110, an arrival at the previously empty server is the only server-related activity at that time instant.

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 10-16.

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 value change** 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 10-14.

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 3-31 for instructions and an example.

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 9, “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 14-8.
- 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 3-17.
- 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 upon the arrival or departure of an entity, when the simulation ends, and when you pause the simulation. 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.

Manipulating Signals

The Signal Latch is a versatile block for manipulating event-based signals. You can use it to delay or resample signals based on events, not time. You can also use it to change the initial condition of event-based signals. The topics here are as follows:

- “Specifying Initial Conditions for Event-Based Signals” on page 3-27
- “Example: Resampling a Signal Based on Events” on page 3-28

In addition, see these examples:

- “Generating Random Event-Based Signals” on page 3-4
- “Generating Random Time-Based Signals” on page 3-6
- “Example: Detecting Collisions by Comparing Events” on page 2-40
- “Example: Compound Switching Logic” on page 5-7

Specifying Initial Conditions for Event-Based Signals

You can use the Signal Latch block to modify the value that an event-based signal assumes between the start of the simulation and the first relevant event. This is especially useful for output signals from Discrete Event Subsystem blocks and Stateflow blocks.

To modify the initial condition of an event-based signal without modifying the signal at other times, use this procedure:

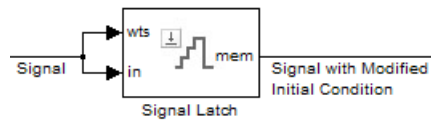
1 Set these parameters in the Signal Latch block:

- **Initial memory value** = your desired initial condition
- **Write to memory upon** = Sample time hit from port wts
- Select **Resolve simultaneous signal updates according to event priority** on the **Write** tab
- **Read from memory upon** = Write to memory event
- Select **Report memory value upon write event, mem**
- Clear **Report memory value upon read event, out**

The block now has signal input ports **wts** and **in**, and a signal output port **mem**.

- 2 Connect the signal whose initial condition you want to define to both the **in** and **wts** ports of the Signal Latch block.

The schematic below illustrates the resulting ports and connections of the Signal Latch block.



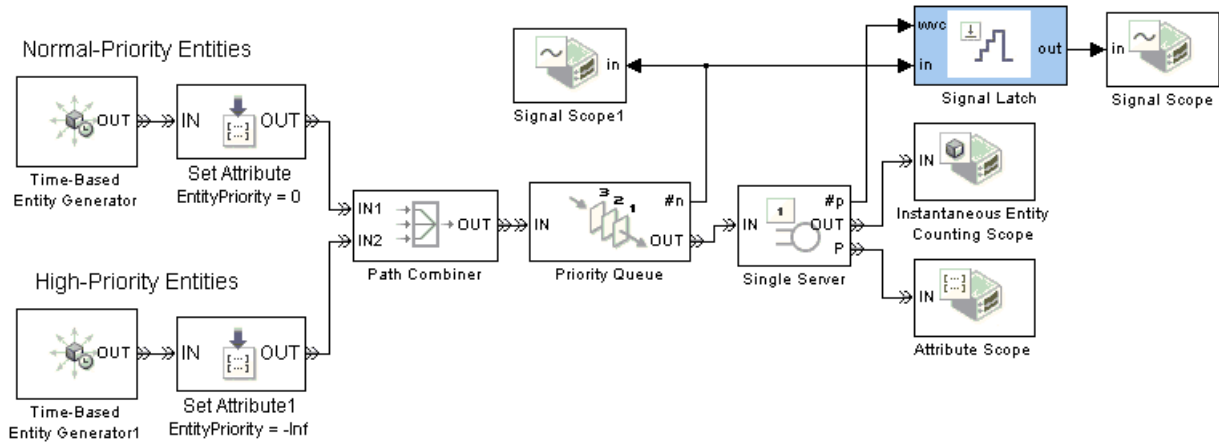
The Signal Latch block’s **mem** output signal uses your initial condition until your original signal has its first update. Afterward, the **mem** signal and your original signal are identical.

The following examples illustrate this technique:

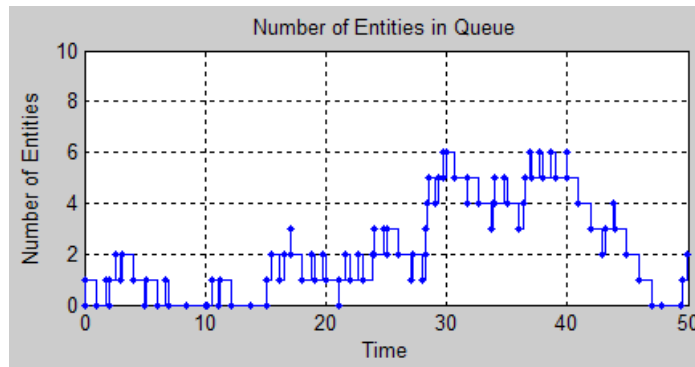
- “Example: Controlling Joint Availability of Two Servers” on page 7-4 initializes an event-based signal for use in a feedback loop.
- “Example: Failure and Repair of a Server” on page 4-17 initializes an event-based signal that is the output of a Stateflow block.

Example: Resampling a Signal Based on Events

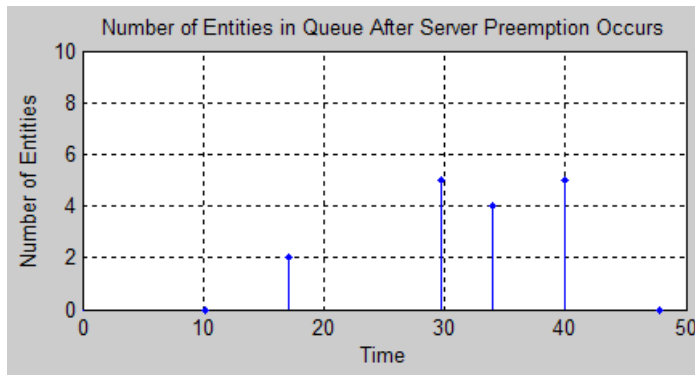
The example below contains a server that supports preemption of normal-priority entities by high-priority entities. This is similar to “Example: Preemption by High-Priority Entities” on page 4-11. Suppose that a preemption and the subsequent service of a high-priority entity represents a time interval 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

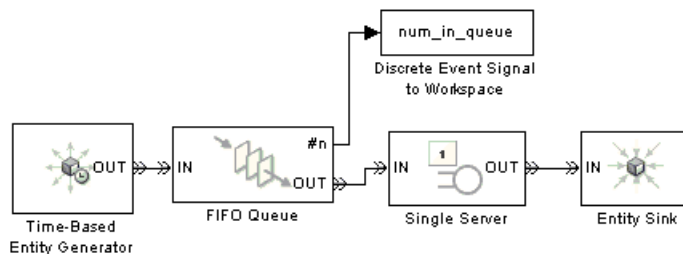
- “Example: Sending Queue Length to the Workspace” on page 3-31
- “Using the To Workspace Block with Event-Based Signals” on page 3-34

The Discrete Event Signal to Workspace block writes event-based signals to the MATLAB workspace when the simulation stops or pauses. 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 MATLAB workspace during a discrete-event simulation, see “Using Data Sets to Create Event-Based Signals” on page 3-9.

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 times 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 below 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 9-22.
- 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 9-7.

Modeling Queues and Servers

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

Using a LIFO Queuing Discipline
(p. 4-2)

Sorting by Priority (p. 4-4)

Preempting an Entity in a Server
(p. 4-10)

Modeling Multiple Servers (p. 4-13)

Modeling the Failure of a Server
(p. 4-15)

Comparing LIFO and FIFO queues

Using attribute values to control the queue discipline

Enabling an entity to replace a lower priority entity in a server

Modeling a bank of servers

Using Stateflow to model the behavior of a server that might require maintenance

Using a LIFO Queuing Discipline

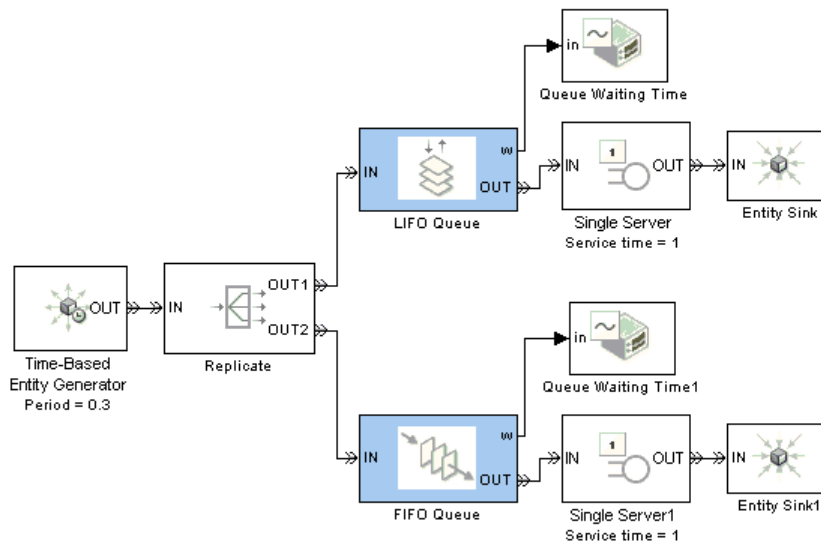
The LIFO Queue block supports the last-in, first-out (LIFO) queuing discipline. The entity that departs from the queue at a given time is the most recent arrival. You can interpret a LIFO queue as a stack.

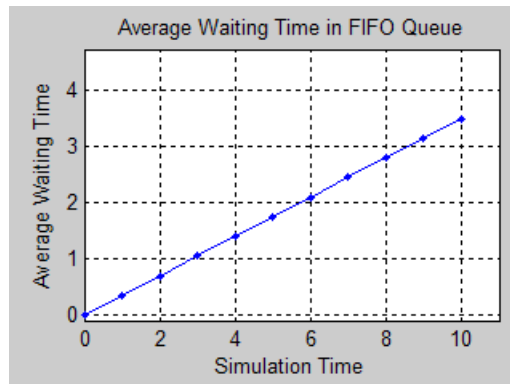
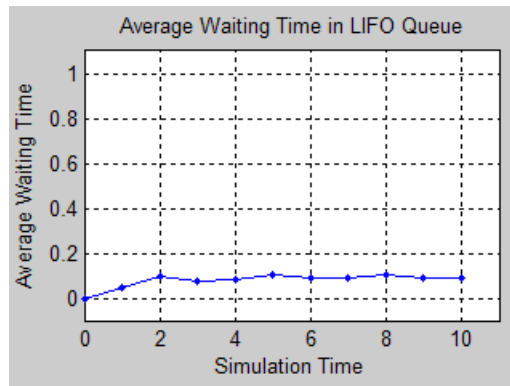
Some ways to see the difference between FIFO and LIFO queuing disciplines are to

- 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 1-13.
- 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.





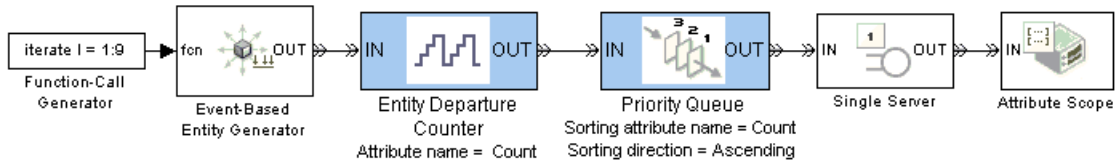
Sorting by Priority

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 1-13.

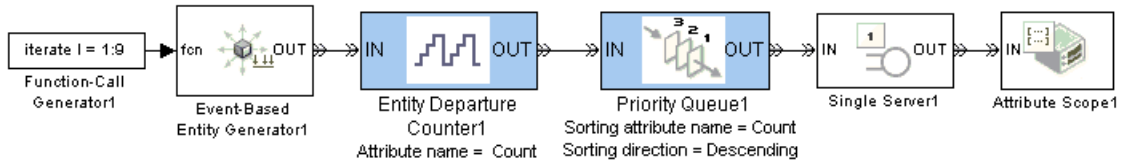
Note While you can view the value of the sorting attribute as an entity priority, this value has nothing to do with event priorities or block priorities.

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.

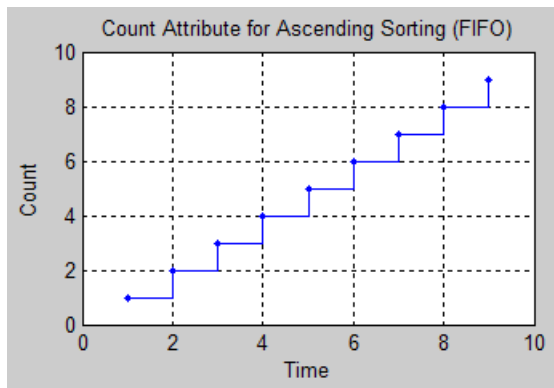
Priority Queue acts like FIFO Queue

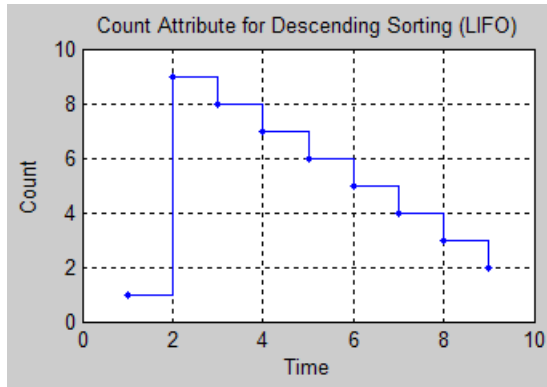


Priority Queue acts like LIFO Queue



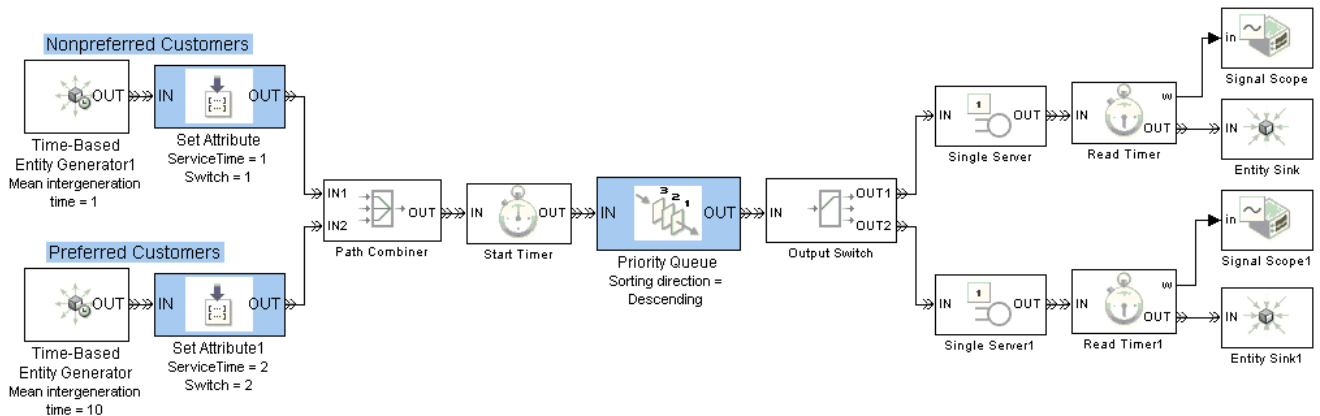
The FIFO plot reflects an increasing sequence of Count values. The LIFO plot reflects a descending sequence of Count values, except for the Count=1 entity that advances to the server before the queue has any other entities to sort.



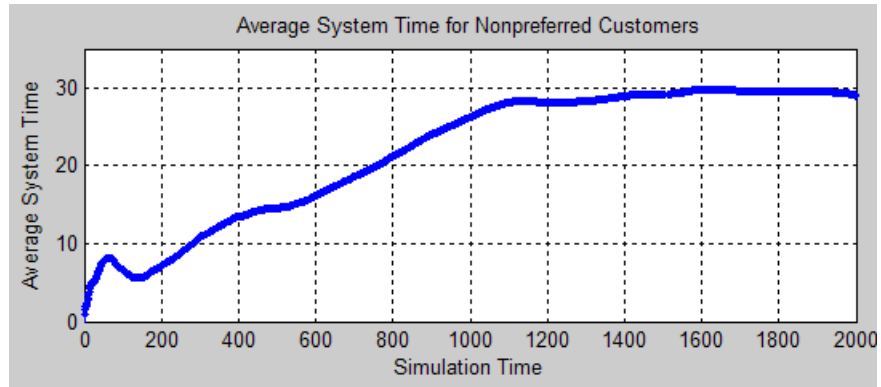


Example: Serving Preferred Customers First

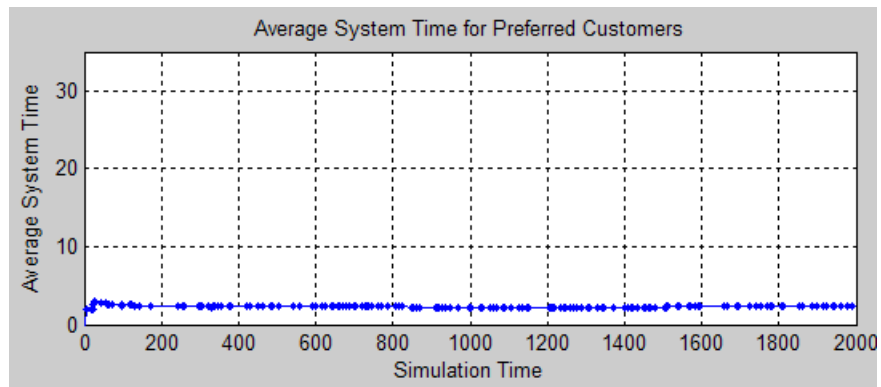
In the example below, two types of customers enter a queuing system. One type, considered to be preferred customers, are less common but require longer service. The priority queue places preferred customers ahead of nonpreferred customers. The model plots the average system time for the set of preferred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



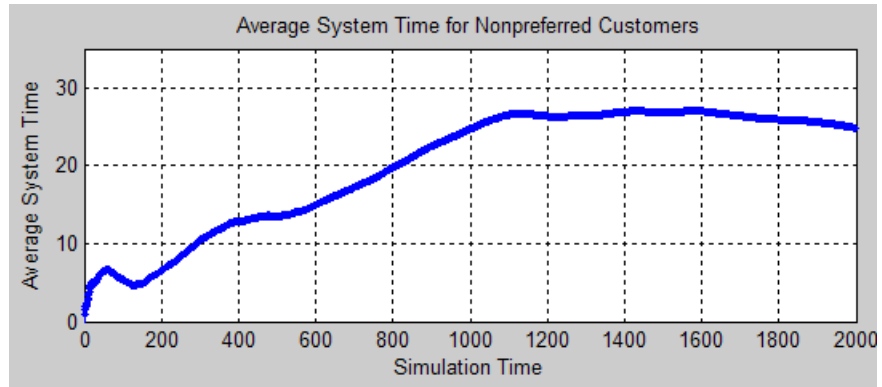
Average System Time for Nonpreferred Customers Sorted by Priority



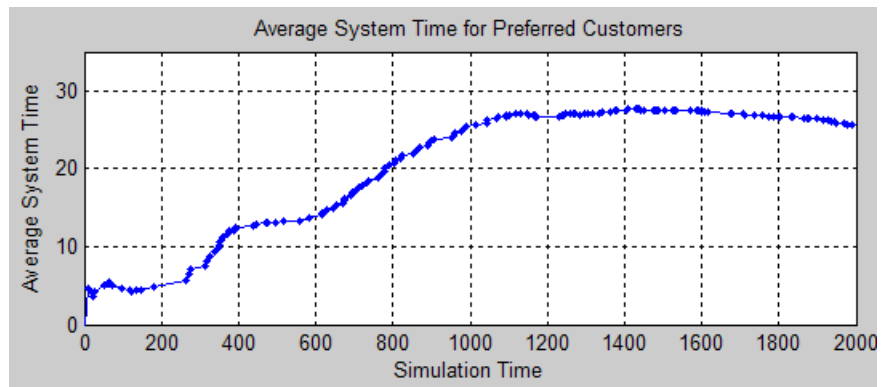
Average System Time for Preferred Customers Sorted by Priority

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

- “Criteria for Preemption” on page 4-10
- “Residual Service Time” on page 4-10
- “Queuing Disciplines for Preemptive Servers” on page 4-11
- “Example: Preemption by High-Priority Entities” on page 4-11

The Single Server block supports preemption, which is the replacement of an entity in a server block by an entity that satisfies certain criteria. The preempted entity departs from the block via the **P** entity output port instead of 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 and the orientation of the comparison using the **Sorting attribute name** and **Sorting direction** parameters in the Single Server block’s dialog box. (These 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 1-13. 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 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.

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.

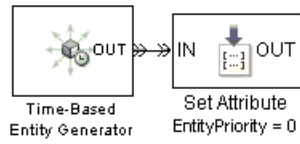
Example: Preemption by High-Priority Entities

The example below 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 -Inf go to the head of the priority queue and immediately preempt any entity in the server except another entity whose sorting attribute value is -Inf.

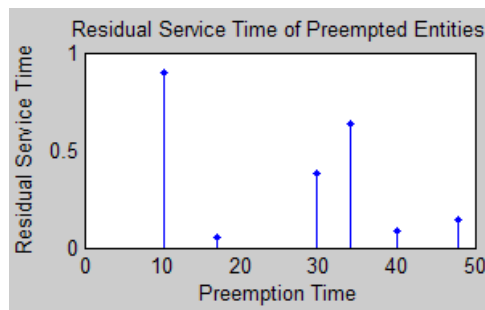
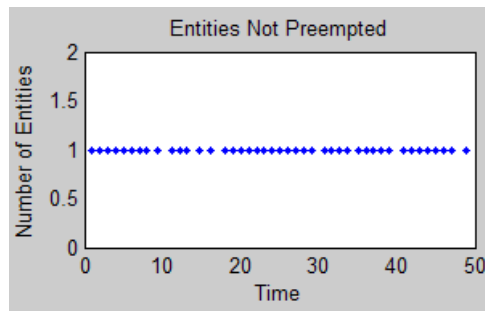
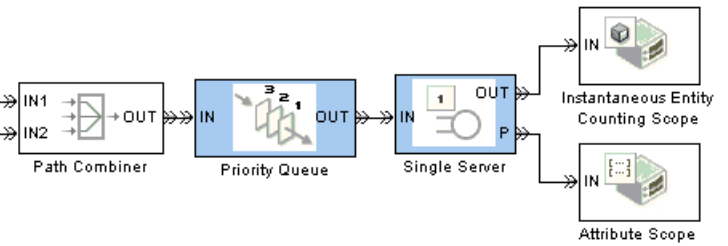
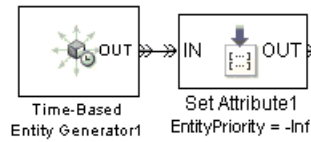
One plot shows when nonpreemptive departures occur, while another plot indicates the residual service time whenever preemptive departures occur.

4 Modeling Queues and Servers

Normal-Priority Entities



High-Priority Entities



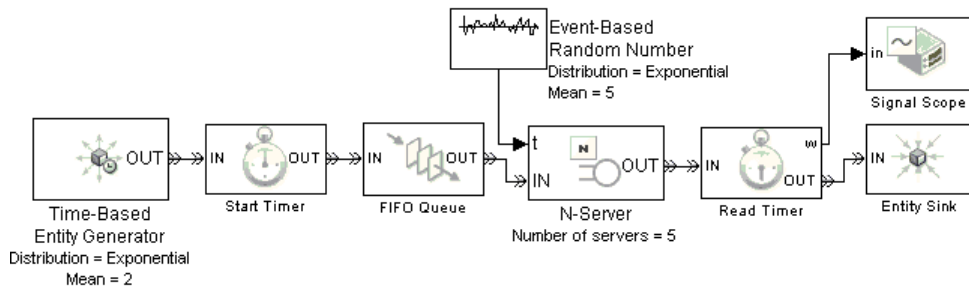
Modeling Multiple Servers

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.

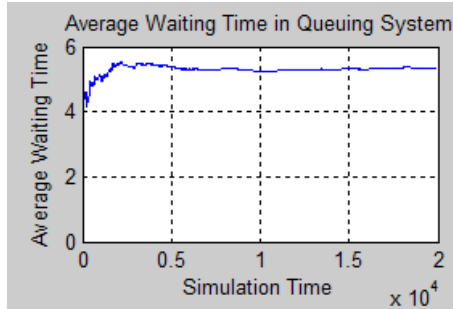
To model multiple servers that are not identical to each other, you must use multiple blocks. For example, to model a pair of servers whose service times do not share the same distribution, use a pair of Single Server blocks rather than a single N-Server block. The example in “Example: Selecting the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



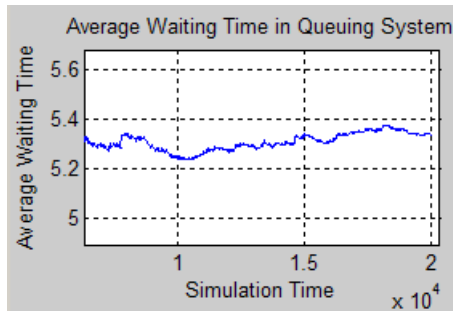
You can compare the empirical values shown in the plot with the theoretical value, $E[S]$, of the mean system time for an $M/M/m$ queuing system with an arrival rate of $\lambda=1/2$ and a service rate of $\mu=1/5$. Using expressions in [2], the computation is as follows.

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

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

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

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



Modeling the Failure of a Server

In 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 topics are as follows:

- “Server States” on page 4-15
- “Using a Gate to Implement a Failure State” on page 4-15
- “Using Stateflow to Implement a Failure State” on page 4-16

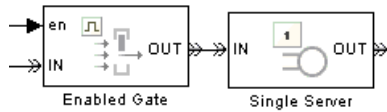
Server States

The server blocks in SimEvents 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 6, “Using Logic” or using Stateflow 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 7-4. 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 to Implement a Failure State

Stateflow is a suitable tool 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.

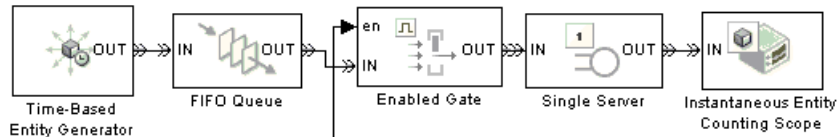
When modeling interactions between the state chart and discrete-event aspects of the model, note that a function call is the recommended way to make Stateflow blocks respond to asynchronous state changes. You can use blocks in the Event Generators and Event Translation libraries to produce a function 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.

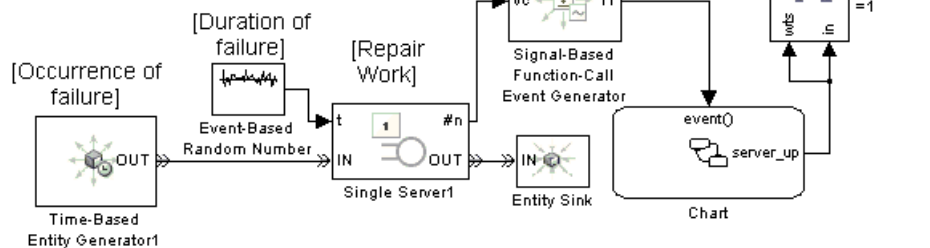
Example: Failure and Repair of a Server

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.

Entities representing customers in queuing system



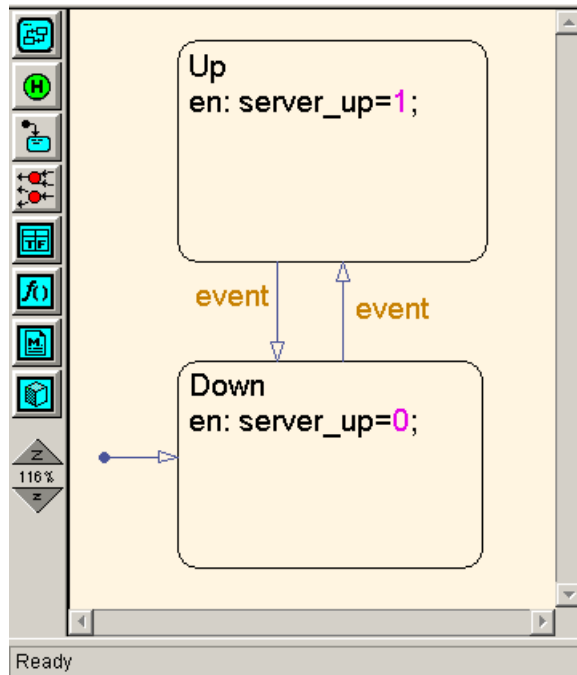
Entities representing failure and repair of server in queuing system



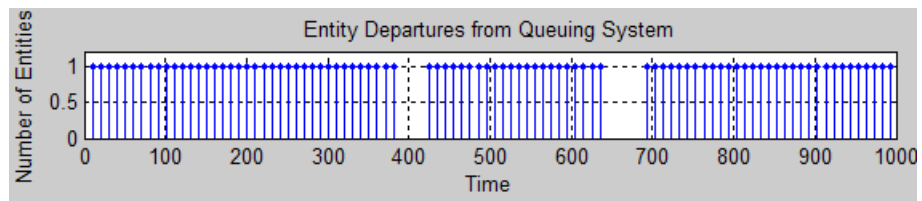
The lower portion of the model contains a parallel queuing system. The entities in the lower queuing system represent failures, not customers. Generation of a failure entity represents a failure occurrence in the upper queuing system. Service of a failure entity represents the time during which the server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

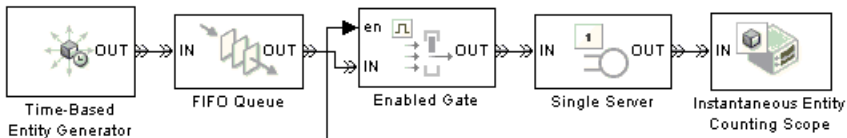


Although this two-state machine could be modeled more concisely with a Signal Latch block instead of a Stateflow block, the Stateflow chart scales more easily to include additional states or other complexity.

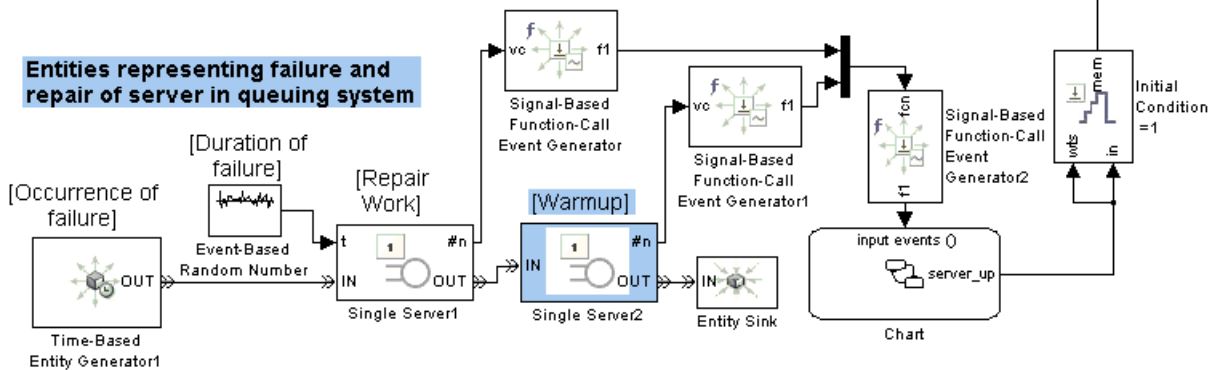
Example: Adding a Warmup Phase

The example below modifies the one in “Example: Failure and Repair of a Server” on page 4-17 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.

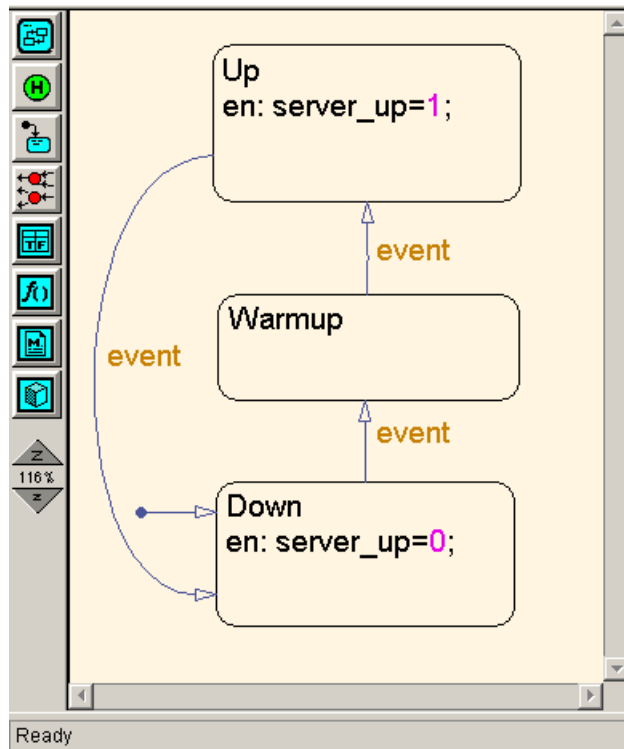
Entities representing customers in queuing system



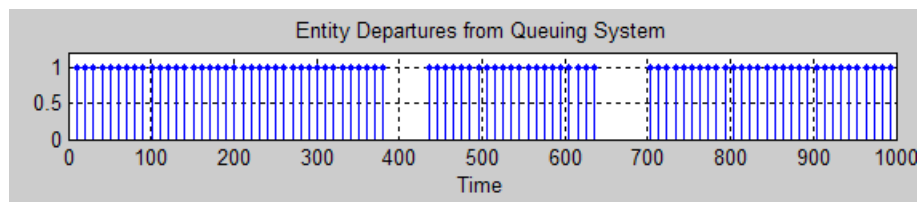
Entities representing failure and repair of server in queuing system



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, Stateflow 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 4-17, 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 getting started documentation.

Output Switching Based on a Signal
(p. 5-2)

Ensuring accurate timing for
signal-based output switching

Example: Cascaded Switches with
Skewed Distribution (p. 5-6)

Random switching using cascaded
switch blocks

Example: Compound Switching
Logic (p. 5-7)

Combination of round-robin and
random switching

Output Switching Based on a Signal

- “Specifying an Initial Port Selection” on page 5-2
- “Using the Storage Option to Prevent Latency Problems” on page 5-2

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 6-3 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.
- 2** Temporarily yields control to blocks in the model along the entity's path. For example, this might give other blocks a chance to update the signal that connects to the **p** port.
- 3** Determines which entity output port is the selected port.
- 4** 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 8, "Forcing Departures Using Timeouts" in the user guide documentation.

Note 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 5-5.

Even if the stored entity departs at the same time that it arrives, step 3 temporarily yields control to blocks in the model along the entity’s path. For example, this might give other blocks a chance to update the signal that connects to the **p** port. Step 3 on page 3 is important for preventing latency.

Example Using Storage. The model in “Example: Choosing the Shortest Queue” on page 6-3 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.
- 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 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.
- 4 Outputs the entity using the up-to-date value of the **p** signal.

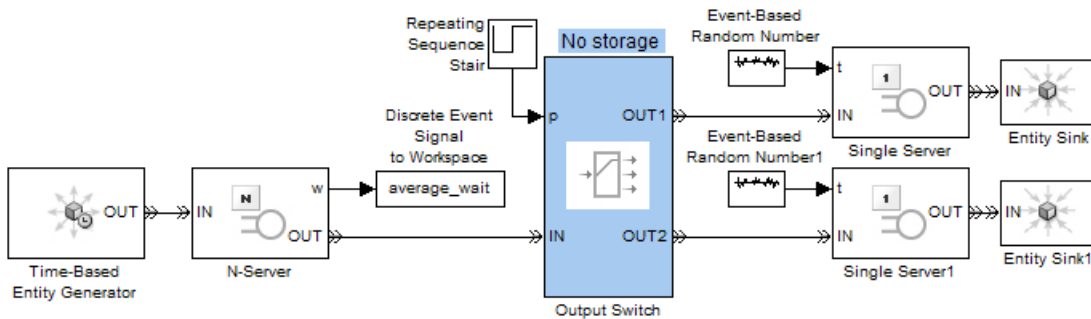
Effect of Disabling Storage

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 13-17 in the 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**.



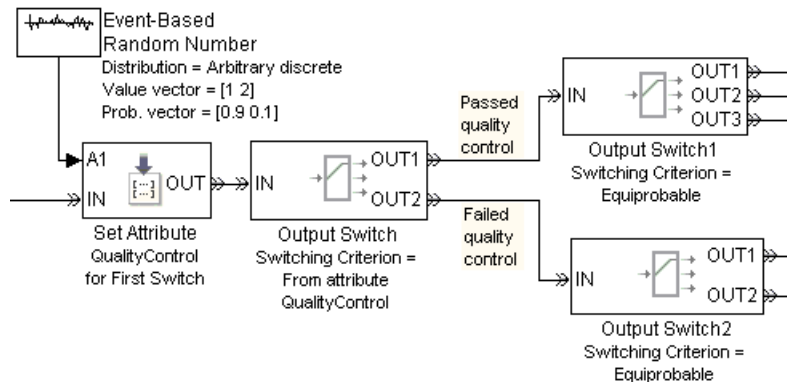
Furthermore, storage in the switch is probably undesirable in this model. Storing entities in the Output Switch block for a nonzero period of time would affect the computation of average wait, which is the Infinite Server block's **w** output signal. If the goal is to compute the average waiting time 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 decisionmaking 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

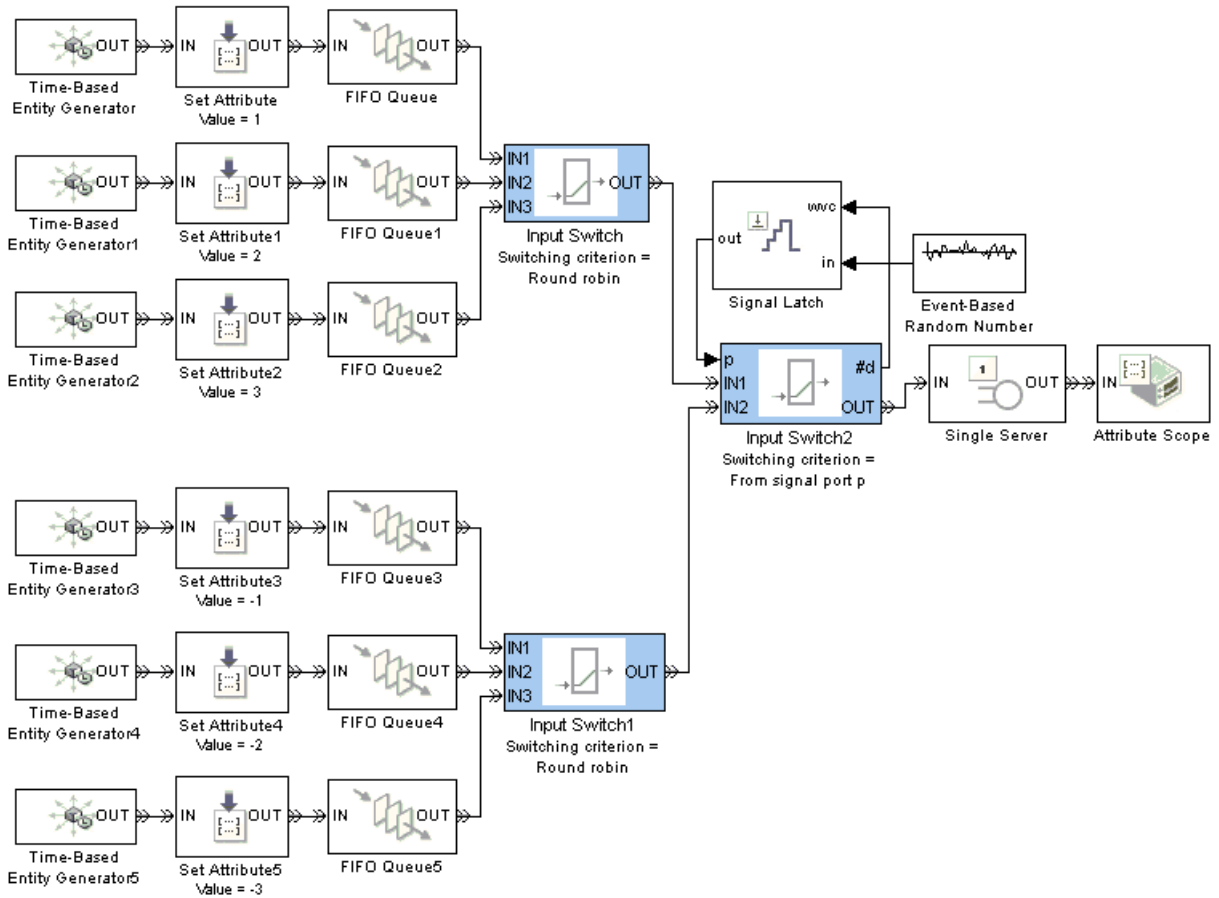
The figure below illustrates the switches and their switching criteria.



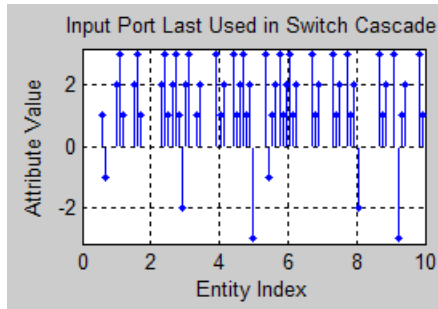
Example: Compound Switching Logic

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 (p. 6-2)	Typical situations in which logic affects simulation behavior
Using Embedded MATLAB Function Blocks for Logic (p. 6-3)	Specifying logic using MATLAB code
Using Logic Blocks (p. 6-10)	Specifying logic using a block diagram

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

If your logic algorithm is easier to express in MATLAB 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. This section provides examples that are particularly relevant for logic in SimEvents models:

- “Example: Choosing the Shortest Queue” on page 6-3
- “Example: Varying Fluid Flow Rate Based on Batching Logic” on page 6-6

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, see

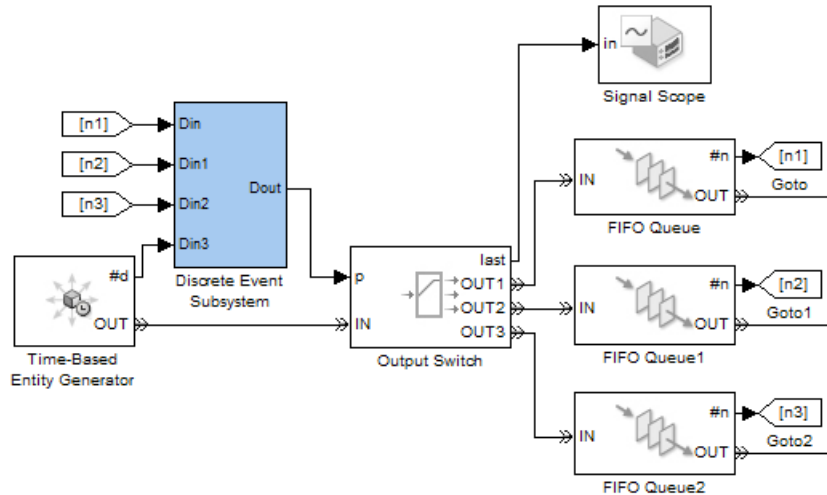
- The Switching Logic subsystems in the Astable Multivibrator Circuit demo.
- “Example: Computing a Time Average of a Signal” on page 11-10

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.

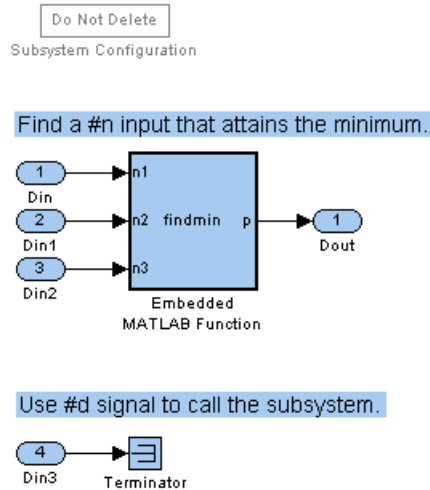
Example: Choosing the Shortest Queue

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 5-2.

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

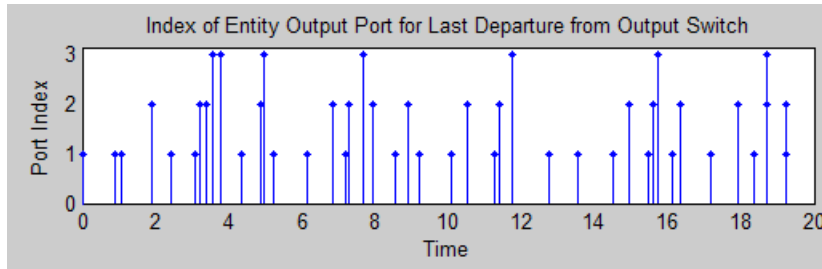
If you double-click an Embedded MATLAB Function block in a model, an editor window shows the MATLAB function that specifies the block. In this example, the following MATLAB function computes the index of a queue having the shortest length, where the individual queue lengths are n_1 , n_2 , and n_3 . 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.
[minlength,p] = min([n1 n2 n3]);
```

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

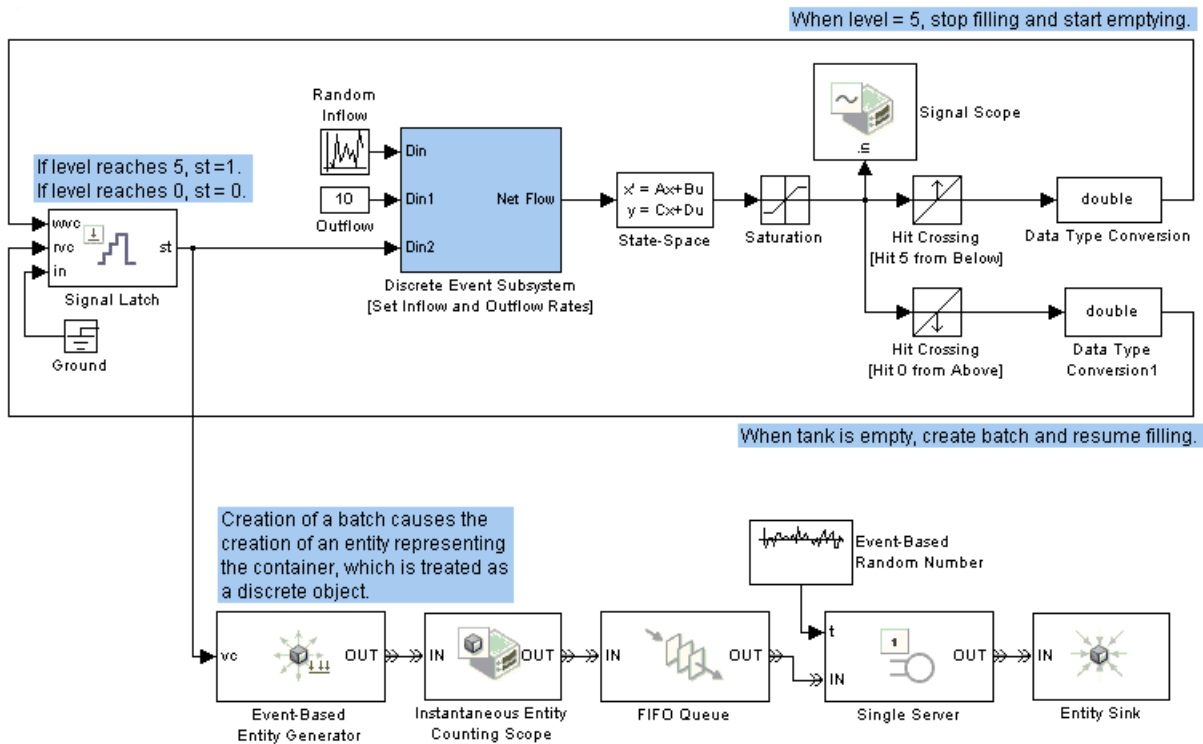
The figure below shows a sample plot. Each stem corresponds to an entity departing from the switch block via one of the three entity output ports.



For a variation on this model that uses logic blocks instead of the Embedded MATLAB Function block, see “Example: Choosing the Shortest Queue Using Logic Blocks” on page 6-16.

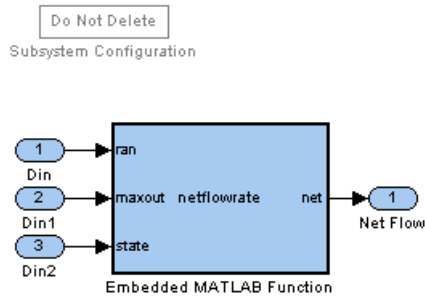
Example: Varying Fluid Flow Rate Based on Batching Logic

The model below represents a batching process in which a tank accumulates fluid up to a desired level of 5. When the level reaches 5, the tank switches to an emptying mode, modeling the creation of a batch. When the tank is empty, an entity generator creates an entity that represents the container for the batch of fluid. Batching logic determines whether the tank is filling or emptying.



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

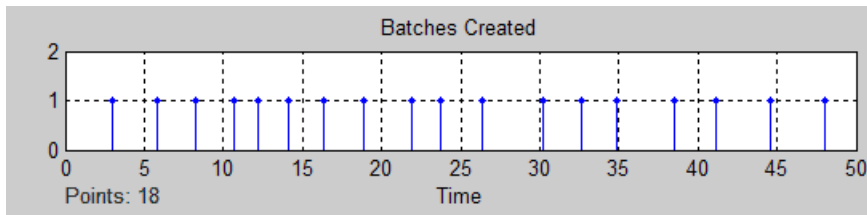
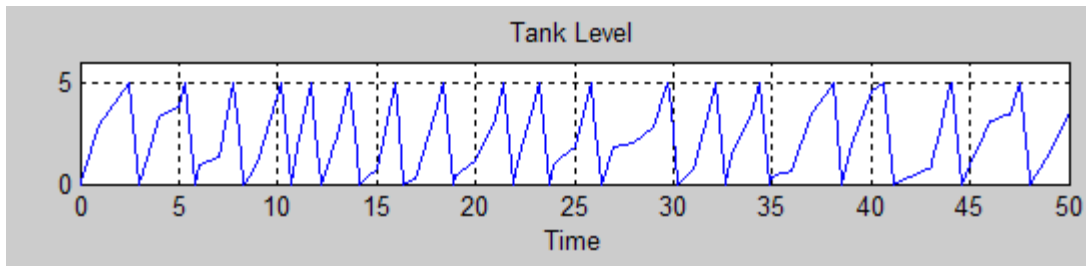
```
function net = netflowrate(ran,maxout,state)

% 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

- “Example: Using Servers in Shifts” on page 6-11
- “Example: Choosing the Shortest Queue Using Logic Blocks” on page 6-16

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

See these examples:

- “Example: Using Servers in Shifts” on page 6-11
- “Example: Choosing the Shortest Queue Using Logic Blocks” on page 6-16

- The logic diagrams depicted in “Stopping Upon Reaching a Particular State” on page 11-36
- The gate examples, especially “Example: Controlling Joint Availability of Two Servers” on page 7-4

Example: Using Servers in Shifts

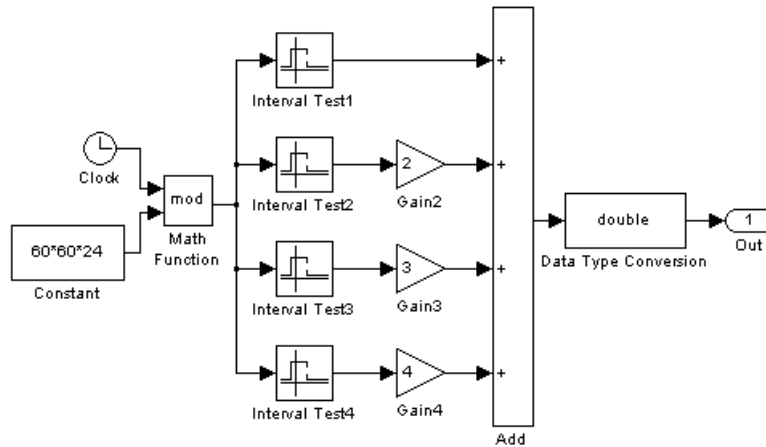
Suppose you have four servers that operate in mutually exclusive shifts of unequal lengths, and you want to direct each entity to the server that is currently on duty. Suppose that the server on duty has the following index between 1 and 4:

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

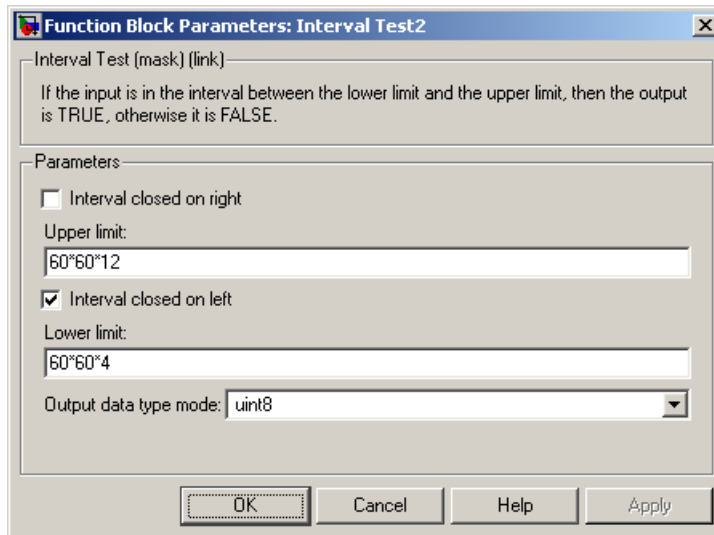
Below are two methods of computing this index in a subsystem. One method uses both logical and numerical blocks, while the other method is purely numerical.

Index Computation 1

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.



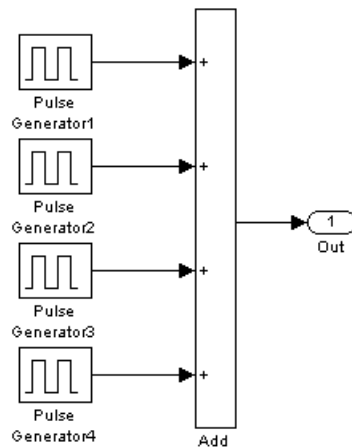
For example, the second shift is represented by the second Interval Test block, whose dialog box is shown below.



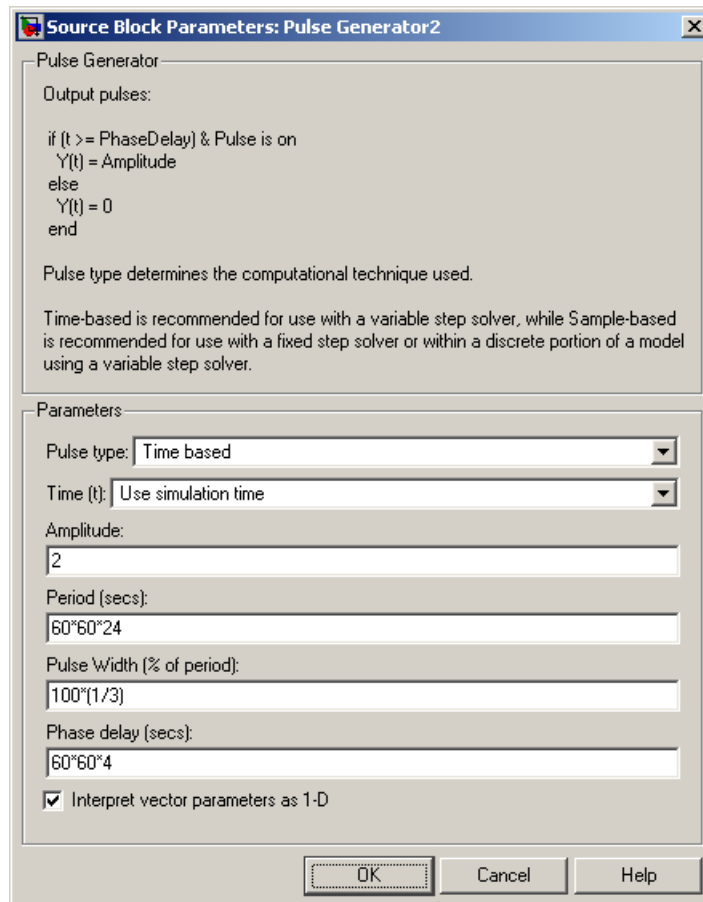
Index Computation 2

Alternatively, you can compute the index of the server on duty using a subsystem like the one shown below, where each Pulse Generator block assumes the corresponding index value when that server is on duty, and assumes the value 0 at other times. In particular,

- **Period** = $60 \times 60 \times 24$ for all Pulse Generator blocks, to represent a daily cycle in seconds
- **Amplitude** gives the index for each server, between 1 and 4
- **Pulse width** gives the length of each server's shift
- **Phase delay** gives the starting time of each server's shift

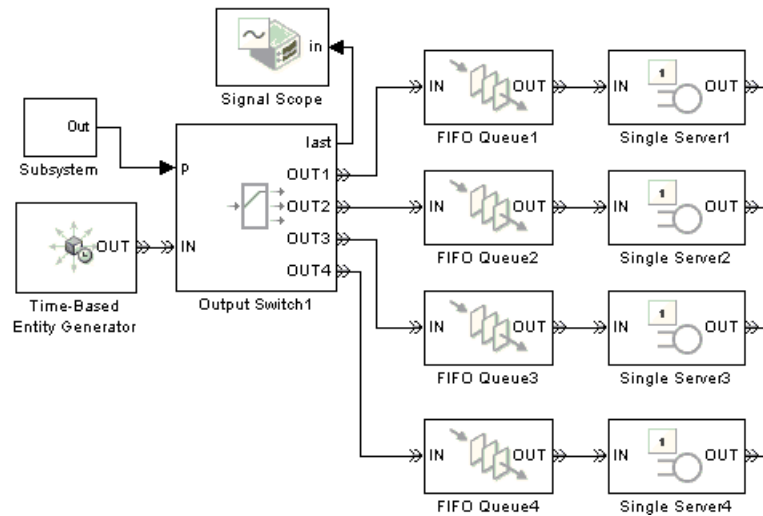


For example, the second shift is represented by the second Pulse Generator block, whose dialog box is shown below.

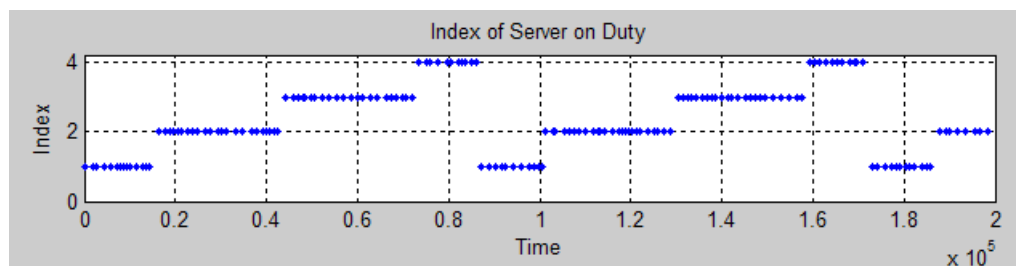


Top-Level Model

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 6-16 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.

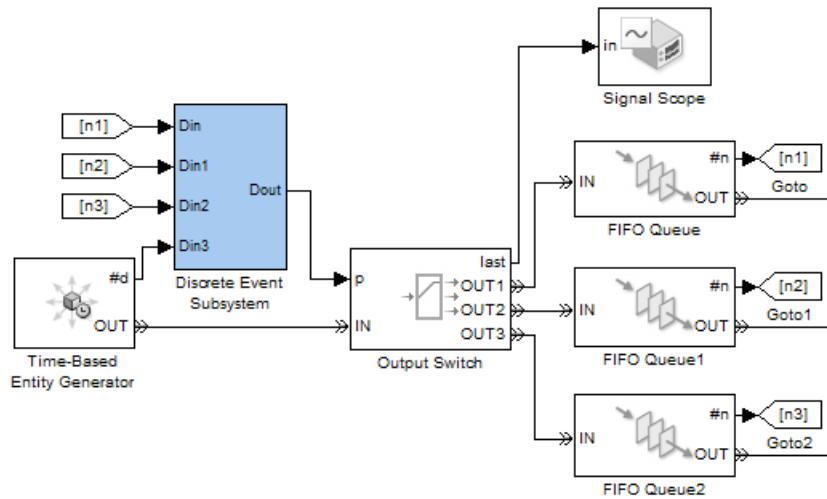


The sample plot below reflects the use of shifts. Each plotting marker corresponds to an entity departing from the switch block via one of the four entity output ports.



Example: Choosing the Shortest Queue Using Logic Blocks

This example, a variation on the model in “Example: Choosing the Shortest Queue” on page 6-3, 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.

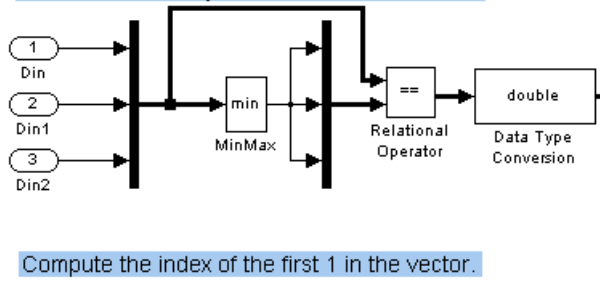


Top-Level Model

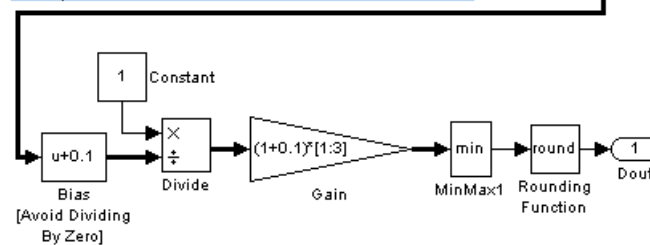
Do Not Delete

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

Regulating Arrivals Using Gates

Role of Gates in SimEvents Models (p. 7-2)	What gates represent in various models
Keeping a Gate Open Over a Time Interval (p. 7-4)	Admitting entities during a time interval of arbitrary length
Opening a Gate Instantaneously (p. 7-6)	Admitting an entity when an event occurs
Using Logical Combinations of Gates (p. 7-9)	Implementing compound logic for entity admission

Role of Gates in SimEvents Models

- “Accessing Gate Blocks” on page 7-3
- “Types of Gates” on page 7-3

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 7-4.

- 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 7-6.

- 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 “Using Logical Combinations of Gates” on page 7-9.

To learn about implementing logic that determines when a gate is open or closed, see Chapter 6, “Using Logic”.

Accessing Gate Blocks

The gate blocks reside in the Gates library of SimEvents.

A gate block forbids or permits entities to advance from the block before the gate to the block after the gate. For example, if you want to control advancement from a queue to a server, then place the gate block after the queue and before the server. Many models follow a gate with a storage block, such as a queue or server.

Types of Gates

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 7-4.
- 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 7-6.

Keeping a Gate Open Over a Time Interval

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

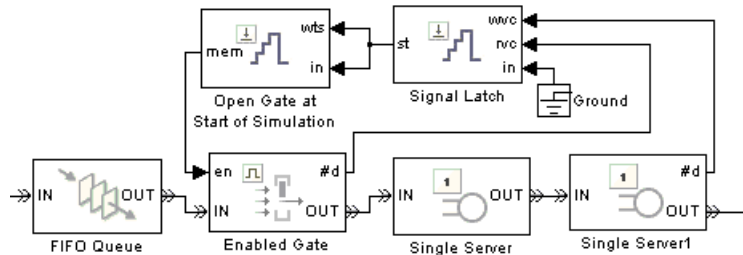
Suppose that each entity undergoes two processes, one at a time, and that the first process does not start if the second process is still in progress for the previous entity. Assume for this example that it is preferable to model the two processes using two Single Server blocks in series rather than 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. The block labeled Open Gate at Start of Simulation is another Signal Latch block; its purpose is to modify the **st** signal only by defining an initial condition of 1 (using the technique described in "Specifying Initial Conditions for Event-Based Signals" on page 3-27). 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

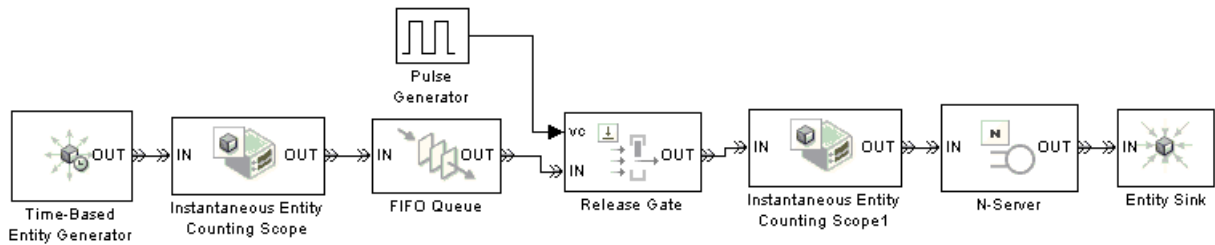
- “Example: Synchronizing Service Start Times with the Clock” on page 7-6
- “Example: Opening a Gate Upon Entity Departures” on page 7-7

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.

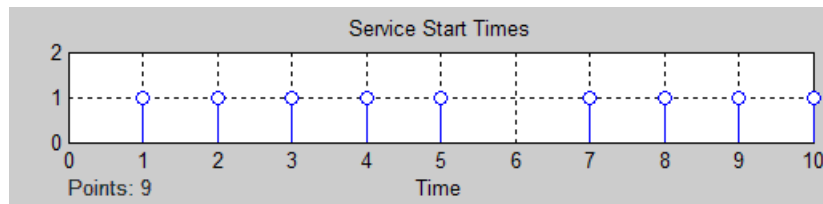
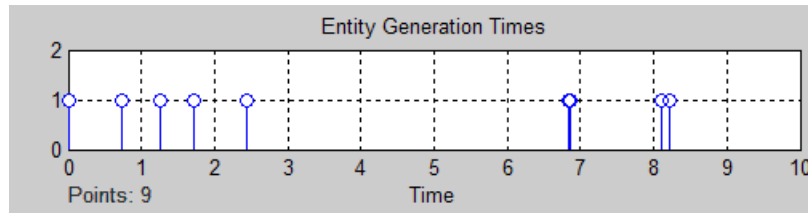
An entity passing through a gate must already be pending before the gate-opening event occurs. Suppose a Release Gate block follows a Single Server block and a gate-opening event is scheduled simultaneously with a service completion event. If the gate-opening event is processed first, then the gate opens 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 “Setting Event Priorities” on page 2-15.

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.)

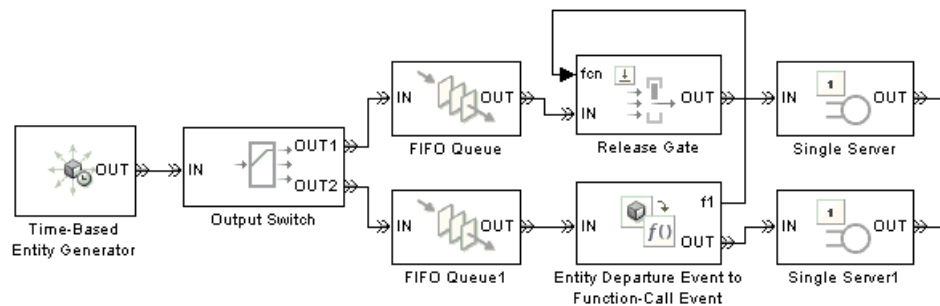


The plots below show that the entity generation times can be noninteger values, but the service beginning times are always integers.



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.

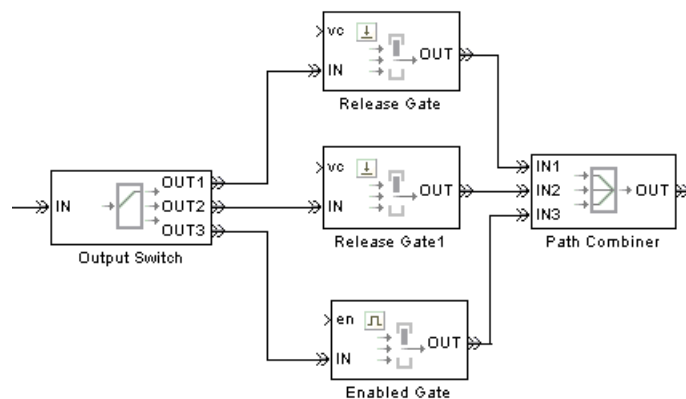
Using Logical Combinations of Gates

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.

Note You should 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.



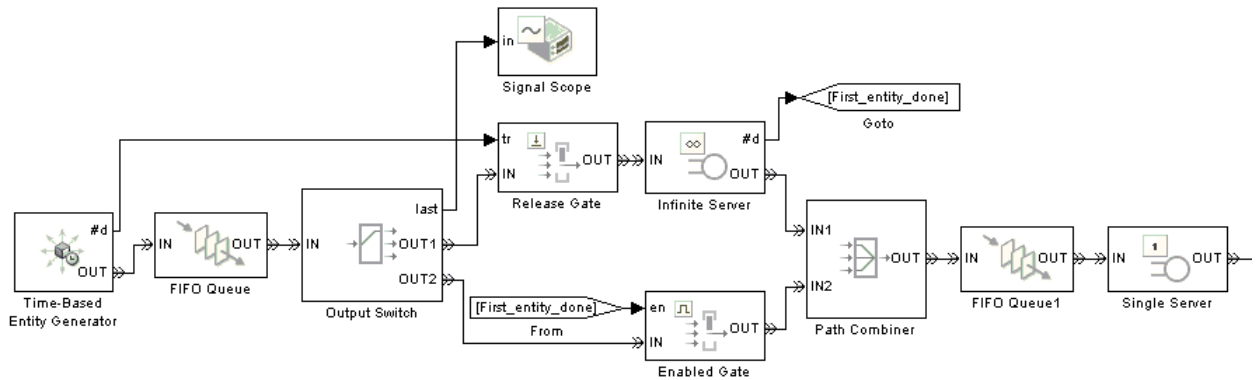
Each entity attempts to arrive at the first gate; if it is closed, the entity attempts to arrive at the second gate, and so on. If all gates are closed, then the Output Switch block's entity input port is unavailable and the entity must stay in a preceding block (such as a queue or server preceding the 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 “Setting Event Priorities” on page 2-15.

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.

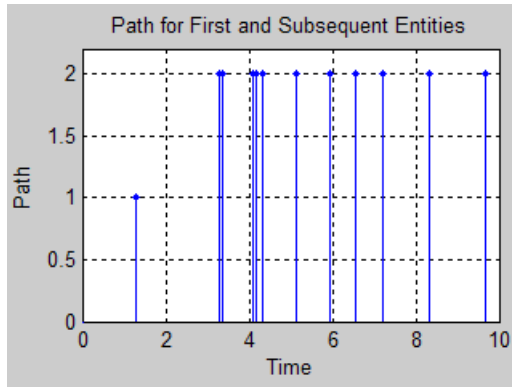


The Release Gate block is open precisely when the **#d** output signal from the Time-Based Entity Generator block rises from 0 to 1. That is, the gate is open for the first entity of the simulation and no other entities. The first entity arrives at an Infinite Server block, which represents the warmup 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 (p. 8-2)

Basic Example Using Timeouts (p. 8-3)

Basic Procedure for Using Timeouts (p. 8-4)

Defining Entity Paths on Which Timeouts Apply (p. 8-7)

Handling Entities That Time Out (p. 8-10)

Example: Limiting the Time Until Service Completion (p. 8-14)

Terminology and typical uses of timeouts

Prototypical arrangement of timeout blocks

Typical procedure for incorporating timeout events into a model

Limiting the time on linear or nonlinear entity paths

Configuring storage blocks to react to timeouts

Canceling a timeout event upon service completion

Role of Timeouts in SimEvents Models

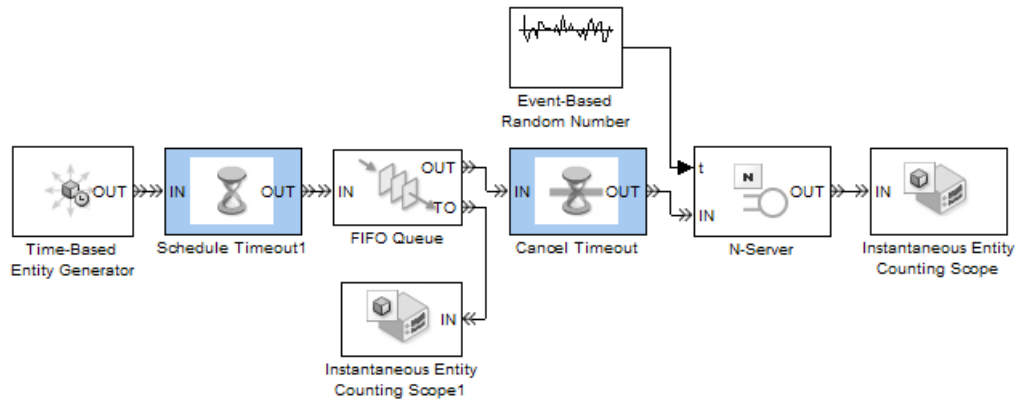
You can limit the amount of time an entity spends during the simulation on designated entity paths. Exceeding the limit causes a *timeout event* and the entity is said to have *timed out*. The duration of the time limit is called the *timeout interval*.

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:

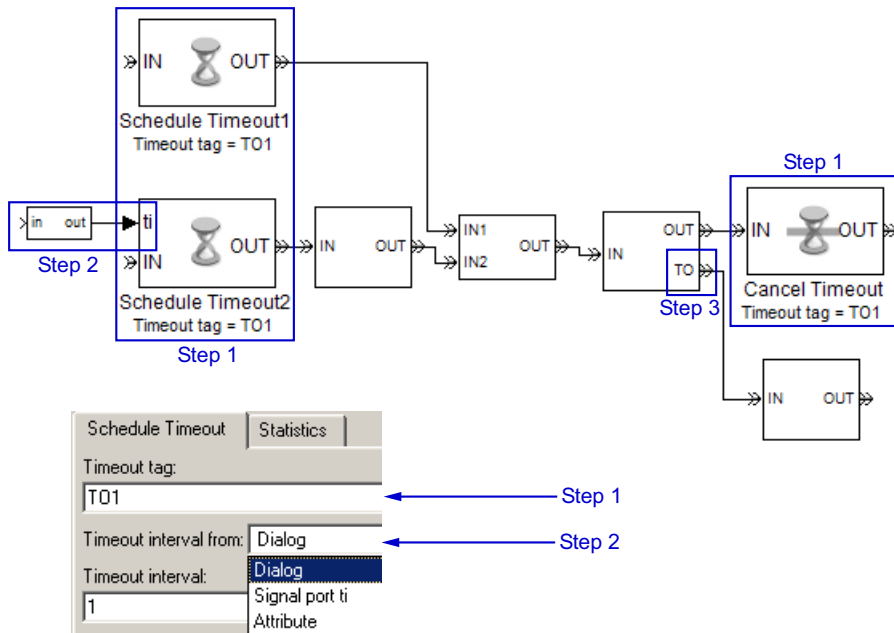
- **Entity Times Out** — If the customer is still in the queue when the clock reaches the time limit, the customer hangs up without reaching an operator. In generic terms, the entity times out, departs from the FIFO Queue block via the **TO** port, and does not reach the server.
- **Entity Advances to 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

A typical procedure for incorporating timeout events into your model is as follows:

Step	Description
Step 1	Designate the entity path on which you want to limit entities' time.
Step 2	Specify the timeout interval.
Step 3	Specify destinations for timed-out entities.

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 8-7.

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 14-8.

- 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 13-17.

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 8-10.

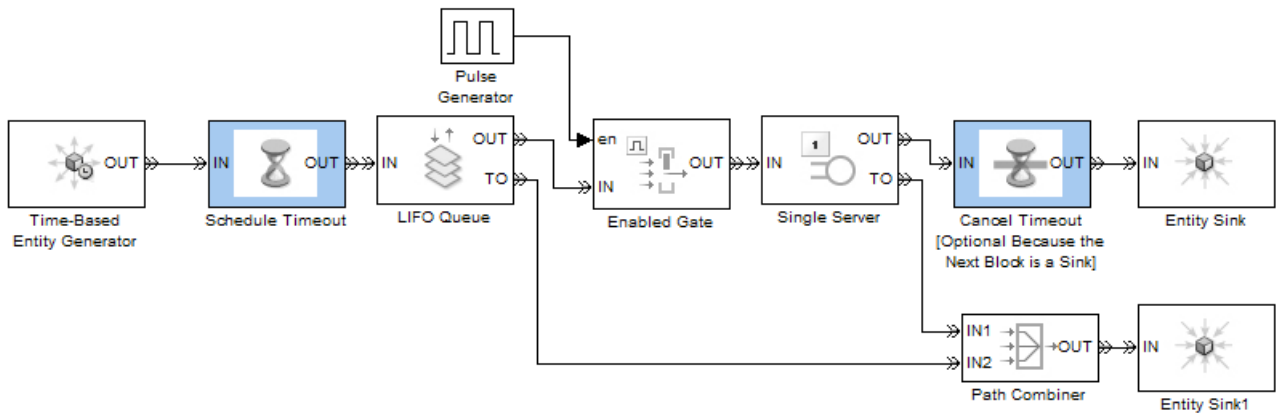
Defining Entity Paths on Which Timeouts Apply

This section illustrates sample topologies for the entity paths on which entities have limited time. Notice the relative positions of Schedule Timeout, Cancel Timeout, and other blocks. The topics are

- “Linear Path for Timeouts” on page 8-7
- “Branched Path for Timeouts” on page 8-8
- “Feedback Path for Timeouts” on page 8-8

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.

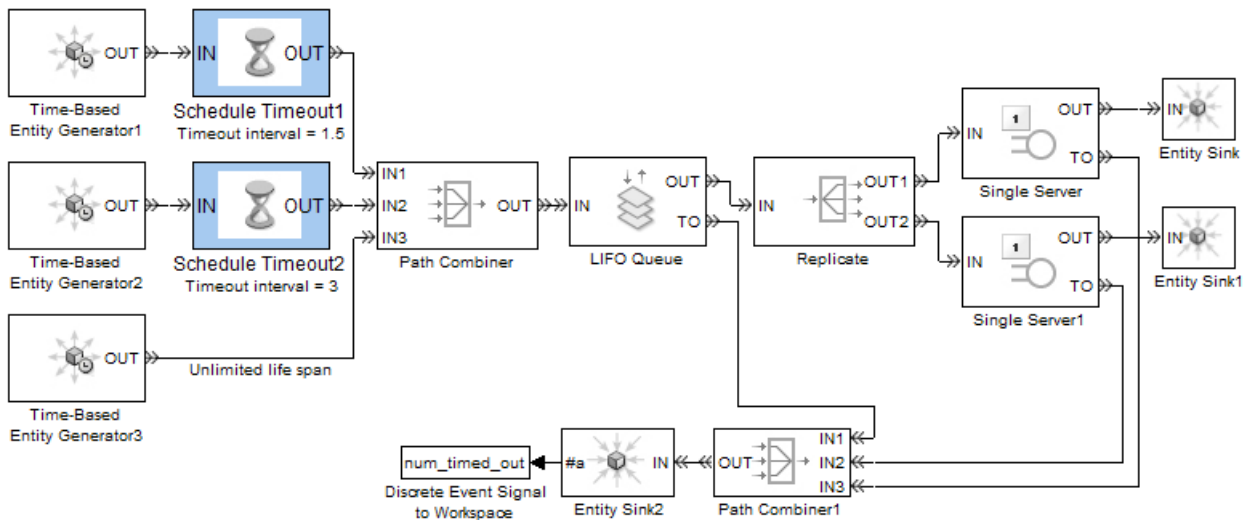
Other examples of timeouts on linear entity paths include these:

- “Basic Example Using Timeouts” on page 8-3
- “Example: Limiting the Time Until Service Completion” on page 8-14

Branched Path for Timeouts

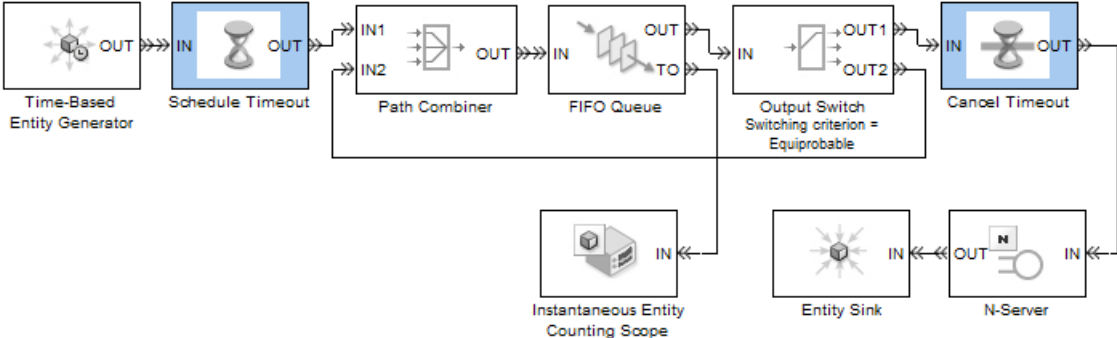
In the example below, entities from two sources have limited lifespans. Entities from a third source do not have limited lifespans.

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

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.

These topics describe and illustrate some common ways to handle timed-out entities:

- “Techniques for Handling Timed-Out Entities” on page 8-10
- “Example: Dropped and Timed-Out Packets” on page 8-11
- “Example: Rerouting Timed-Out Entities to Expedite Handling” on page 8-12

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

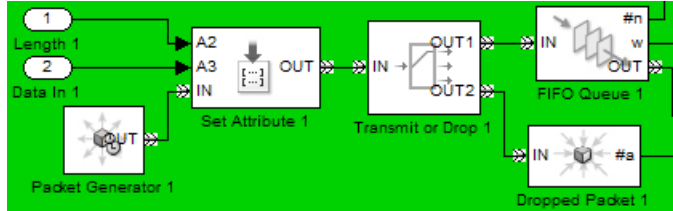
To combine paths of timed-out entities from multiple blocks, use a Path Combiner block.

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.

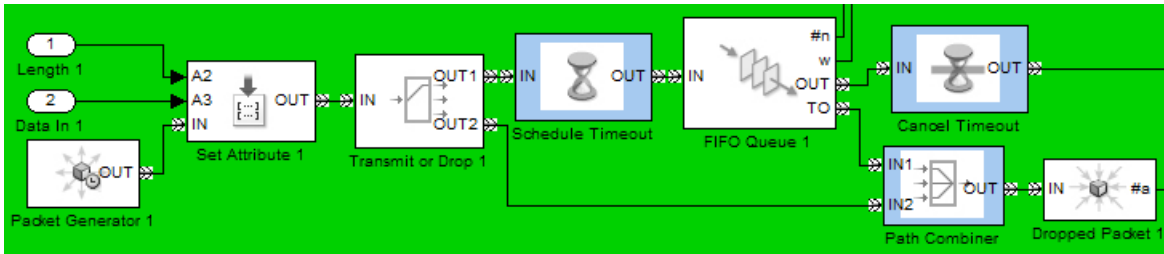
Example: Dropped and Timed-Out Packets

This example modifies a portion of the “Shared Access Communications Media” demonstration model, 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.

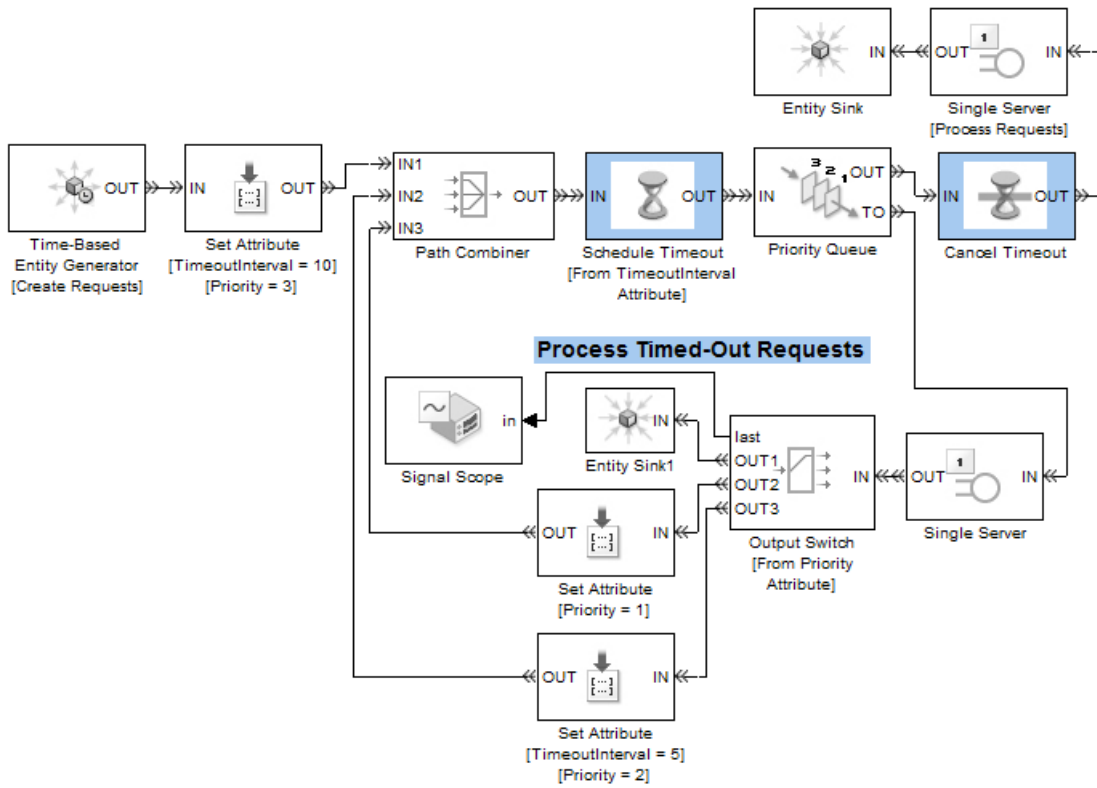


Transmitter Counting Dropped and Timed-Out Packets

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.

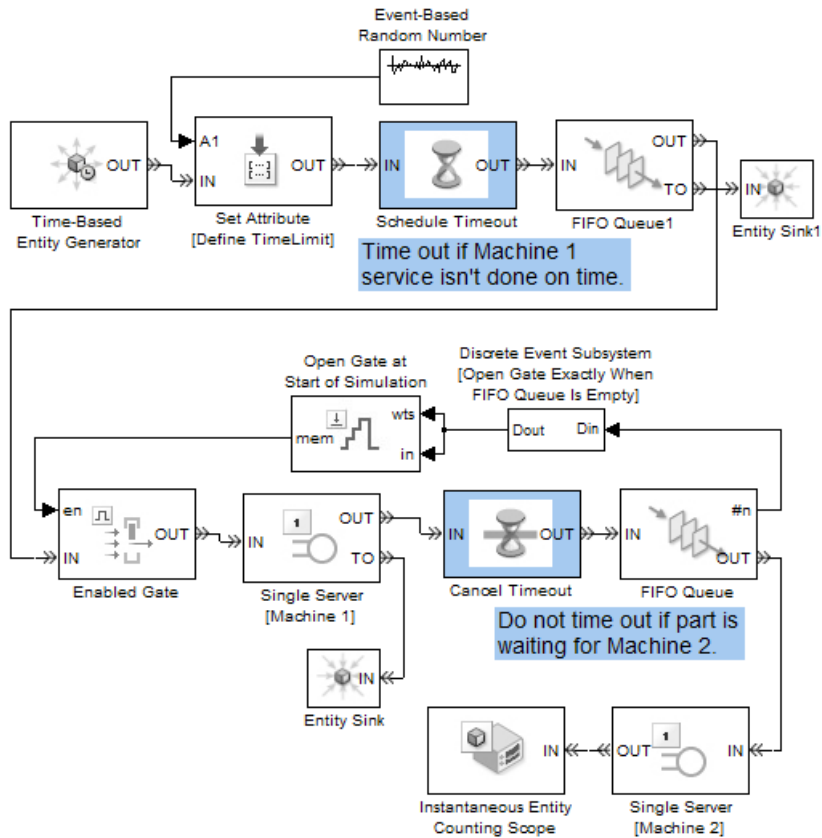


Example: Limiting the Time Until Service Completion

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.



The block labeled Open Gate at Start of Simulation is a Signal Latch block that defines an initial condition of 1 using the technique described in “Specifying Initial Conditions for Event-Based Signals” on page 3-27. Without this initial condition, the gate would never open for the first part.

Controlling Timing with Subsystems

Timing Issues in SimEvents Models
(p. 9-2)

Understanding timing in situations where time-based and event-based behavior interact

Role of Discrete Event Subsystems in SimEvents Models (p. 9-7)

Overview of discrete event subsystems as a way to ensure appropriate simulation timing

Blocks Inside Discrete Event Subsystems (p. 9-10)

Blocks that are suitable for use in a discrete event subsystem

Working with Discrete Event Subsystem Blocks (p. 9-11)

Setting up and configuring Discrete Event Subsystem blocks

Examples Using Discrete Event Subsystem Blocks (p. 9-17)

Examples of situations in which discrete event subsystems are useful

Creating Entity-Departure Subsystems (p. 9-27)

How to call a subsystem when an entity departs from a block

Examples Using Entity-Departure Subsystems (p. 9-30)

Examples of situations in which entity-departure subsystems are useful

Using Function-Call Subsystems (p. 9-33)

Using the Function-Call Subsystem block in SimEvents models

Timing Issues in SimEvents Models

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.

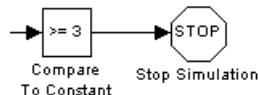
When you combine time-based and event-based blocks, consider whether you want the operation of a time-based block to depend *only* on time-oriented considerations or whether you want the time-based block to respond to events. The following example scenarios illustrate the difference:

- “Timing for the End of the Simulation” on page 9-2
- “Timing for a Statistical Computation” on page 9-3
- “Timing for Choosing a Port Using a Sequence” on page 9-4

The section “Role of Discrete Event Subsystems in SimEvents Models” on page 9-7 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 would *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 would cause the simulation to stop upon the next time step for the time-based Compare To Constant and Stop Simulation blocks, which could 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 Simulink defaults to time-based behavior for most blocks.

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 9-21.

Timing for a Statistical Computation

Suppose that you want to compute the total length of time for which the queue length equals or exceeds a threshold value during the simulation. A symbolic expression for this computation is

$$f(T) = \int_0^T I_{\#n \geq 3}(t) dt$$

where T is the total length of the simulation, $\#n$ is the instantaneous queue length, the threshold is three entities, and I is an indicator function defined by

$$I_{\#n \geq 3}(t) = \begin{cases} 1 & \text{if } \#n(t) \geq 3 \\ 0 & \text{otherwise} \end{cases}$$

You can use the optional **#n** signal output port from the FIFO Queue block to produce the queue length signal, and the Compare To Constant block to implement the indicator function. You can connect the resulting signal to an integrator block.

Time-Based Integration

The Integrator and Discrete-Time Integrator blocks in the Simulink libraries are inherently time-based. In this example, you are integrating a signal that experiences asynchronous discontinuities. A discrete-time integrator with an explicit sample time yields incorrect results because a discontinuity might occur between successive sample time values. A continuous-time integrator can make the simulation clock adjust to detected discontinuities in the signal, yet it is still possible for asynchronous discontinuities to go undetected by the time-based simulation clock.

Some applications require time-based integration, but this application requires that the integration respond to events that change the value of the **#n** signal.

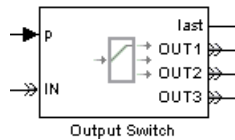
Achieving Correct Event-Based Behavior

To make the Compare To Constant and integrator blocks respond to events, not the time-based simulation clock, you can 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. This solution causes the simulation to compute the indicator function correctly, which results in a correct value for the statistic.

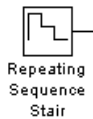
For details on this solution, see “Example: Using Event-Based Timing for a Statistical Computation” on page 9-20.

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.

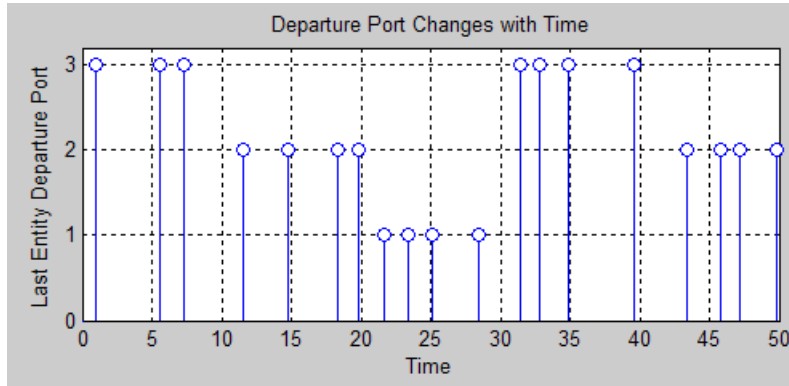
- It chooses a new value from the repeating sequence whenever an entity arrives at the Output Switch block, which might be appropriate if the switch is intended to distribute a load fairly among three operators who work in parallel.

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 9-27.

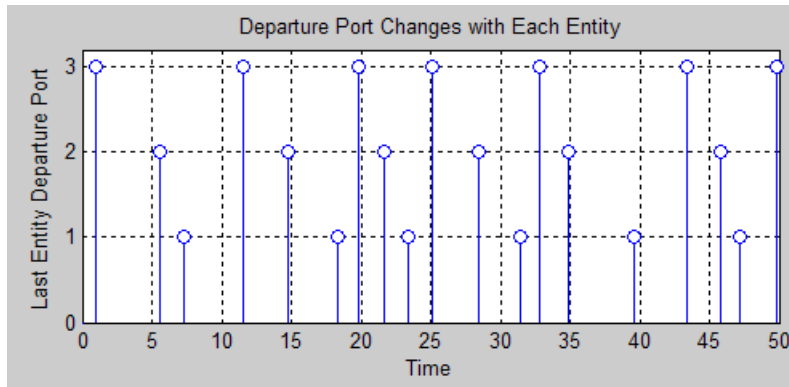
For details on this approach, see “Example: Using Entity-Based Timing for Choosing a Port” on page 9-30.

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. Sample plots are below.



Departure Port Changes with Time



Departure Port Changes with Each Entity

Role of Discrete Event Subsystems in SimEvents Models

- “Purpose of Discrete Event Subsystems” on page 9-8
- “Processing Sequence for Events in Discrete Event Subsystems” on page 9-8

Given the questions raised in “Timing Issues in SimEvents Models” on page 9-2 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, then use an ordinary Subsystem block from the Simulink Ports & Subsystems library. Alternatively, select one or more blocks and use the **Edit > Create Subsystem** menu option. In either case, the use of a subsystem 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.
- 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 Simulink 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

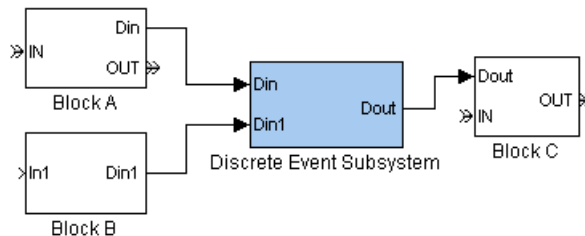
The purpose of a discrete event subsystem is to call the blocks in the subsystem at the exact time of each qualifying event and not at times suggested by the time-based simulation clock. This is an important change in the semantics of the model, not merely an optimization.

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 schematic below involving 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, you should 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 9-24 for an example in which the last-updated signal executes the subsystem. See “Update Sequence for Output Signals” on page 3-18 to learn more about when blocks update their output signals.

- Before the entity arrival at Block C, which uses an output signal from the subsystem. “Example: Normalizing a Statistic to Use for Routing” on page 9-18 shows how an extra server block whose service time is zero can help produce the correct processing sequence.

For details on processing sequences, see “Interleaving of Block Operations” on page 14-8 and “Processing Sequence for Simultaneous Events” on page 2-11.

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.
- Blocks that always inherit a sample time from the driving block, such as the Bias block or the Unary Minus 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
- 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.

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

Building on the conceptual information in “Role of Discrete Event Subsystems in SimEvents Models” on page 9-7, this section provides some practical information to help you incorporate Discrete Event Subsystem blocks into your SimEvents models. The topics are as follows:

- “Setting Up Signal-Based Discrete Event Subsystems” on page 9-11
- “Signal-Based Events That Control Discrete Event Subsystems” on page 9-14

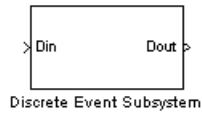
For discrete event subsystems that respond to entity departures rather than signal-based events, see “Creating Entity-Departure Subsystems” on page 9-27.

Setting Up Signal-Based Discrete Event Subsystems

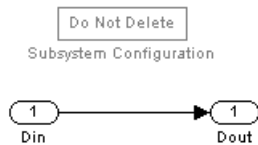
To create discrete event subsystems that respond to signal-based events, follow the procedure below using blocks in the SimEvents Ports and Subsystems library.

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, then you can copy them into the subsystem window of a Discrete Event Subsystem block while following the procedure below.

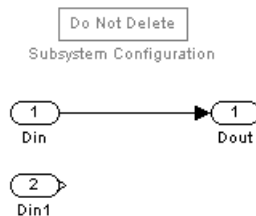
- 1 Drag the Discrete Event Subsystem block from the SimEvents Ports and Subsystems library into your model. Initially, it shows one signal input port **Din** and one signal output port **Dout**. Note that these are signal ports, not entity ports, because the subsystem is designed to process signals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



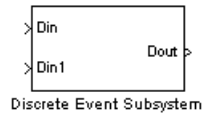
- 2** In the model window, double-click the Discrete Event Subsystem block to open the subsystem it represents. Initially, the subsystem contains an inport block connected to an outport block. Note that these are Discrete Event Inport and Discrete Event Outport blocks, which are not the same as the Inport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



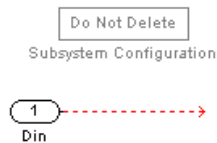
- 3** A discrete event subsystem must have at least one input that determines when the subsystem executes. To change the number of inputs or outputs to the subsystem, change the number of inport and outport blocks in the subsystem window:
- If your subsystem requires an additional input or output, then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



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

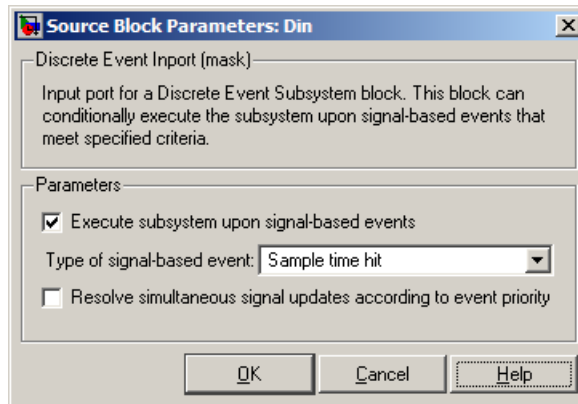


As a result, the subsystem block at the upper level of your model omits the output port. The figure below shows the absence of output ports on the subsystem block.



- 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 9-10 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.
 - 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 are called in response to signal-based events. Using the dialog box of the Discrete Event Inport blocks inside the subsystem, you configure the subsystem so that it is executed in response to

- An updated value in an input signal, even if the updated value is the same as the previous value
- A change in the value of an input signal
- A rising edge or falling edge in an input signal known as a trigger signal

Note To call a subsystem upon an entity departure or upon a function call, see “Creating Entity-Departure Subsystems” on page 9-27 or “Using Function-Call Subsystems” on page 9-33, respectively.

Multiple-Input Subsystems

In a discrete event subsystem containing multiple Discrete Event Inport blocks, the subsystem is executed when at least one of the inport blocks detects a qualifying event. If N qualifying events occur at the same simulation time (whether at the same or different Discrete Event Inport blocks), then the subsystem executes N times and updates its output signals N times.

If you want one of the inport blocks to provide an input signal without affecting the times at which the subsystem is executed, then clear the **Execute subsystem upon signal-based event** check box on that inport block. However, always select **Execute subsystem upon signal-based event** for at least one inport block of the subsystem or else the subsystem will never be executed.

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:

- 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, then 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 “Multiple Simultaneous Updates of an Output Signal” on page 3-21 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.

For more information about signal-based events, see “Types of Supported Events” on page 2-2.

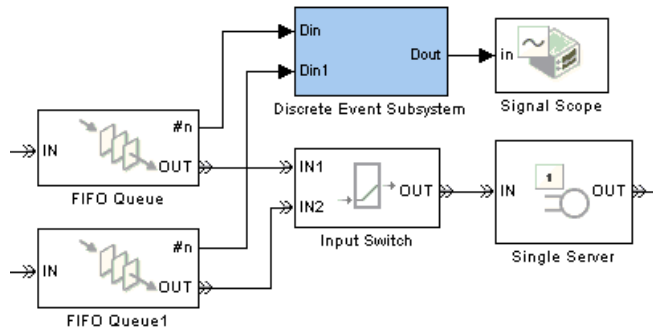
Examples Using Discrete Event Subsystem Blocks

The topics listed below illustrate the use of the Discrete Event Subsystem block.

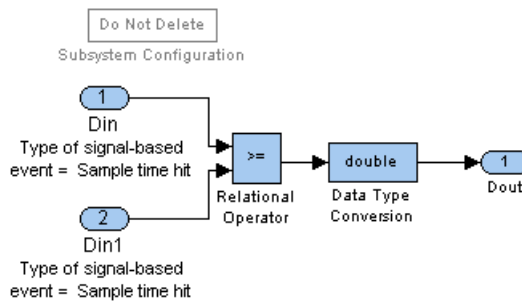
Example: Comparing the Lengths of Two Queues (p. 9-17)	Manipulating two event-based signals having different timing
Example: Normalizing a Statistic to Use for Routing (p. 9-18)	Performing event-oriented computation for use in a subsequent block
Example: Using Event-Based Timing for a Statistical Computation (p. 9-20)	Integrating a signal having asynchronous discontinuities
Example: Ending the Simulation Upon an Event (p. 9-21)	Responding immediately to asynchronous discontinuities
Example: Sending Unrepeated Data to the MATLAB Workspace (p. 9-22)	Logging data only when a signal changes
Example: Focusing on Events, Not Values (p. 9-23)	Counting changes in a signal's value
Example: Detecting Changes from Empty to Nonempty (p. 9-24)	Manipulating one signal when another signal crosses zero
Example: Logging Data About the First Entity on a Path (p. 9-25)	Focusing on the first entity that uses a path

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 model below 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, then 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 model shown below, 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. The subsystem's configuration and the presence of the Delay of Length Zero block

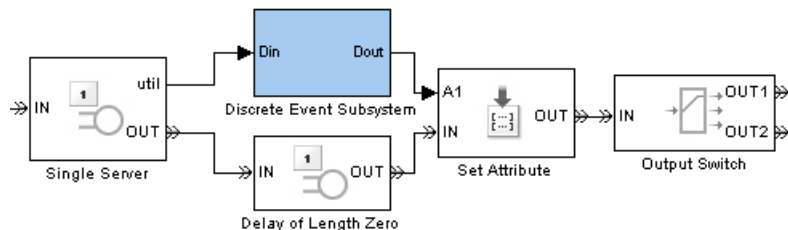
ensure that the attribute assignment and the routing decision always use the most up-to-date value of the **util** signal.

At a time instant at which an entity departs from the Single Server block, the sequence of operations is as follows:

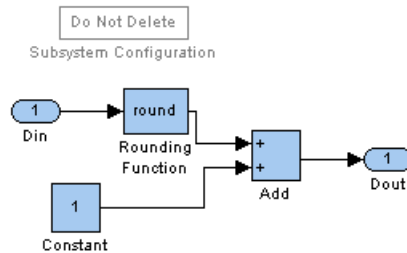
- 1 The entity advances from the Single Server block to the block labeled Delay of Length Zero, which is merely a single server whose service time is zero.
- 2 The Single Server block updates its **util** output signal.

The fact that this occurs after the entity has already departed from the Single Server block and arrived at a storage block is the reason for the Delay of Length Zero block. See “Interleaving of Block Operations” on page 14-8 for details.

- 3 The subsystem reacts to the updated value of **util** by computing an updated value at the **A1** input port of the Set Attribute block.
- 4 The entity advances from the Delay of Length Zero block to the Set Attribute block, which uses an up-to-date signal to determine the attribute value.
- 5 The entity advances from the Set Attribute block to the Output Switch block, which uses the entity’s attribute value to determine the routing behavior.



Top-Level Model

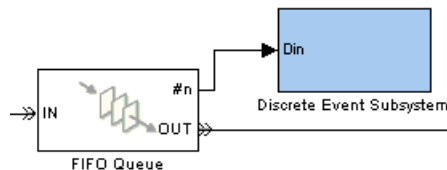


Subsystem Contents

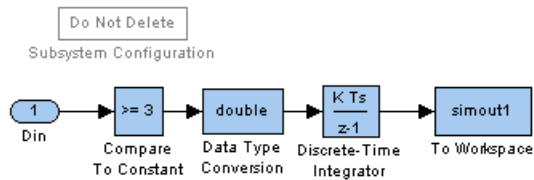
Example: Using Event-Based Timing for a Statistical Computation

This example performs the computation described in “Timing for a Statistical Computation” on page 9-3, to find the total length of time during the simulation that a queue’s length equals or exceeds a threshold value.

In the model shown below, the threshold is three entities. The Compare To Constant block produces an indicator function for that threshold. The Discrete Time Integrator block, configured to have an inherited sample time rather than an explicit discrete sample time, computes the area under the curve of the indicator function, that is, the total amount of time that the input to the subsystem exceeds the threshold. 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 queue block updates the value of the queue length signal. The subsystem’s configuration ensures that the comparison block does not miss any asynchronous discontinuities in the queue length signal and that the integrator processes the correct indicator function.



Top-Level Model

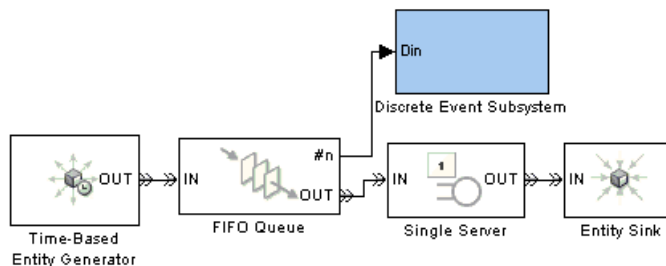


Subsystem Contents

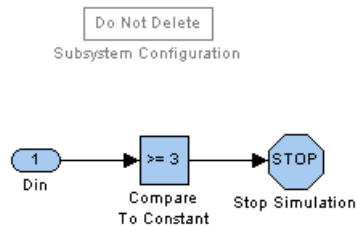
Note In this example, it is important to use a discrete-time integrator with an inherited sample time instead of a continuous-time integrator. See “Blocks Inside Discrete Event Subsystems” on page 9-10 for more information.

Example: Ending the Simulation Upon an Event

This example ends the simulation as described in “Timing for the End of the Simulation” on page 9-2, precisely when the number of entities in a queue first equals or exceeds a threshold. In the model shown below, 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 Simulink calls the subsystem at exactly those times when the FIFO Queue block updates the value of the queue length signal.



Top-Level Model

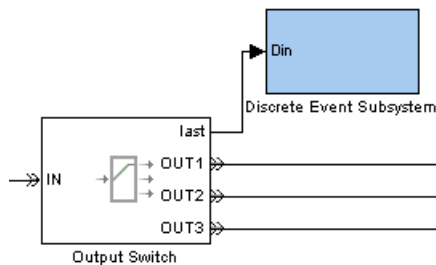


Subsystem Contents

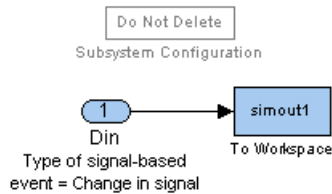
Example: Sending Unrepeated Data to the MATLAB Workspace

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 model shown below, 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



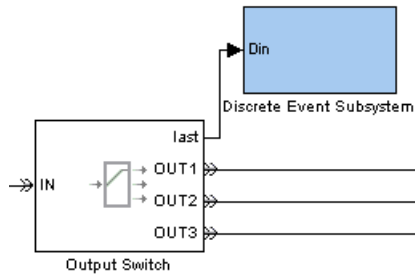
Subsystem Contents

Example: Focusing on Events, Not Values

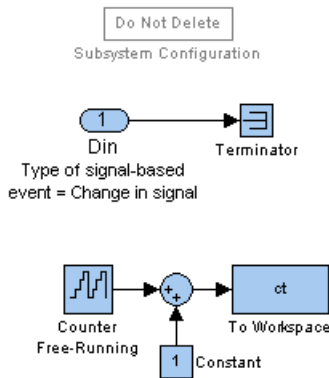
The example below counts the number of times a signal changes its value, ignoring times at which the signal might be updated with the same value. The discrete event subsystem contains a Counter Free-Running block with an inherited sample time. Because the Counter Free-Running counts starting from 0, 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 results.



Top-Level Model



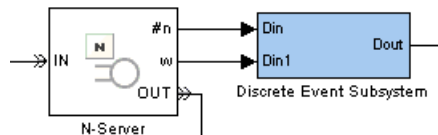
Subsystem Contents

Example: Detecting Changes from Empty to Nonempty

The example below 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 zero. 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 zero to one or from one to zero. 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, then 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

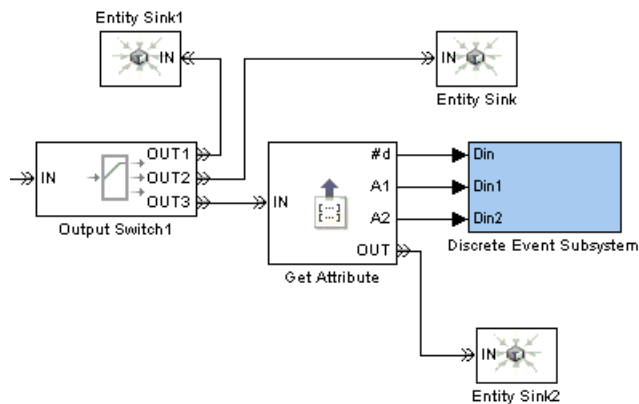
Suppose your model includes a particular entity path that entities rarely use, and you want to record certain attributes of the *first* entity that takes that path during the simulation. You can send the attribute values to the MATLAB workspace by using a To Workspace block inside a discrete event subsystem.

In the example below, 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 MATLAB 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 MATLAB 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.

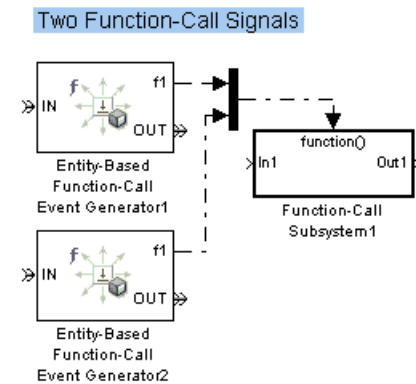
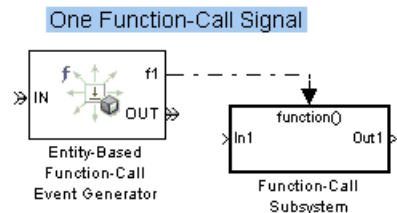
Note Because the Get Attribute block updates the **A1** and **A2** signals before updating the **#d** signal, all signals are up to date when the trigger edge occurs.



Creating Entity-Departure Subsystems

- “Accessing Blocks for Entity-Departure Subsystems” on page 9-28
- “Setting Up Entity-Departure Subsystems” on page 9-29

You can create a subsystem that Simulink calls only when an entity departs from a particular block in your model. The figure below 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 below.

Block	Library Location	Purpose
Entity-Based Function-Call Event Generator	Event Generators library in SimEvents	Issues a function call corresponding to each entity departure
Entity Departure Event to Function-Call Event	Event Translation library in SimEvents	
Mux	Signal Routing library in Simulink	Combines multiple function-call signals into a single function-call signal, if needed
Function-Call Subsystem	Ports & Subsystems library in Simulink	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 9-27.
Inport	Ports & Subsystems library in Simulink	Links a subsystem to its parent system
Outport		

Setting Up Entity-Departure Subsystems

To create subsystems that respond to entity departures, follow the procedure below.

- 1 Insert and configure the Function-Call Subsystem block as described in “Setting Up Function-Call Subsystems in SimEvents Models” on page 9-34.
- 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 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, then combine multiple function-call signals using a Mux block.

Examples Using Entity-Departure Subsystems

The topics listed below illustrate the use of entity-departure subsystems.

Example: Using Entity-Based Timing for Choosing a Port (p. 9-30) Calling a signal source when an entity departs from a block

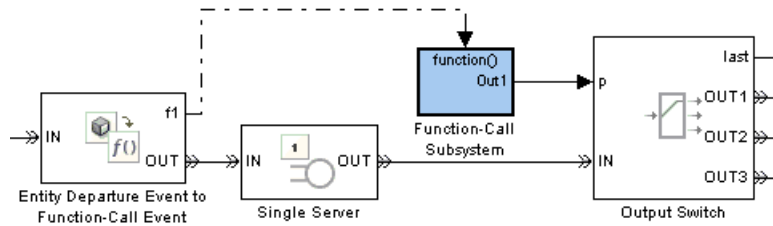
Example: Performing a Computation on Selected Entity Paths (p. 9-32) Detecting departures from multiple blocks

Example: Using Entity-Based Timing for Choosing a Port

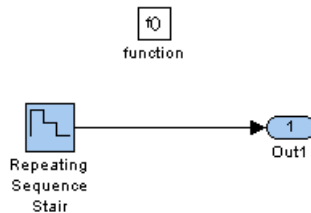
This example performs the entity-based routing described in “Timing for Choosing a Port Using a Sequence” on page 9-4. The example routes entities by establishing a sequence of paths and then choosing a number from that sequence for each entity that arrives at the routing block. This is the situation shown in the figure *Departure Port Changes with Each Entity* on page 9-6.

In the model shown below, 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 9-6, rather than on entity departures.



Top-Level Model



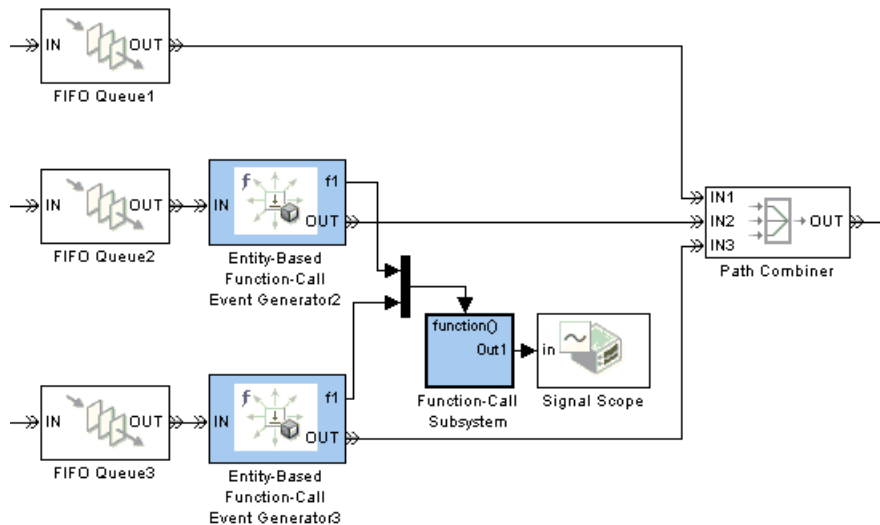
Subsystem Contents

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 rather is stored in the server. Possible alternative approaches are to

- Place the Entity Departure Event to Function-Call Event block, followed by an extra server, between the Single Server block and the switch. This approach is similar to the use of the Delay of Length Zero block in “Example: Normalizing a Statistic to Use for Routing” on page 9-18.
- Configure the Entity Departure Event to Function-Call Event block to issue function calls before entity departures and place the block between the server and the switch. Issuing function calls before departures does not lead to causality problems in this particular example.

Example: Performing a Computation on Selected Entity Paths

The model below 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, then the subsystem is called twice at that time.



Using Function-Call Subsystems

- “Use Cases for Function-Call Subsystems” on page 9-33
- “Setting Up Function-Call Subsystems in SimEvents Models” on page 9-34

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 9-11 and “Creating Entity-Departure Subsystems” on page 9-27, 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 2-50.

- 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).

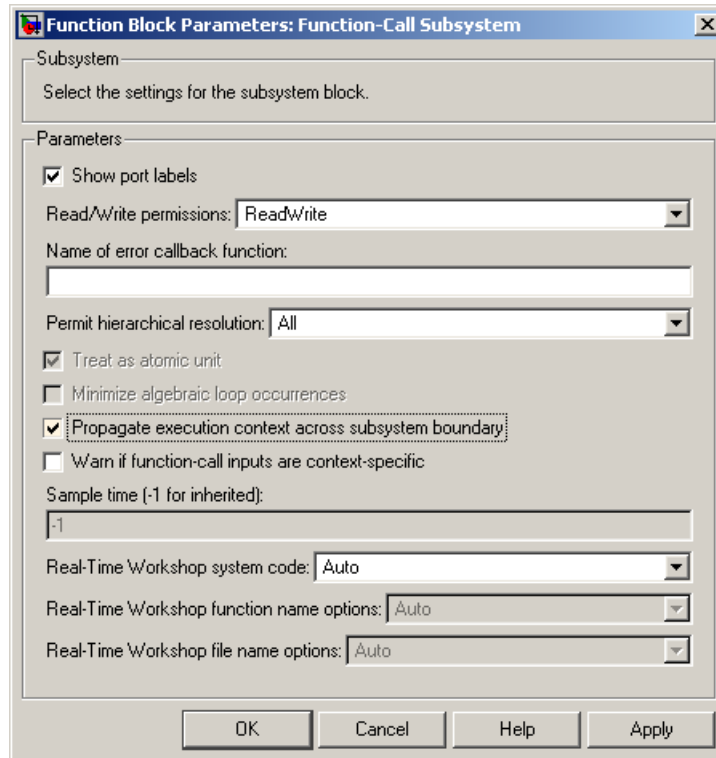
For an example, see “Example: Detecting Changes in the Last-Updated Signal” on page 3-18.

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 procedure below.

Note The selection of 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()**. Note that 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 dialog box that opens, 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. Note that these 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.
- 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 9-10 for details.

Plotting Data

Choosing and Configuring Plotting Blocks (p. 10-2)

Features of Plot Window (p. 10-8)

Using Plots for Troubleshooting (p. 10-9)

Example: Plotting Entity Departures to Verify Timing (p. 10-10)

Example: Plotting Event Counts to Check for Simultaneity (p. 10-14)

Comparison with Time-Based Plotting Tools (p. 10-16)

Available types of plots and how to access them

Customizing plots

Visualizing a simulation to understand and debug it

Graphical way to detect latency

Graphical way to check whether events are simultaneous

Notes about plots designed for time-based simulations

Choosing and Configuring Plotting Blocks

SimEvents provides scope blocks that help you visualize data from your simulation. When you insert a scope block into your model, it creates a plot as the simulation runs. This section describes capabilities of the scope blocks in these topics:

- “Sources of Data for Plotting” on page 10-2
- “Inserting and Connecting Scope Blocks” on page 10-3
- “Connections Among Points in Plots” on page 10-4
- “Varying Axis Limits Automatically” on page 10-5
- “Caching Data in Scopes” on page 10-6
- “Examples Using Scope Blocks” on page 10-6

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	

Data	Block	Parameter
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		
Number of entity arrivals per time instant	Instantaneous Entity Counting Scope	
Number of events per time instant	Instantaneous Event Counting Scope	

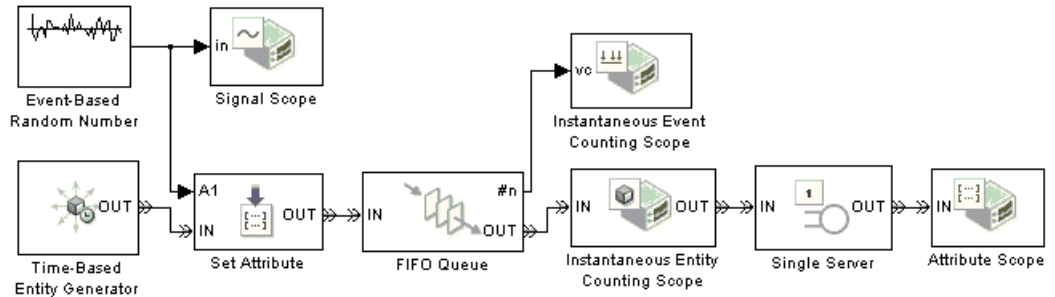
The Signal Scope and X-Y Signal Scope blocks are particularly appropriate for data arising from discrete-event simulations because the plots can include zero-duration values. That is, the plots can include multiple values of the signal at a given time. By contrast, the Scope block in the Simulink Sources 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 below indicates the input ports on each scope block.

Block	Input Port(s)	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

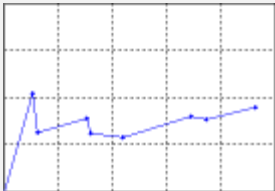
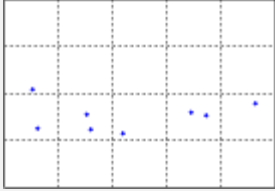
The figure below shows some typical arrangements of scope blocks in a model. Notice that the blocks that have entity input ports can have optional entity output ports, and that signal lines can branch whereas entity connection lines cannot.



Connections Among Points in Plots

You can configure certain scope blocks to determine if and how it connects the points that it plots. The table below 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's 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's dialog box	

Connection Characteristics	Setting	Sample Plot
Single line segment from point to point. Also known as a first-order hold.	Plot type = Continuous in the block's 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 $T=0$, then the plot window does not show a point at $T=0$ and 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 horizontal and vertical axes of the plot. Also, the **If X value is beyond limit** and **If Y value is beyond limit** parameters let you choose how the block responds if a point does not fit within the current axis limits. Choices are in the table below.

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 closed during part or all of 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, you might see uncached data points disappear when the simulation ends or if you interact with the plot while the simulation is paused.

Examples Using Scope Blocks

The following examples use scope blocks to create different kinds of plots:

Example	Description
“Plotting the Pending-Entity Signal” and “Observations from Plots” in the getting started documentation	Stairstep and continuous plots of statistical signals
“Example: Round-Robin Approach to Choosing Inputs” in the getting started documentation	Stem plot of data from an attribute
“Example: Using Servers in Shifts” on page 6-11	Unconnected plot of a signal using dots
“Example: Setting Attributes” on page 1-14	Stairstep plots of data from attributes using Attribute Scope blocks as sinks

Example	Description
“Example: Synchronizing Service Start Times with the Clock” on page 7-6	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

Features of Plot Window

After a scope block opens its plot window, you can modify several aspects of the plot by using the plot window's menu and toolbar:

- **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 are saved when you save the model.
- **Axes > Save axes limits** updates the **Initial X axis lower limit**, **Initial X axis upper limit**, **Initial Y axis lower limit**, and **Initial Y axis upper limit** parameters on the **Axes** tab of the block's dialog box to reflect the current limits on the horizontal and vertical axes.
- **Axes > Save position** updates the **Position** parameter on the **Figure** tab of the block's dialog box to reflect the window's current position and size.

Note When a menu option duplicates the behavior of a parameter in the block's dialog box, 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 this is the **Show grid** menu option and dialog box parameter.

The Save Figure toolbar button lets you save the current state of the plot to a FIG-file. You can reload the file in a different MATLAB session. Reloading the file creates a plot that is *not* associated with the original scope block and that does not offer the same menu and toolbar options as in the original plot window.

Using Plots for Troubleshooting

Typical ways to use plotting blocks in the SimEvents Sinks library to troubleshoot problems are described in the table below.

Technique	Example
Check when an entity departs from the block. To do this, plot a block's #d output signal.	“Example: Plotting Entity Departures to Verify Timing” on page 10-10
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 6-11 and “Timing for Choosing a Port Using a Sequence” on page 9-4
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: Race Conditions at a Switch” on page 2-25
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 4-13
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 10-14

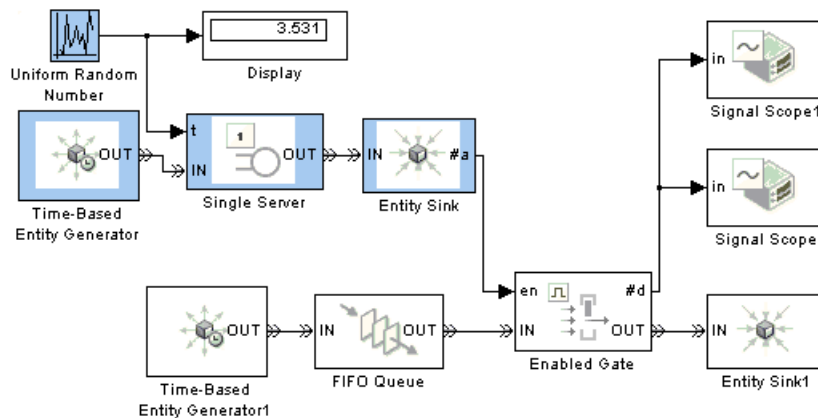
Example: Plotting Entity Departures to Verify Timing

- “Model Exhibiting Correct Timing” on page 10-10
- “Model Exhibiting Latency” on page 10-11

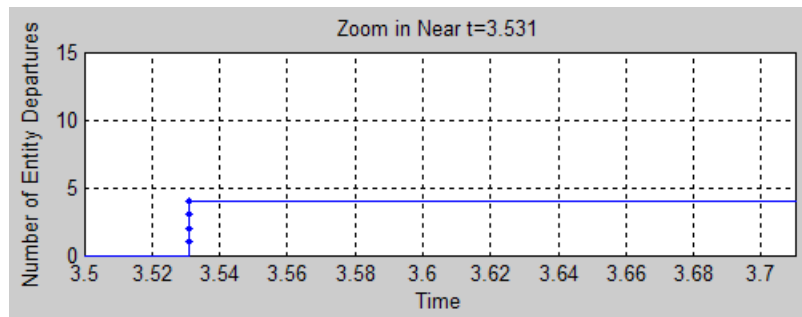
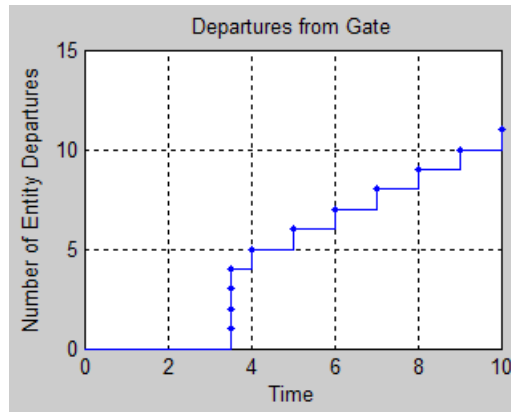
The example below compares two methods of opening a gate at a random time and leaving the gate open for the rest of the simulation. The Signal Scope block lets you see when the gate opens, to check whether the timing is what you expected. One method exhibits latency in the gate’s opening. For a nonvisual way to determine when entities depart from the gate, see “Viewing Entity Locations” on page 13-9.

Model Exhibiting Correct Timing

The model below 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 seconds in this case. At $T=3.531$, the entity arrives at the Entity Sink block. This time is exactly when the sink block’s **#a** signal changes from 0 to 1, which in turn causes the gate to open.

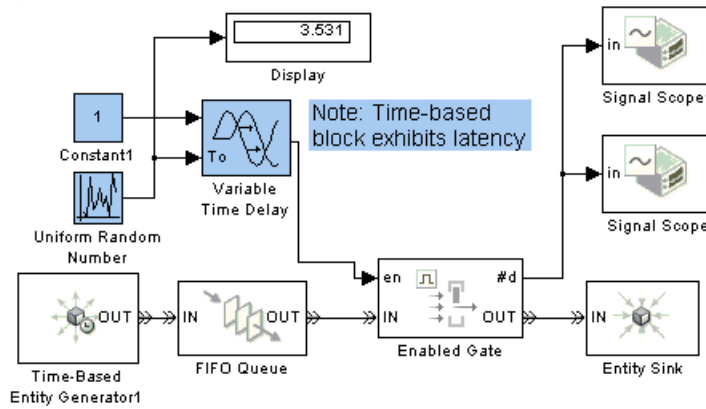


By using the zoom feature of the scope, you can compare the time at which entities depart from the Enabled Gate block with the random time shown on the Display block in the model.

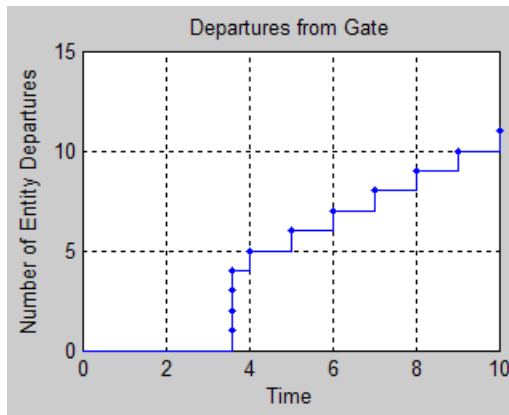


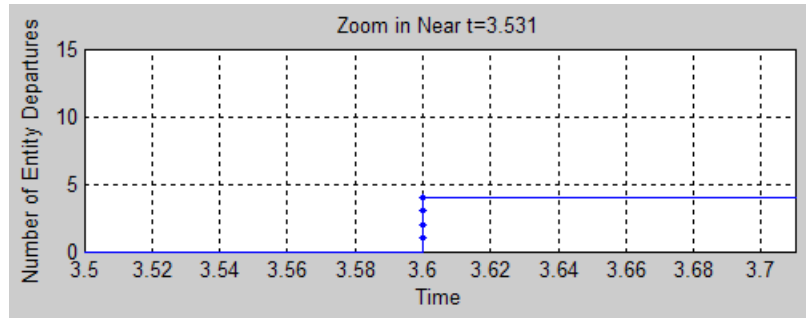
Model Exhibiting Latency

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.



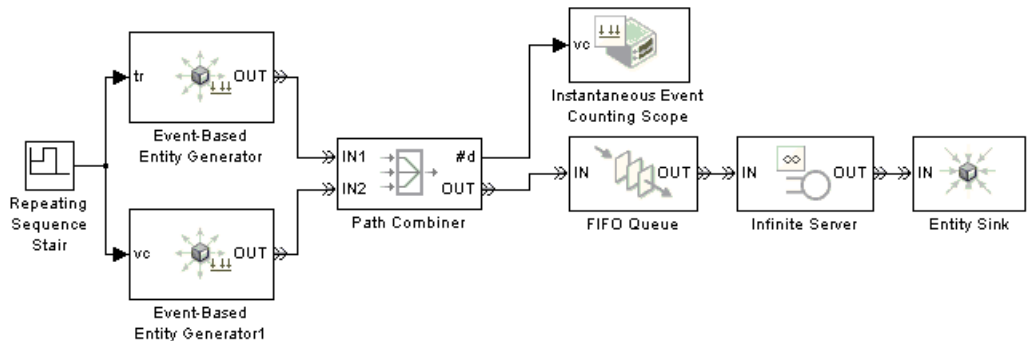


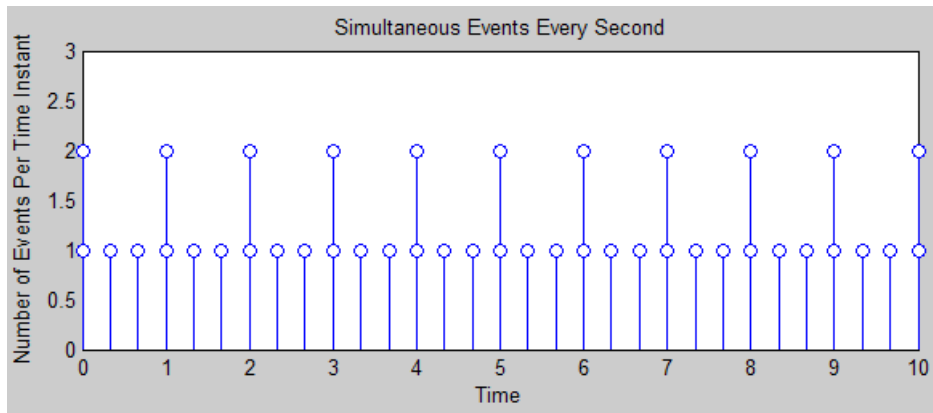
Example: Plotting Event Counts to Check for Simultaneity

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.



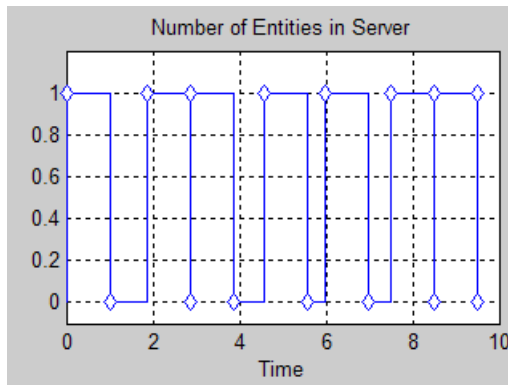


Although this example uses the Instantaneous Event Counting Scope to plot a #d signal, you can alternatively use the Instantaneous Entity Counting Scope to count entities departing from the Path Combiner block.

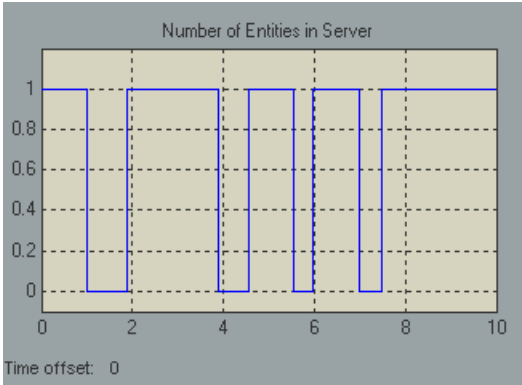
Comparison with Time-Based Plotting Tools

Simulink 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 (p. 11-2)	How well-designed simulation statistics can help you learn or make decisions about your system
Accessing Statistics from SimEvents Blocks (p. 11-4)	Computing statistics while the simulation runs
Deriving Custom Statistics (p. 11-7)	Creating your own statistics by manipulating outputs from SimEvents blocks
Using Timers (p. 11-19)	Recording how long entities take to reach a block
Running a Series of Simulations (p. 11-26)	Running multiple simulations and regulating the simulation length

Role of Statistics in Discrete-Event Simulation

Most SimEvents blocks are capable of 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:

- “Statistics for Data Analysis” on page 11-2
- “Statistics for Run-Time Control” on page 11-3

The rest of this chapter illustrates some modeling and analysis techniques that you can use with SimEvents. 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 replications of the simulation do you need? One simulation run, no matter how long, is still a single sample and probably inadequate for valid statistical analysis.

For details concerning statistical analysis and variance reduction techniques, see the works [7], [4], [1], and [2] listed in “Selected Bibliography” in the getting started documentation.

Statistics for Run-Time Control

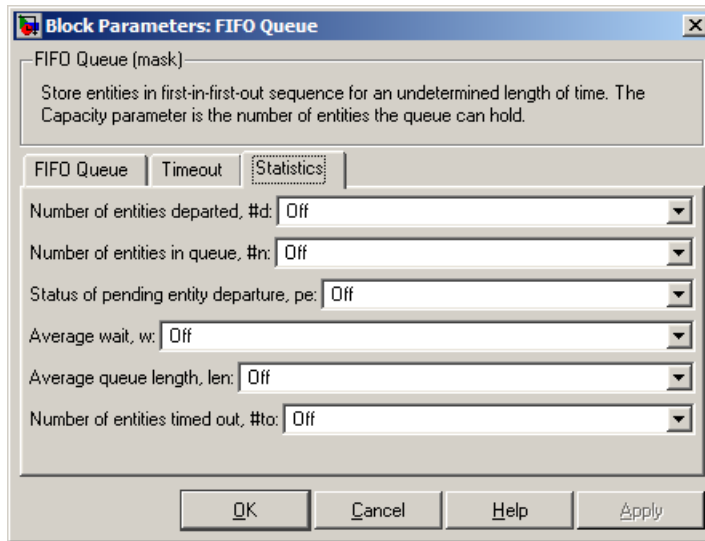
Some systems rely on statistics to influence the dynamics. For example, a queuing system with discouraged arrivals has a feedback loop that adjusts the arrival rate throughout the simulation based on statistics reported by the queue and server, as illustrated in the `sedemo_discourgearrival` demo model.

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 3, “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

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.

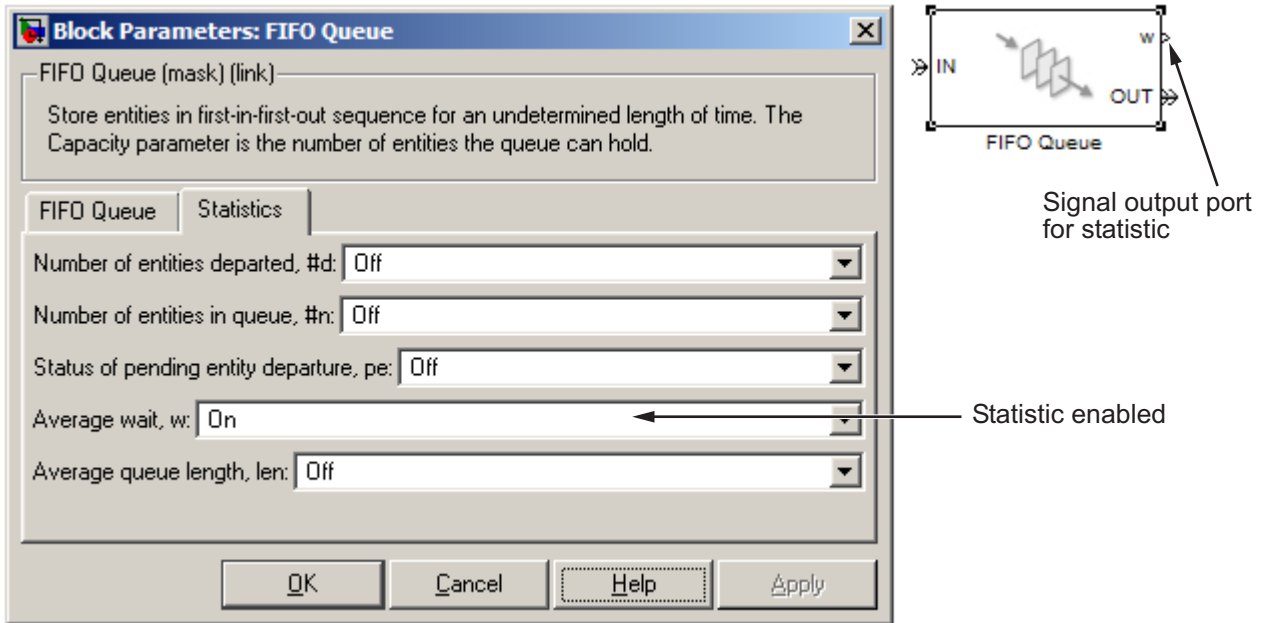
To use one or more statistics, see the options described in these sections:

- “Accessing Statistics Throughout the Simulation” on page 11-4
- “Accessing Statistics When Stopping or Pausing Simulation” on page 11-6

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 3-31.
- A custom subsystem or computational block, for further data processing. See “Deriving Custom Statistics” on page 11-7 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 3, “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. To do this, find the dialog box parameter that corresponds to the given statistic and 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

- “Graphical Block-Diagram Approach” on page 11-7
- “Coded Approach” on page 11-8
- “Post-Simulation Analysis” on page 11-8
- “Example: Fraction of Dropped Messages” on page 11-8
- “Example: Computing a Time Average of a Signal” on page 11-10
- “Example: Resetting an Average Periodically” on page 11-12

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 in 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 Simulink and the Statistics library in the Signal Processing Blockset can help you compute statistics using blocks. For examples using Simulink blocks, see

- “Example: Using Event-Based Timing for a Statistical Computation” on page 9-20, which computes the length of time during the simulation that a queue length is above a threshold
- “Example: Fraction of Dropped Messages” on page 11-8
- “Example: Detecting Changes in the Last-Updated Signal” on page 3-18, 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

Coded Approach

The blocks in the User-Defined Functions library in Simulink 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 11-10
- “Example: Resetting an Average Periodically” on page 11-12

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.

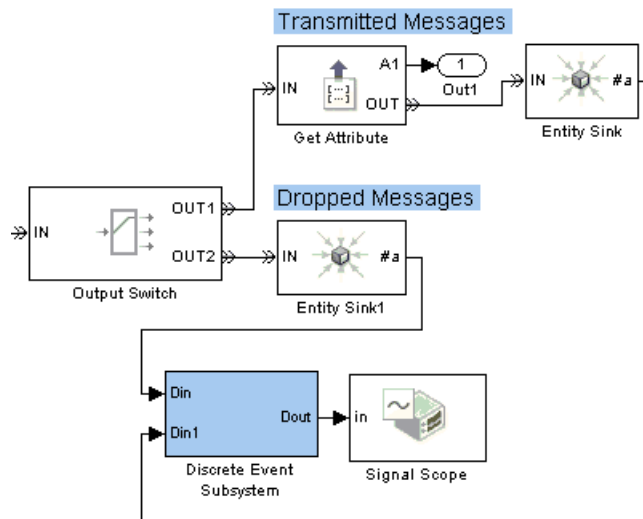
Post-Simulation Analysis

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

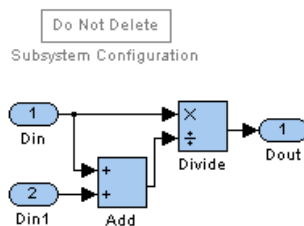
- “Example: Observing Service Completions” on page 2-38, which computes the times at which a signal value either increases or remains the same
- “Example: Running a Simulation and Varying a Parameter” on page 11-30

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.

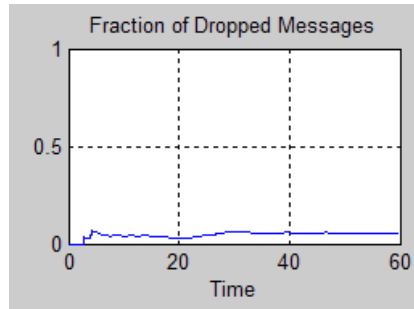


Upper-Level System



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.

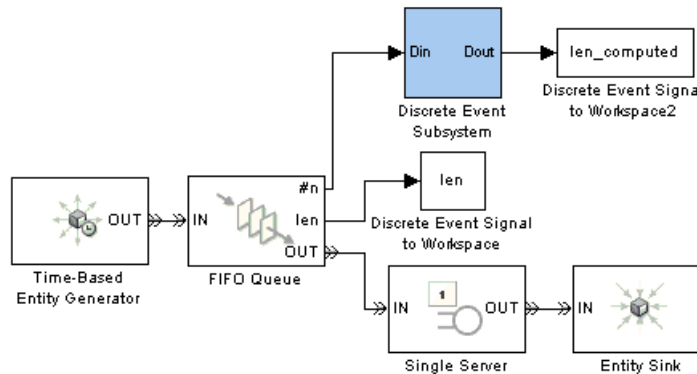


Example: Computing a Time Average of a Signal

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

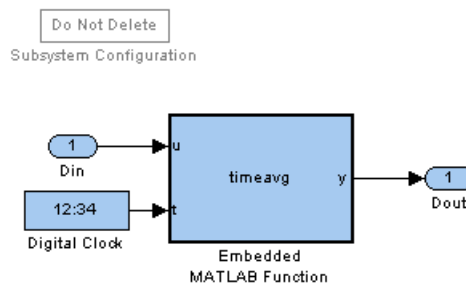
The discrete event subsystem uses **#n** to compute the time average. In this case, the time average should equal **len**. You can use a similar subsystem in your own models to compute the time averages of other signals.

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 =
```

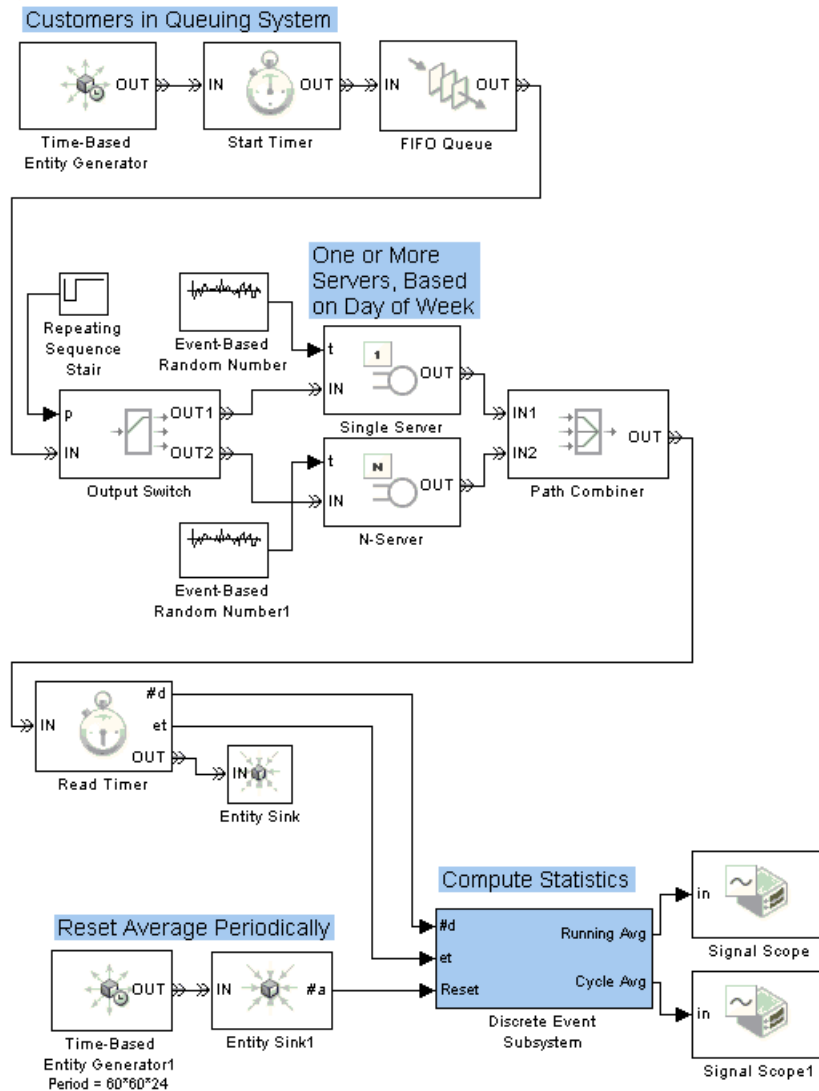
```
1
```

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.

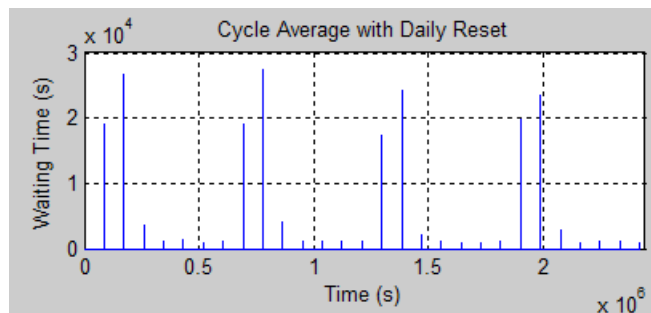
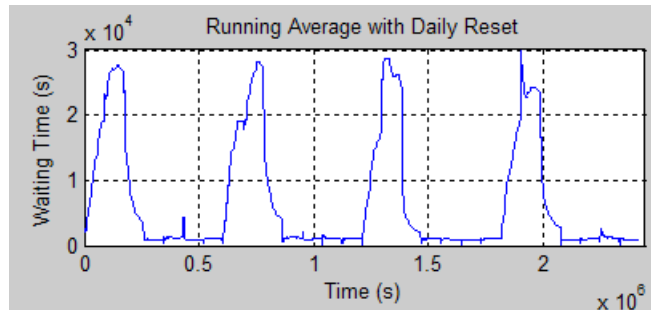
The model below uses a time-based Repeating Sequence Stair block to determine whether entities advance to a Single Server or N-Server block, thus creating variations in the number of operational servers. The Start Timer and Read Timer blocks compute each entity's waiting time in the queuing system. A 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 11-16.



Top-Level Model

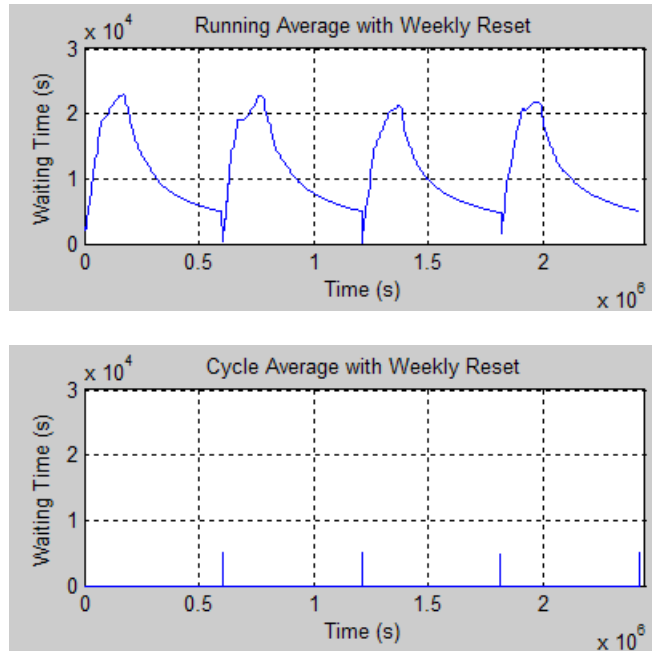
Performance of Daily Averages

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 $60 \cdot 60 \cdot 24 \cdot 7$, which is the number of seconds in a week.



Computation of the Cycle Average

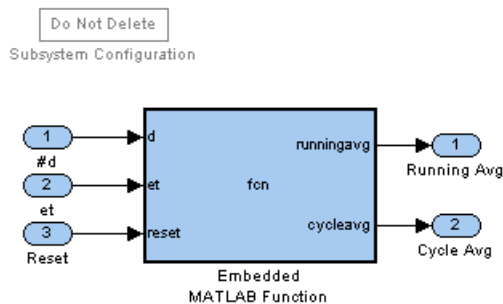
In the example, the discrete event subsystem performs computations each time a customer departs from the queuing system and at each boundary of a daily or weekly cycle. Within the subsystem, an Embedded MATLAB Function block counts the number of customers and the total waiting time among all customers 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

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.

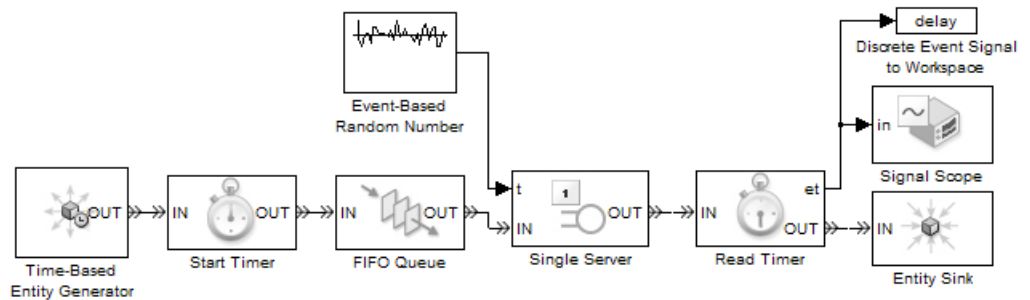
The following sections describe how to arrange the Start Timer and Read Timer blocks to accomplish several common timing goals:

- “Basic Example Using Timer Blocks” on page 11-19
- “Basic Procedure for Using Timer Blocks” on page 11-21
- “Timing Multiple Entity Paths with One Timer” on page 11-21
- “Restarting a Timer from Zero” on page 11-23
- “Timing Multiple Processes Independently” on page 11-24

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.

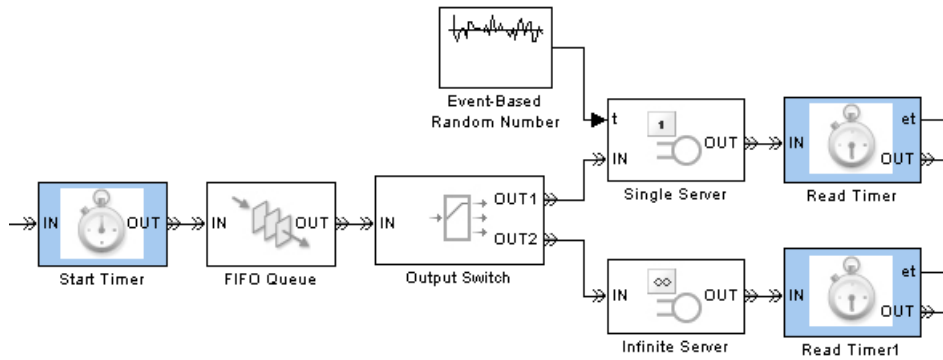
If you need multiple independent timers per entity (for example, to time an entity's progress through two possibly overlapping regions of the model), then follow the procedure above for each of the independent timers. For more information, see "Timing Multiple Processes Independently" on page 11-24.

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.

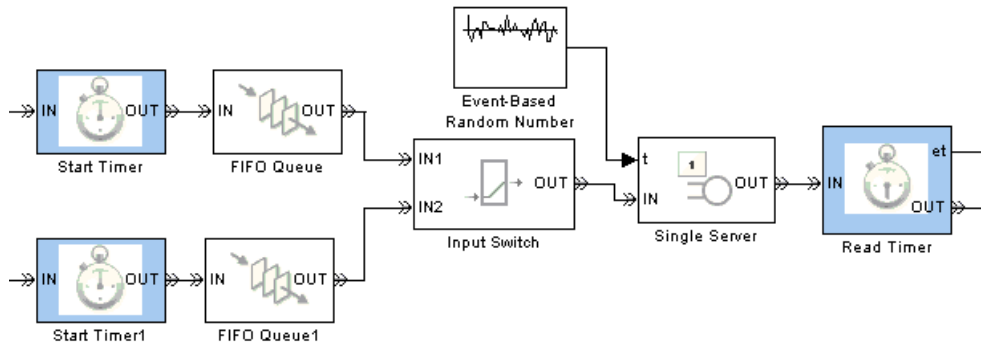
Output Switch Example

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

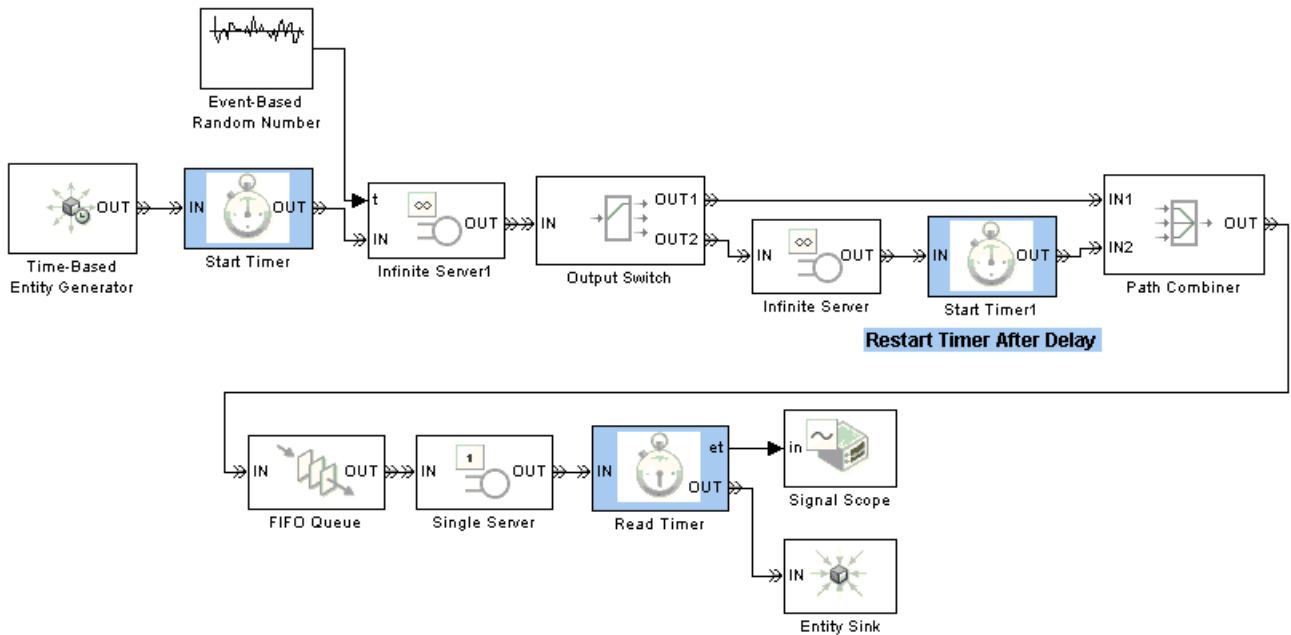
In the figure below, entities wait in two different queues before advancing to a single server. The timer blocks measure the time each entity spends in its respective queue-server pair. Two Start Timer blocks, configured with the same **Timer tag** parameter value, ensure that all entities possess a 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:

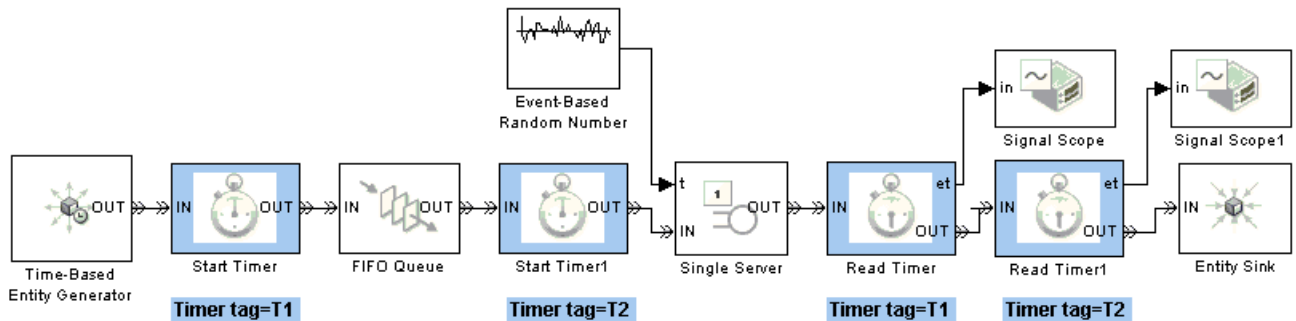
- The initial delay plus the time in the queue plus the time in the server, for entities that departed from the Output Switch block's **OUT1** port
- The time in the queue plus the time in the server, for entities that departed from the Output Switch block's **OUT2** port

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.

Running a Series of Simulations

This section describes some techniques that can help you gather statistical data from a series of simulations of your model. The topics are as follows:

- “Creating Independent Replications” on page 11-26
- “Running Simulations from MATLAB” on page 11-28
- “Regulating the Simulation Length” on page 11-33

Creating Independent Replications

When you run a simulation multiple times to gather statistics, you can create independent replications by using a different stream of random numbers in each replication. To vary the stream of random numbers, vary the *initial seed* on which the stream of random numbers is based. **Initial seed** is a parameter in the following blocks:

- Time-Based Entity Generator
- Event-Based Random Number
- Uniform Random Number
- Random Number
- Blocks in the Routing library

Also, if your simulation is configured to randomize the sequence of certain simultaneous events, then the Configuration Parameters dialog box has a parameter called **Seed for event randomization**, which indicates the initial seed for that stream of random numbers.

Choosing Values for Initial Seed

Here are some recommendations for choosing appropriate values for **Initial seed** parameters:

- Choose a large odd number, such as a five-digit odd number.
- To obtain the same stream of random numbers the next time you run the same simulation, set **Initial seed** to an unchanging value.

- To obtain a different stream of random numbers the next time you run the same simulation, either change the value of **Initial seed** or set it to a varying expression such as `ceil(cputime*99999)*2+1`. See the `cputime` function for more details.
- If **Initial seed** parameters appear in multiple places in your model, then choose different values or different expressions for each **Initial seed** parameter.

Setting Values for Initial Seed

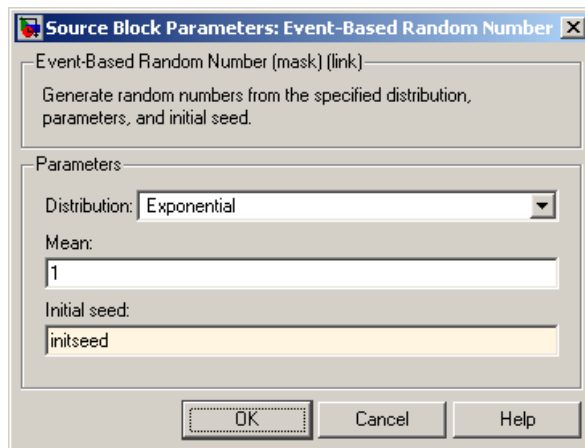
When running a simulation interactively, you can simply enter your chosen value for the initial seed in the dialog box for a given block.

When you run a simulation multiple times to gather statistics, you might want to use a MATLAB variable to make it easier to assign a different seed each time you run the simulation. Follow this procedure for each **Initial seed** parameter that you need to set:

- 1 Assign a value to the MATLAB variable. For example, the code below defines a variable called `initseed`.

```
initseed = 68521;
```

- 2 In the **Initial seed** field of the dialog box, enter the name of the MATLAB variable. An example is below.



- 3** Run the simulation using **Simulation > Start** or the `sim` function. During the simulation, the value that the block uses for the initial seed is the one that you assigned to the MATLAB variable.
- 4** Repeat steps 1 and 3 if appropriate, using a different value each time.

If you are using the above procedure within an M-file that runs a series of simulations in a loop, put the variable assignment and the simulation command (`sim`) in the M-file. For an example of this approach, see “Example: Running a Simulation Repeatedly to Gather Results” on page 11-28.

Running Simulations from MATLAB

When you analyze simulation statistics, you typically need to run the simulation many times. One simulation run, no matter how long, is still a single sample and probably inadequate for valid statistical analysis. The `sim` function enables you to run simulations unattended, while the Discrete Event Signal to Workspace block writes signal contents to the MATLAB workspace for subsequent storage or analysis. To learn about this function and block, see “Running a Simulation Programmatically” in the Simulink documentation and the reference page for the Discrete Event Signal to Workspace block. This section provides examples and tips that focus on SimEvents simulations, in these topics:

- “Example: Running a Simulation Repeatedly to Gather Results” on page 11-28
- “Example: Running a Simulation and Varying a Parameter” on page 11-30

Example: Running a Simulation Repeatedly to Gather Results

Suppose you want to run the M/D/1 Queuing System demo model many times to check whether the ensemble average for the waiting time is close to the value predicted by queuing theory. You can do this by modifying the model to make it suitable for statistical analysis, simulating the modified model multiple times, and then analyzing the results.

- 1** Open the model by entering `sedemo_md1` in the MATLAB Command Window.

- 2 Save the model as `md1_stats.mdl` in either the current directory or a writable directory on the MATLAB path.
- 3 From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model.
- 4 In the Discrete Event Signal to Workspace block's 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.
- 5 Create a branch line from the output of the subsystem labeled Waiting Time Evaluation and connect the branch line to the Discrete Event Signal to Workspace block.
- 6 In the Exponential Interarrival Time Distribution block's dialog box, set **Initial seed** to `seedvalue`. The use of a variable facilitates using a different set of random numbers in each run. However, do not run the simulation yet because `seedvalue` is not defined.
- 7 Resave the model, which is now suitable for statistical analysis of the waiting time.
- 8 To run the simulation repeatedly and capture the statistic from each run, execute the following code in MATLAB.

```
% Set up.
load_system('md1_stats'); % Load system if not already loaded.
nruns = 100; % Number of simulation runs
w = zeros(nruns,1); % Preallocate space for results
opts = simset('SrcWorkspace','Current','DstWorkspace','Current');
h = waitbar(0,'Running simulations. Please wait...');
seedarray = ceil(rand(nruns,1)*999999)*2+1;

% Main simulation loop
for k = 1:nruns
    waitbar(k/nruns,h); % Update progress indicator.
    seedvalue = seedarray(k);
    sim('md1_stats',[],opts); % Run simulation.
    w(k) = simout(2); % Store empirical value of w.
```

```
end
```

```
w_theor = simout(1); % Store theoretical value.  
close(h); % Close progress indicator window.
```

- 9 To display some information about the collected statistics, enter the following code in the MATLAB Command Window.

```
format long; % Show more digits in Command Window output.  
disp('Theoretical value and ensemble average are:');  
disp([w_theor, mean(w)]);  
disp('Standard deviation for empirical values is:');  
disp(std(w));  
disp('Relative error in ensemble average is:');  
disp([num2str(100*abs(w_theor - mean(w))/w_theor), '%']);
```

Sample output is below. Your results might vary because the initial seed for the random number generator is itself random.

```
Theoretical value and ensemble average are:  
0.333333333333333 0.33163393290150
```

```
Standard deviation for empirical values is:  
0.01961357797243
```

```
Relative error in ensemble average is:  
0.50982%
```

Example: Running a Simulation and Varying a Parameter

Suppose you want to run the M/D/1 Queuing System demo model with different values of the mean arrival rate. You can do this by modifying the model to make it suitable for statistical analysis with a varying parameter, simulating the modified model multiple times with different values of the parameter, and then analyzing the results. Part of the procedure below is similar to the one in “Example: Running a Simulation Repeatedly to Gather Results” on page 11-28; however, the filename, MATLAB code, and instructions regarding the Constant and Arrival Rate Gain blocks are different.

- 1 Open the model by entering `sedemo_md1` in the MATLAB Command Window.
- 2 Save the model as `md1_varymean.mdl` in either the current directory or a writable directory on the MATLAB path.
- 3 From the SimEvents Sinks library, drag the Discrete Event Signal to Workspace block into the model.
- 4 In the Discrete Event Signal to Workspace block's 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.
- 5 Create a branch line from the output of the subsystem labeled Waiting Time Evaluation and connect the branch line to the Discrete Event Signal to Workspace block.
- 6 In the Exponential Interarrival Time Distribution block's dialog box, set **Initial seed** to `ceil(cputime*99999)*2+1`. This causes the simulation to use a different set of random numbers in each run, although the results are not repeatable.
- 7 Remove the Arrival Rate Gain block and close the connection gap between the Constant block (labeled Maximal Arrival Rate, but now signifying the mean arrival rate) and the subsystem labeled Exponential Interarrival Time Distribution.
- 8 Open the dialog box of the Constant block labeled Maximal Arrival Rate and set **Constant value** to `m`, which is a MATLAB variable to be defined later.
- 9 Remove the model's callbacks that are not relevant for this modified version, by executing the following in MATLAB.

```
set_param('md1_varymean', 'PostLoadFcn', '')
set_param('md1_varymean', 'CloseFcn', '')
```
- 10 Resave the model, which is now suitable for statistical analysis of the waiting time with varying mean arrival rates.

- 11** To run the simulation repeatedly while varying the mean arrival rate, and capture the statistic from each run, execute the following code in MATLAB. Note that it takes some time to run.

```
% Set up.
load_system('mdl_varymean'); % Load system if not yet loaded.
nruns = 100; % Number of simulation runs
mvec = (0.2 : 0.1 : 0.4); % Values of mean arrival rate
nm = length(mvec); % Number of values in mvec
opts = simset('SrcWorkspace','Current','DstWorkspace','Current');

% Preallocate space for results.
w = zeros(nruns,1);
wavg = zeros(nm,1);
w_theor = zeros(nm,1);

% Vary the mean arrival rate.
for midx = 1:nm
    % m is a parameter in the Constant block, so changing m
    % changes the mean arrival rate in the simulation.
    m = mvec(midx);
    disp(['Simulating with mean arrival rate=' num2str(m)]);

    % Replicate for each value of m
    for k = 1:nruns
        sim('mdl_varymean',[],opts); % Run simulation.
        w(k) = simout(2); % Store empirical value of w.
    end

    wavg(midx) = mean(w); % Average for fixed value of m
    w_theor(midx) = simout(1); % Theoretical value
end
```

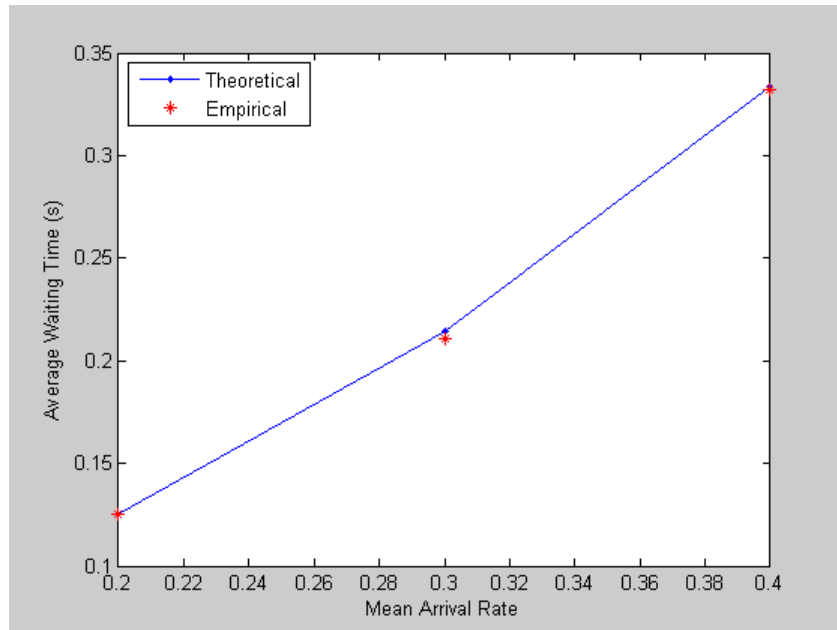
- 12** To plot the average waiting time (averaged over multiple simulations for each fixed value of mean arrival rate) against mean arrival rate, execute the following code in MATLAB.

```
figure; % Create new figure window.
plot(mvec,w_theor,'b.-'); % Plot theoretical curve.
hold on;
plot(mvec,wavg,'r*'); % Plot empirical values.
```



```
legend('Theoretical','Empirical','Location','NorthWest')
xlabel('Mean Arrival Rate')
ylabel('Average Waiting Time (s)')
hold off
```

A sample plot is below.



Regulating the Simulation Length

When you run a simulation interactively to observe behavior qualitatively, the stop time of the simulation might not matter. However, when you need to gather statistics from a simulation, knowing when to end the simulation is more important. Typical criteria for ending a discrete-event simulation include the following:

- A fixed amount of time passes
- The cumulative number of entity departures from a particular block crosses a fixed threshold. This might be analogous to processing a fixed number of packets, parts, or customers, for example.

- 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 from MATLAB, use syntax like

```
sim('model',timespan)
```

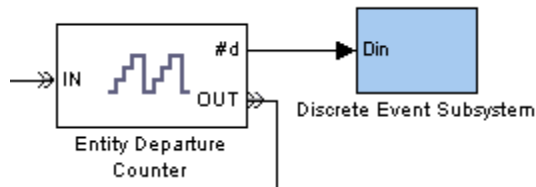
where `model` is the name of the model and `timespan` is the desired stop time.

Stopping Based on Entity Count

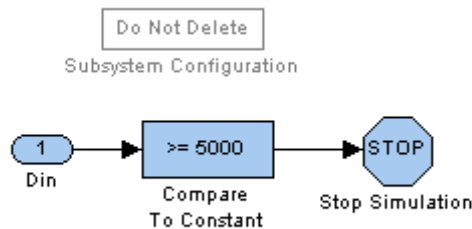
The basic procedure for stopping a simulation based on the total number of entity departures is as follows:

- 1** Find the block's parameter that enables the departure counter as a signal output, usually called **Number of entities departed**. Exceptions are the **Write count to signal port #d** parameter of the Entity Departure Counter block and the **Number of entities arrived** parameter of the Entity Sink block.
- 2** Set the parameter to On. This 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 Outport 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 Simulink.
- 7 In the Compare To Constant block,
 - Set **Operator** to \geq .
 - 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 Sinks library in Simulink. 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

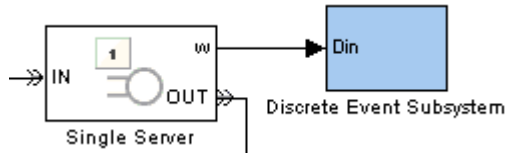
See the considerations discussed in “Tips for Using State-Based Stopping Conditions” on page 11-39 below. They are relevant if you are stopping the simulation based on an entity count, where “desired state” means the entity-count threshold.

Stopping Upon Reaching a Particular State

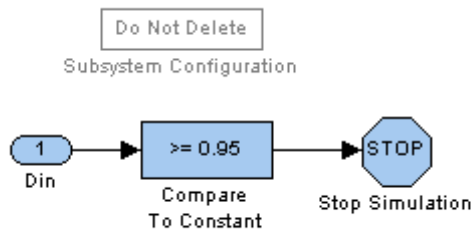
Suppose you want the simulation to end when it achieves a particular state, such as an overflow or a machine failure. The state might be the only criterion for ending the simulation, or the state might be one of multiple criteria, each of which is sufficient reason to end the simulation. An example that 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 Logic and Bit Operations library in Simulink. The figure below illustrates one possibility.

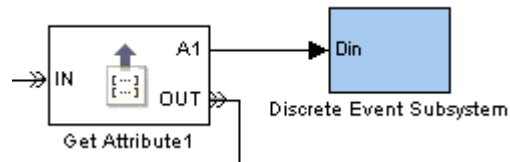


Top-Level Model

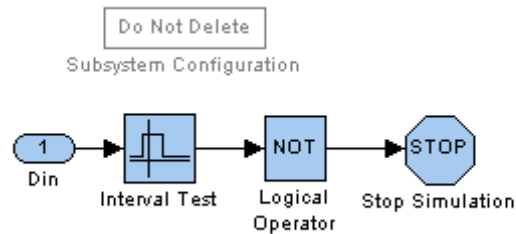


Subsystem Contents

- 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.

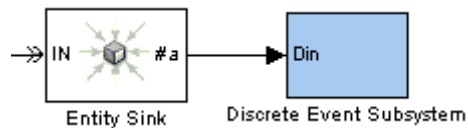


Top-Level Model

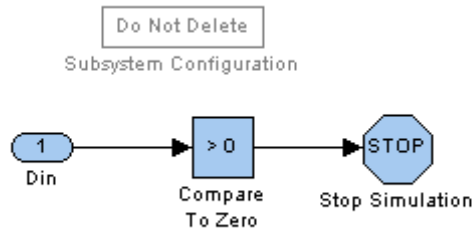


Subsystem Contents

- 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.

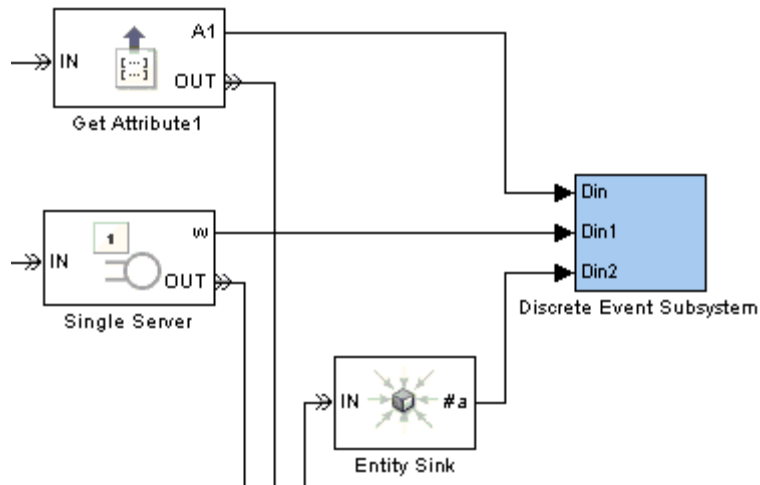


Top-Level Model

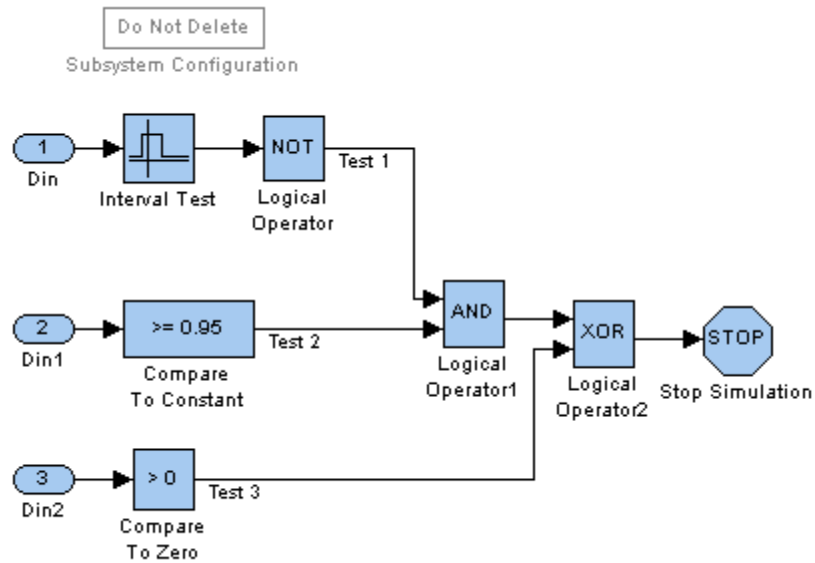


Subsystem Contents

- 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 11-34 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 with SimEvents

Role of Stateflow in SimEvents Models (p. 12-2)

Guidelines for Using Stateflow and SimEvents Blocks (p. 12-3)

Examples Using Stateflow and SimEvents Blocks (p. 12-5)

Overview of uses of Stateflow in discrete-event system modeling

Rules for models that use both Stateflow and SimEvents blocks

Stateflow usage in some SimEvents demo and example models

Role of Stateflow in SimEvents Models

SimEvents works with Stateflow to represent systems containing state-transition diagrams that can produce or be controlled by discrete events. SimEvents and Stateflow are both 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.
- 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 and SimEvents Blocks” on page 12-5.

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 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:
 - a** Select the Stateflow block and choose **Edit > Subsystem Parameters** from the model window's menu bar.
 - b** 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,
 - a** Select the **File > Chart Properties** from the chart window's menu bar.
 - b** 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 4-17 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 Signal Latch block as in “Specifying Initial Conditions for Event-Based Signals” on page 3-27. If you do this, you should select **Resolve simultaneous signal updates according to event priority** on the **Write** tab of the Signal Latch block. Because the chart is not entered at initialization, you cannot use the chart itself to provide a nonzero initial condition for the output signal.

- If the chart has both a function-call output and a signal output, be aware of possible latency in the signal. You might need to specify an event priority in a SimEvents block that reacts to the function call, as a way to ensure that the reaction to the function call occurs after the update of the chart's output signal has been fully processed in the model.

Examples Using Stateflow and SimEvents Blocks

This section describes some Stateflow usage in SimEvents demo and example models. The topics are

- “Failure State of Server” on page 12-5
- “Go-Back-N ARQ Model” on page 12-5

Failure State of Server

The examples in “Using Stateflow to Implement a Failure State” on page 4-16 use Stateflow 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 Stateflow approach is more intuitive and scales more easily to include additional complexity.

Go-Back-N ARQ Model

The Go-Back-N Automatic Repeat Request (ARQ) 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).

Troubleshooting Discrete-Event Simulations

Viewing the Event Calendar (p. 13-2)	Logging information about events
Viewing Entity Locations (p. 13-9)	Logging information about entities advancing from block to block
Common Problems in SimEvents Models (p. 13-13)	Some modeling errors and ways to avoid them
Configuration Parameters for SimEvents Models (p. 13-28)	SimEvents parameters in the Configuration Parameters dialog box

Additional troubleshooting techniques are in “Using Plots for Troubleshooting” on page 10-9.

Viewing the Event Calendar

Knowing which events are on the event calendar at relevant times during the simulation can help you learn and troubleshoot. SimEvents models offer an option to have the MATLAB Command Window tell you

- When each event is placed on the event calendar
- When each event is processed
- The list of events on the event calendar

This section describes how to enable this option and interpret the information. The topics are as follows:

- “Turning Event Logging On” on page 13-2
- “Logging the Processing of Events” on page 13-3
- “Logging the Scheduling of Events” on page 13-4
- “Logging the List of Events” on page 13-5
- “Example: Event Logging” on page 13-6

To create a file containing messages that appear in the MATLAB Command Window, use the `diary` function.

Turning Event Logging On

To enable event logging for a particular model that contains one or more blocks from the SimEvents libraries, use this procedure:

- 1** Select **Simulation > Configuration Parameters** from the model window’s menu to open the Configuration Parameters dialog box.
- 2** Click **SimEvents** on the left side of the Configuration Parameters dialog box.
- 3** Select one or more of the following options on the right side of the dialog box:
 - **Display events in event calendar**
 - **Log events when executed**

- **Log events when scheduled**

Selecting **Display events in event calendar** disables the other two options because displaying the events in the event calendar automatically includes logging events when scheduled and executed. For details, see “Logging the List of Events” on page 13-5.

The next time you run the simulation, the MATLAB Command Window displays information about the event calendar.

Logging the Processing of Events

When you select **Log events when executed** as described in “Turning Event Logging On” on page 13-2 and run the simulation, the MATLAB Command Window displays a message like the following each time an event is processed:

```
SimEvents: Executing @ 1.0000000000000000 (T=1.0000000000000000
P=300 B='mymodel/Time-Based Entity Generator' N='EntityGeneration')
```

This indicates that the Time-Based Entity Generator block in a model called `mymodel` generates an entity at time 1 and that this event has event priority 300.

The table below lists the pieces of information contained in messages like this.

Portion of Message	Description
Executing	Distinguishes this message from a scheduling message described in “Logging the Scheduling of Events” on page 13-4
T= followed by a number	Simulation time at which the event is processed
P= followed by a number	Event priority of the event
B= followed by a block pathname	Block that processes the event
N= followed by a string	Name of the event, such as <code>EntityGeneration</code> or <code>ServiceCompletion</code>

For a Discrete Event Inport block whose name is blockname, the B= portion of the message uses f_blockname instead of blockname.

Logging the Scheduling of Events

When you select **Log events when scheduled** as described in “Turning Event Logging On” on page 13-2 and run the simulation, the MATLAB Command Window displays a message like the following each time a new event appears on the event calendar:

```
SimEvents: Scheduling @ 0.000000000000000 (T=1.000000000000000
P=300 B='mymodel/Time-Based Entity Generator' N='EntityGeneration')
```

This indicates that at time 0, the Time-Based Entity Generator block in a model called mymodel schedules an entity generation to occur at time 1 and that this event has event priority 300.

The table below lists the pieces of information contained in messages like this.

Portion of Message	Description
Scheduling or Deleting	Distinguishes this message from an execution message described in “Logging the Processing of Events” on page 13-3
@ followed by a number	Simulation time at which the scheduling occurs
T= followed by a number	Simulation time at which the event occurs
P= followed by a number	Event priority of the event
B= followed by a block pathname	Block that schedules the event
N= followed by a string	Name of the event, such as EntityGeneration or ServiceCompletion

The word **Deleting** instead of **Scheduling** describes the deletion of timeout events from the event calendar when an entity arrives at the Cancel Timeout block.

The processing sequence and the scheduling sequence might differ for simultaneous events having equal priority.

For a Discrete Event Inport block whose name is `blockname`, the `B=` portion of the message uses `f_blockname` instead of `blockname`.

Logging the List of Events

When you select **Display events in event calendar** as described in “Turning Event Logging On” on page 13-2 and run the simulation, the MATLAB Command Window displays a message like the following each time a new event appears on the event calendar or is processed:

```
SimEvents: Scheduling @ 2.000000000000000 (T=3.000000000000000
P=300 B='mymodel/Time-Based Entity Generator' N='EntityGeneration')
%BEGIN list event in calendar @ 2.000000000000000
SimEvents: Event in calendar @ 2.000000000000000 (T=2.300000000000000
P=500 B='mymodel/Single Server' N='ServiceCompletion')
SimEvents: Event in calendar @ 2.000000000000000 (T=3.000000000000000
P=300 B='mymodel/Time-Based Entity Generator' N='EntityGeneration')
%END list event in calendar @ 2.000000000000000
```

The message has these key components:

- A line that begins with `SimEvents: Scheduling` or `SimEvents: Deleting` is a scheduling message as described in “Logging the Scheduling of Events” on page 13-4. The excerpt above indicates that at time 2, the Time-Based Entity Generator block in a model called `mymodel` schedules an entity generation to occur at time 3 and that this event has event priority 300.

Alternatively, a line that begins with `SimEvents: Executing` is an event processing message as described in “Logging the Processing of Events” on page 13-3.

- Lines between `%BEGIN` and `%END` form a list of all events on the event calendar at a given time, after the scheduling or execution mentioned in the other part of the message has occurred.

The table below describes the pieces of information contained in the list of all events.

Portion of Message	Description
@ followed by a number	Simulation time at which the event calendar is being displayed
T= followed by a number	Simulation time at which the event occurs
P= followed by a number	Event priority of the event
B= followed by a block pathname	Block that processes the event or, in the case of a scheduled timeout event, the last storage block that held the entity
N= followed by a string	Name of the event, such as EntityGeneration or ServiceCompletion

For a Discrete Event Inport block whose name is blockname, the B= portion of the message uses f_blockname instead of blockname.

Example: Event Logging

You can view the event calendar from the first few seconds of simulation of the M/M/1 Queuing System demo using this procedure:

- 1** Open the demo by entering `sedemo_mm1` in the MATLAB Command Window or by using the MATLAB Help browser.
- 2** Select **Simulation > Configuration Parameters** from the model window's menu to open the Configuration Parameters dialog box.
- 3** Click **SimEvents** on the left side of the Configuration Parameters dialog box.
- 4** Select **Display events in event calendar** on the right side of the dialog box and click **OK**.
- 5** Run the simulation for 3 seconds by entering the following in the MATLAB Command Window:

```
sim('sedemo_mm1',3);
```

Interpreting the Event Logging Messages

The resulting messages in the MATLAB Command Window show you the state of the event calendar during the simulation:

- The first entity generation event is scheduled at time 0, to occur at time 0. This is because the entity generator has the **Generate entity at simulation start** option selected. The event calendar contains the entity generation event.

```
SimEvents: Scheduling @ 0.000000000000000 (T=0.000000000000000)
P=1 B='sedemo_mm1/Time-Based Entity Generator' N='EntityGeneration')
%BEGIN list event in calendar @ 0.000000000000000
SimEvents: Event in calendar @ 0.000000000000000 (T=0.000000000000000)
P=1 B='sedemo_mm1/Time-Based Entity Generator')
%END list event in calendar @ 0.000000000000000
```

- The entity generation event is processed, leaving the event calendar empty.

```
SimEvents: Executing @ 0.000000000000000 (T=0.000000000000000)
P=1 B='sedemo_mm1/Time-Based Entity Generator' N='EntityGeneration')
%BEGIN list event in calendar @ 0.000000000000000
%END list event in calendar @ 0.000000000000000
```

- The entity advances immediately to the server, which schedules a service completion event. The event calendar contains the service completion event.

```
SimEvents: Scheduling @ 0.000000000000000 (T=2.991406386946900)
P=1 B='sedemo_mm1/Single Server' N='ServiceCompletion')
%BEGIN list event in calendar @ 0.000000000000000
SimEvents: Event in calendar @ 0.000000000000000 (T=2.991406386946900)
P=1 B='sedemo_mm1/Single Server')
%END list event in calendar @ 0.000000000000000
```

- The second entity generation event is scheduled for a future time. The event calendar contains two events: the service completion event and the entity generation event.

```
SimEvents: Scheduling @ 0.000000000000000 (T=3.184988194595833)
P=1 B='sedemo_mm1/Time-Based Entity Generator' N='EntityGeneration')
%BEGIN list event in calendar @ 0.000000000000000
SimEvents: Event in calendar @ 0.000000000000000 (T=2.991406386946900)
```

```
P=1 B='sedemo_mm1/Single Server')
SimEvents: Event in calendar @ 0.0000000000000000 (T=3.184988194595833
P=1 B='sedemo_mm1/Time-Based Entity Generator')
%END list event in calendar @ 0.0000000000000000
```

- The service completion event is processed, leaving only the entity generation event on the event calendar.

```
SimEvents: Executing @ 2.991406386946900 (T=2.991406386946900
P=1 B='sedemo_mm1/Single Server' N='ServiceCompletion')
%BEGIN list event in calendar @ 2.991406386946900
SimEvents: Event in calendar @ 2.991406386946900 (T=3.184988194595833
P=1 B='sedemo_mm1/Time-Based Entity Generator')
%END list event in calendar @ 2.991406386946900
```

Viewing Entity Locations

Knowing when an entity departs from one block and arrives at another block during the simulation can help you learn and troubleshoot. SimEvents models offer an option to have the MATLAB Command Window display information about entity locations. This section describes how to enable this option and interpret the information. The topics are as follows:

- “Turning Entity Logging On” on page 13-9
- “Interpreting Entity Logging Messages” on page 13-9
- “Example: Entity Logging” on page 13-10

To create a file containing messages that appear in the MATLAB Command Window, use the `diary` function.

Turning Entity Logging On

To enable entity logging for a particular model that contains one or more blocks from the SimEvents libraries, use this procedure:

- 1** Select **Simulation > Configuration Parameters** from the model window’s menu to open the Configuration Parameters dialog box.
- 2** Click **SimEvents** on the left side of the Configuration Parameters dialog box.
- 3** Select **Log entities advancing from block to block** on the right side of the dialog box:

The next time you run the simulation, the MATLAB Command Window displays information about entities advancing from block to block.

Interpreting Entity Logging Messages

When you select **Log entities advancing from block to block** as described in “Turning Entity Logging On” on page 13-9 and run the simulation, the MATLAB Command Window displays a message like the following each time an entity departs from one block and advances to another block:

```
SimEvents: Entity advancing @ 0.0000000000000000
(From='mymodel/Time-Based Entity Generator' To='mymodel/FIFO Queue')
```

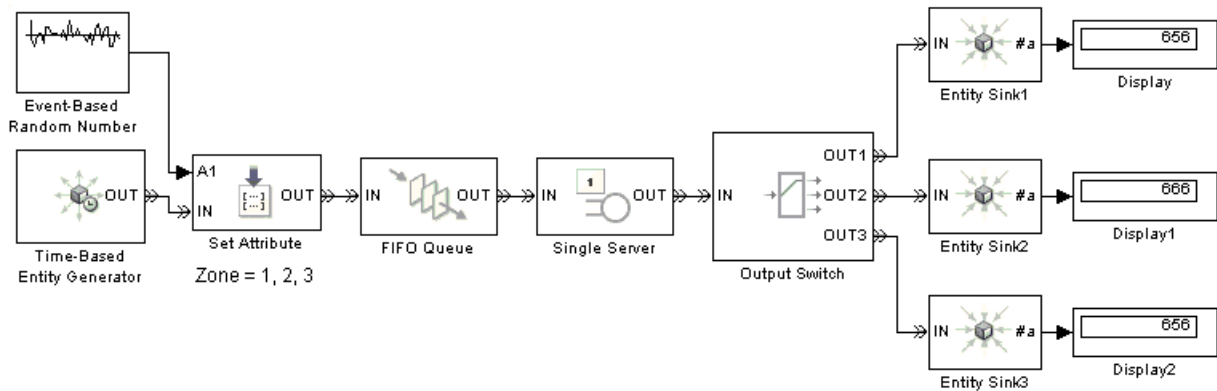
This indicates that at time 0, an entity departs from the Time-Based Entity Generator block and arrives at the FIFO Queue block, in a model called mymodel.

The table below lists the pieces of information contained in messages like this.

Portion of Message	Description
@ followed by a number	Simulation time at which the entity advances
From= followed by a block pathname	Block from which the entity departs
To= followed by a block pathname	Block at which the entity arrives

Example: Entity Logging

By viewing entity locations, you can get information about the model described in “Example: Using an Attribute to Select an Output Port” in the getting started documentation.



- At the beginning of the simulation, the first entity advances from the entity generator to the server.

```
SimEvents: Entity Advancing @ 0.0000000000000000
(From='doc_outsw_attr/Time-Based Entity Generator' To='doc_outsw_attr/Set Attribute')
SimEvents: Entity Advancing @ 0.0000000000000000
(From='doc_outsw_attr/Set Attribute' To='doc_outsw_attr/FIFO Queue')
SimEvents: Entity Advancing @ 0.0000000000000000
(From='doc_outsw_attr/FIFO Queue' To='doc_outsw_attr/Single Server')
```

- After completing its service, the first entity departs from the server and is routed to Entity Sink1.

```
SimEvents: Entity Advancing @ 1.0000000000000000
(From='doc_outsw_attr/Single Server' To='doc_outsw_attr/Output Switch')
SimEvents: Entity Advancing @ 1.0000000000000000
(From='doc_outsw_attr/Output Switch' To='doc_outsw_attr/Entity Sink1')
```

- The second entity advances from the entity generator to the server.

```
SimEvents: Entity Advancing @ 4.898639080694728
(From='doc_outsw_attr/Time-Based Entity Generator' To='doc_outsw_attr/Set Attribute')
SimEvents: Entity Advancing @ 4.898639080694728
(From='doc_outsw_attr/Set Attribute' To='doc_outsw_attr/FIFO Queue')
SimEvents: Entity Advancing @ 4.898639080694728
(From='doc_outsw_attr/FIFO Queue' To='doc_outsw_attr/Single Server')
```

- The third entity advances from the entity generator to the queue. (The third entity cannot advance to the server because the server is busy serving the second entity.)

```
SimEvents: Entity Advancing @ 5.234759100998515
(From='doc_outsw_attr/Time-Based Entity Generator' To='doc_outsw_attr/Set Attribute')
SimEvents: Entity Advancing @ 5.234759100998515
(From='doc_outsw_attr/Set Attribute' To='doc_outsw_attr/FIFO Queue')
```

- At the end of its service time, the second entity departs from the server and is routed to Entity Sink2. As a result, the third entity advances from the queue to the server.

```
SimEvents: Entity Advancing @ 5.898639080694728
(From='doc_outsw_attr/Single Server' To='doc_outsw_attr/Output Switch')
```

```
SimEvents: Entity Advancing @ 5.898639080694728  
(From='doc_outsw_attr/Output Switch' To='doc_outsw_attr/Entity Sink2')  
SimEvents: Entity Advancing @ 5.898639080694728  
(From='doc_outsw_attr/FIFO Queue' To='doc_outsw_attr/Single Server')
```

The entity logging messages do not count or otherwise identify which entity is advancing. The descriptions above indicate when the first, second, or third entity is the one that advances because such inferences are straightforward in this example.

Common Problems in SimEvents Models

Troubleshooting a discrete-event simulation can be challenging because blocks that form an entity path operate in coupled ways. The block whose behavior surprises you might not be the source of a mistake. For example, after troubleshooting a surprising set of values in the **#d** output signal from a server, you might find that the problem is not in the server itself but rather in the configuration of a gate or switch block in another part of the model.

Some common problems relate to the simultaneity of events and the sequence in which the events are processed. When events occur simultaneously, it is because they have a causal relationship to each other or because their occurrence times happen to be equal or close enough.

This section describes some common problems. Specific symptoms and fixes are difficult to generalize, but this section offers examples or tips where feasible.

The problems are presented in these topics:

- “Unexpectedly Simultaneous Events” on page 13-13
- “Unexpectedly Nonsimultaneous Events” on page 13-14
- “Unexpected Processing Sequence for Simultaneous Events” on page 13-14
- “Time-Based Block Not Recognizing Certain Trigger Edges” on page 13-15
- “Incorrect Timing of Signals” on page 13-15
- “Unexpected Use of Old Value of Signal” on page 13-17
- “Effect of Initial Condition on Signal Loops” on page 13-21
- “Loops in Entity Paths Without Storage Blocks” on page 13-23
- “Unexpected Timing of Random Signal” on page 13-24
- “Unexpected Correlation of Random Processes” on page 13-26

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 $T+dt$ are considered simultaneous if $0 \leq dt \leq 128 \cdot \text{eps} \cdot T$ where eps is the floating-point relative accuracy in MATLAB and T is the simulation time.

If you have a guess about which events' processing is suspect, then adjusting event priorities or using the Instantaneous Event Counting Scope block can help diagnose the problem. For examples involving event priorities, see "Example: Race Conditions at a Switch" on page 2-25 and the Event Priorities demo. For an example using the Instantaneous Event Counting Scope block, see "Example: Counting Events from Multiple Sources" on page 2-48.

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 $T+dt$ are considered simultaneous if $0 \leq dt \leq 128 \cdot \text{eps} \cdot T$ where eps is the floating-point relative accuracy in MATLAB and T is the simulation time.

If roundoff error is very small, then the event logging feature and 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 MATLAB as in the example below.

If your model requires that certain events be simultaneous, then use modeling techniques aimed at effecting simultaneity. For an example, see "Example: Race Conditions at a Switch" on page 2-25.

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, mentioned in "Processing Sequence for Simultaneous Events" on page 2-11. The sequence might even change when you run the simulation again. When the sequence is arbitrary, you should 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 14-8.

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, then the blocks might not run when you expect them to. For more information, suggestions, and an example, see “Zero-Duration Values and Time-Based Blocks” on page 14-17.

Incorrect Timing of Signals

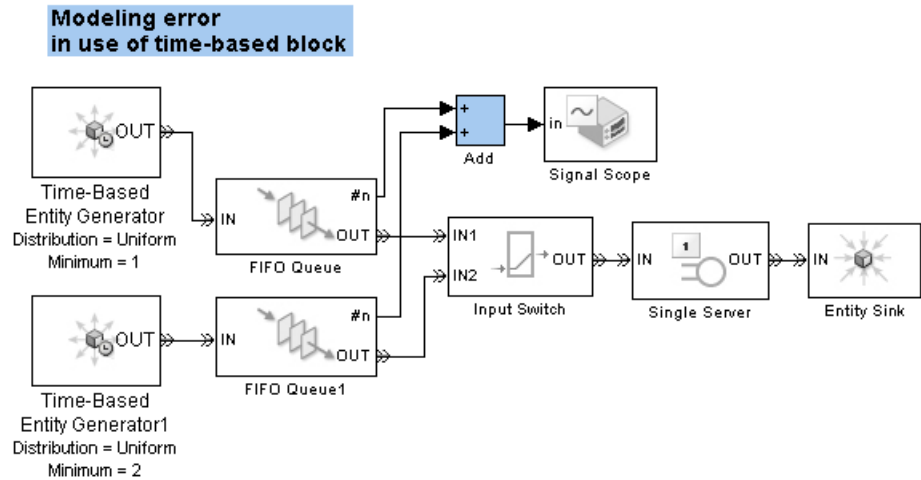
If you use a time-based block to process event-based signals, then the output signal might be a time-based signal. Depending on your model, you might notice that

- 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.

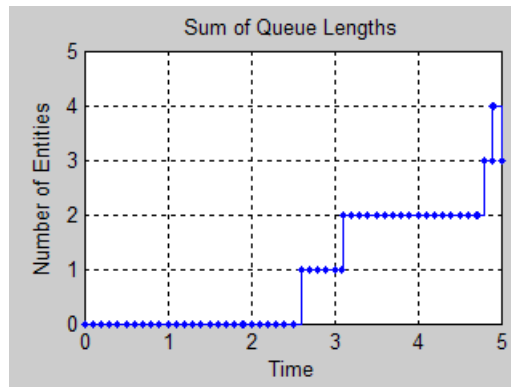
You can avoid these problems by putting the time-based block in a discrete event subsystem, as described in Chapter 9, “Controlling Timing with Subsystems”. If your time-based block is in a Function-Call Subsystem, then be sure to select **Propagate execution context across subsystem boundary** as described in “Setting Up Function-Call Subsystems in SimEvents Models” on page 9-34.

Example: Time-Based Addition of Event-Based Signals

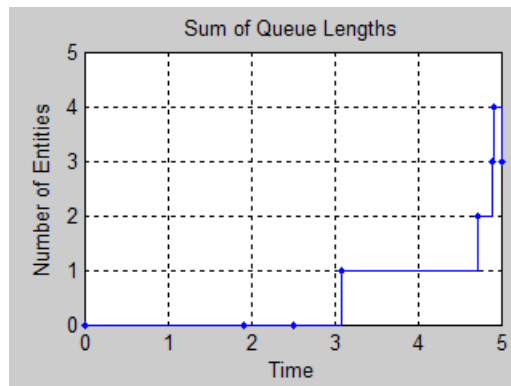
The model below adds the lengths of two queues. The queue lengths are event-based signals, while the Add block is a time-based block. It is important that the Add block use up-to-date values of its input signals each time the queue length changes and that the output signal's updates correspond to updates in one of the queue length signals.



If you build this model without having used `simeventsstartup` previously, or without using `simeventsconfig` later, then you might see the plot below. 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), then the plot reveals correct update times but incorrect values. To check 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, then 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 finished advancing to a subsequent storage block, 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 an example, see “Example: Using a Signal or an Attribute” on page 13-19. For details, see “Interleaving of Block Operations” on page 14-8. 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 5-2.

- 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 3-18.

- An inappropriate processing sequence for simultaneous events causes a signal update to occur after a block uses that signal in a computation or to control an operation. See the example below.

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 5-2.

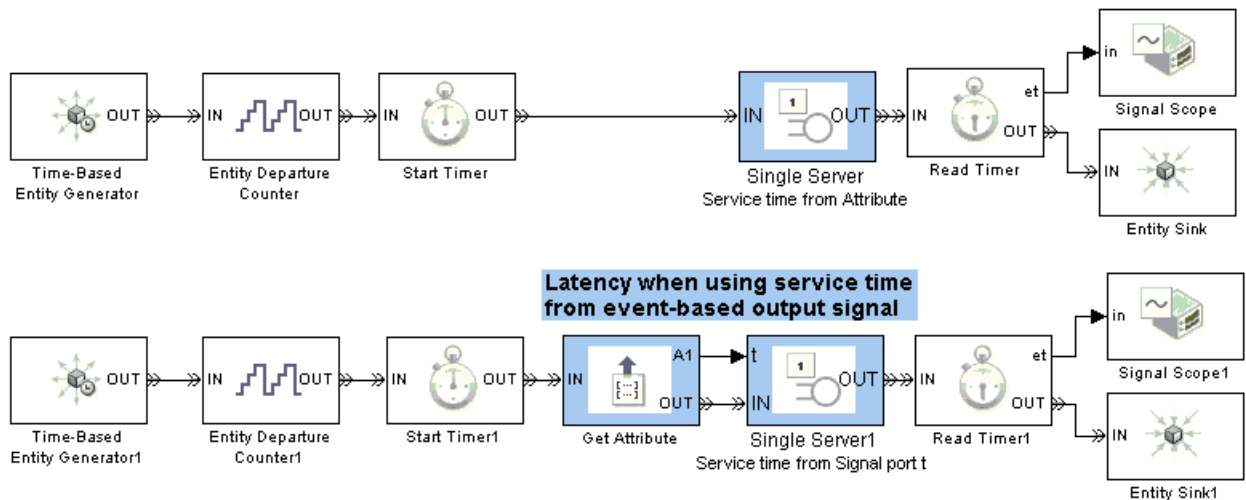
- A time-based block’s use of a value of an event-based signal persists until the next time step of the time-based simulation clock, even if the block producing the event-based signal has already updated the value. In many cases, this is the correct behavior of the time-based block.

For an example, see “Example: Plotting Entity Departures to Verify Timing” on page 10-10.

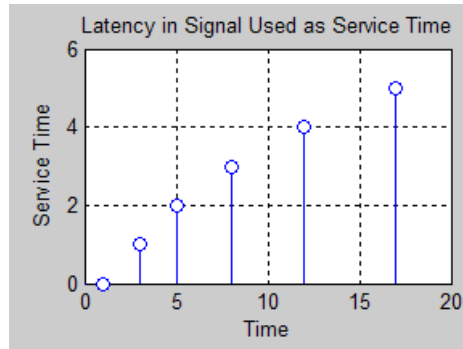
If you need a time-based block to respond to events, consider using a discrete event subsystem as described in Chapter 9, “Controlling Timing with Subsystems”.

Example: Using a Signal or an Attribute

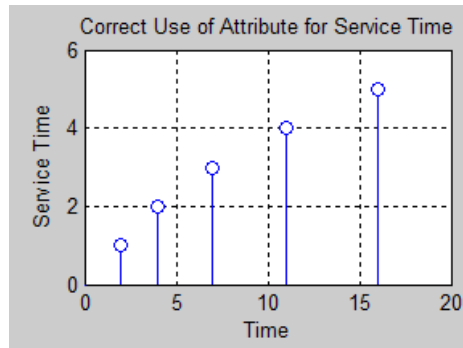
The goal in the next model is to use a service time of N seconds for the N th entity. The Entity Counter block stores each entity's index, N , in an attribute. The top portion of the model uses the attribute directly to specify the service time, while the bottom portion creates a signal representing the attribute value and attempts to use the signal to specify the service time. These might appear to be equivalent approaches, but in fact only the top approach satisfies the goal.



The plot of the time in the bottom server block reveals 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 14-8.



The top portion of the model, where the server directly uses the attribute of each arriving entity, uses the expected service times. The phenomenon involving sequential processing of an entity departure and a signal update does not occur here 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, it is usually better to use the attribute directly than to create a signal with the attribute's value and ensure that the signal is up-to-date when the entity arrives.

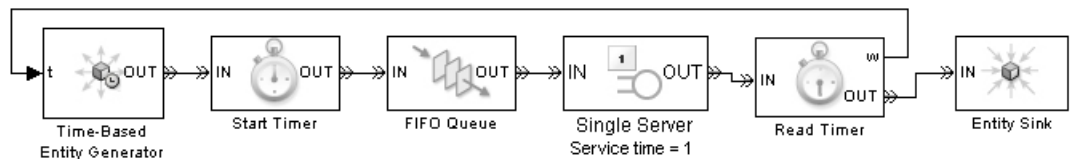
Effect of Initial Condition on Signal Loops

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 Conditions for Event-Based Signals” on page 3-27.

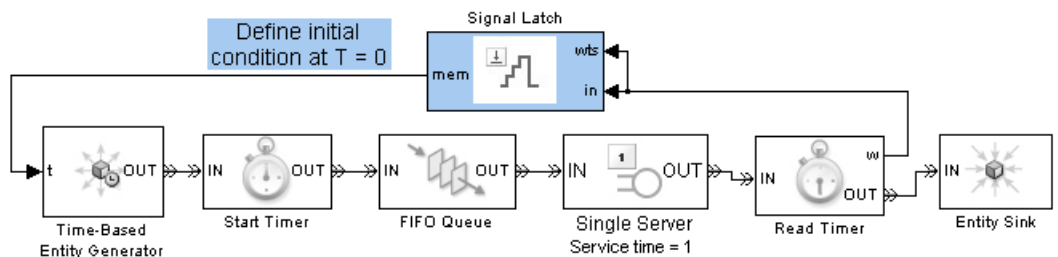
Example: Intergeneration Time of Zero at Simulation Start

The model below 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 $T = 0$



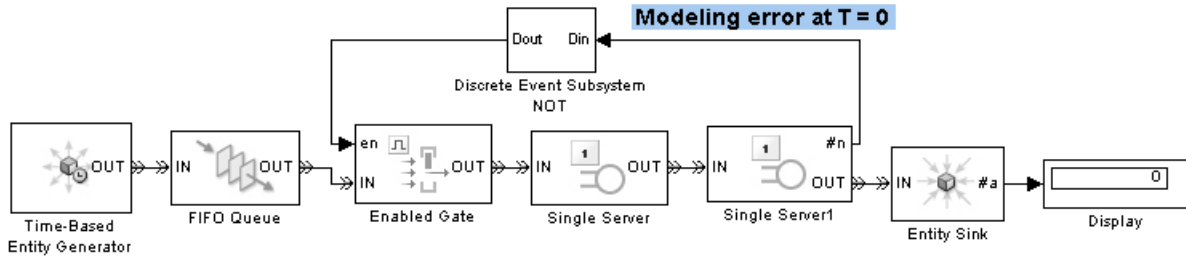
A better model would use the technique described in “Specifying Initial Conditions for Event-Based Signals” on page 3-27 to specify a nonzero initial condition for the w output signal from the Read Timer block.



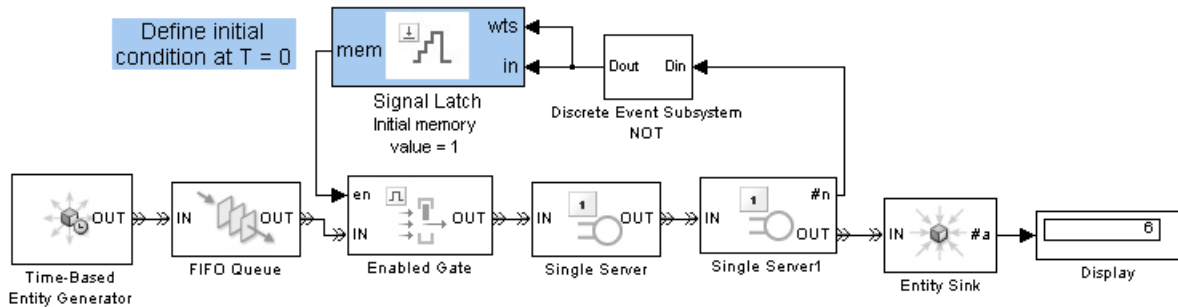
Example: Absence of Sample Time Hit at Simulation Start

In the model below, 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.

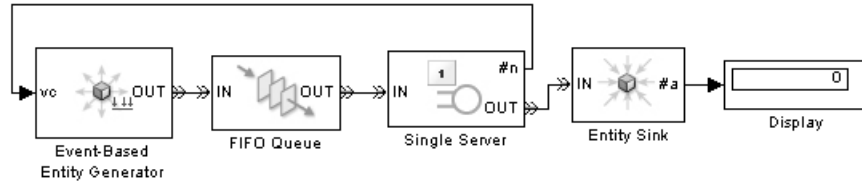


A better model would use the technique described in “Specifying Initial Conditions for Event-Based Signals” on page 3-27 to define a positive initial condition for the **en** input signal to the gate.

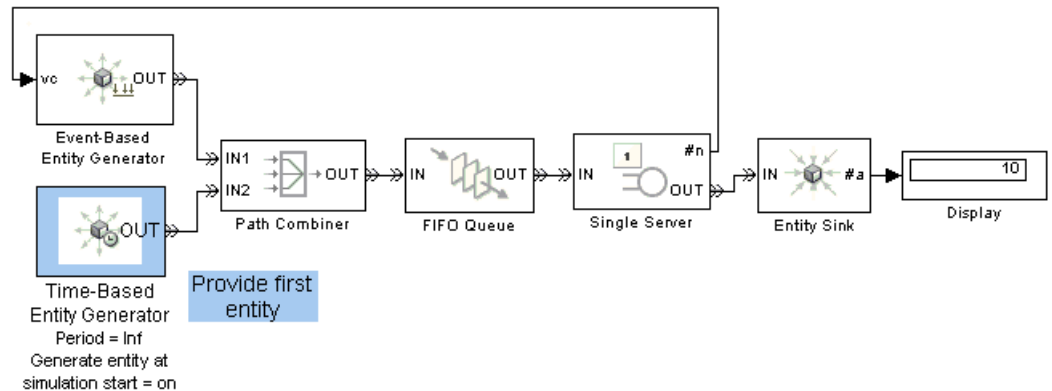


Example: Faulty Logic in Feedback Loop

The model below 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

A better model would provide the first entity in a separate path. In the revised model below, the Time-Based Entity Generator block generates exactly one entity during the simulation, at T=0.

**Loops in Entity Paths Without Storage Blocks**

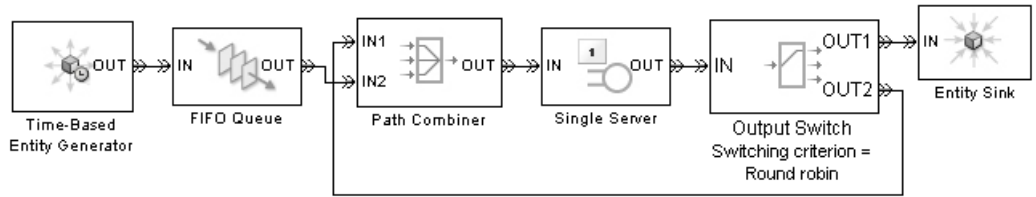
An entity path that forms a loop should contain a storage block. Storage blocks include queues and servers; for a list of storage blocks, see “Storage and Nonstorage Blocks” on page 14-9. The example below illustrates how the storage block can prevent a deadlock.

Example: Deadlock Resulting from Loop in Entity Path

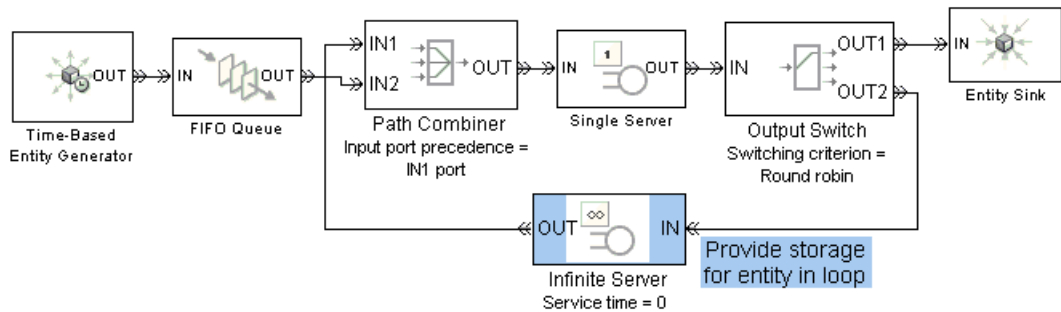
The model below contains a loop in the entity path from the Output Switch block to the Path Combiner block. The problem occurs when the switch selects the entity output port **OUT2**. The entity attempting to depart from the server looks for a subsequent storage block where it can reside, and it cannot reside

in a routing block. Until the entity confirms that it can advance to a storage 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



A better model would include 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. Notice also that 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.



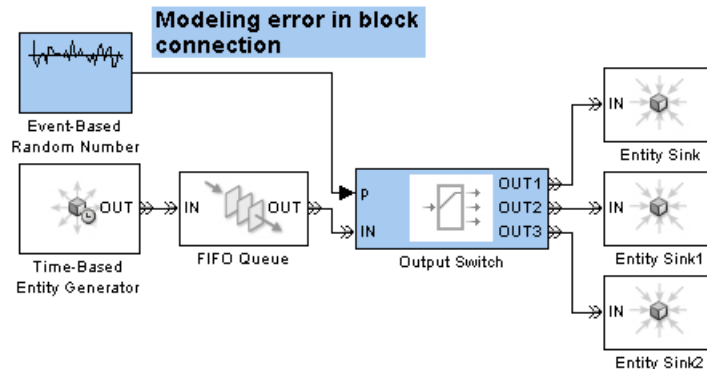
Unexpected Timing of Random Signal

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

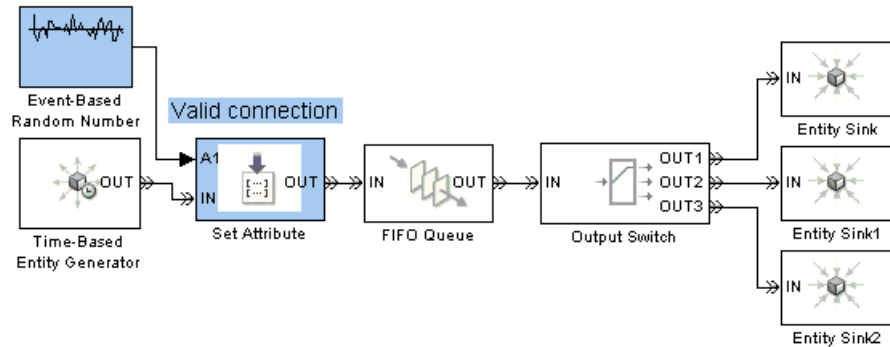
of times at which the block generates a new random number depends on the port to which the block is connected and on events occurring in the simulation. To learn how to use this block, see “Generating Random Signals” on page 3-4.

Example: Invalid Connection of Event-Based Random Number Generator

The model below 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; that is, 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 your goal is to generate a new random number corresponding to each entity that arrives at the switch, then 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 could result from equal initial seeds in different dialog boxes. If you copy and paste blocks that have an **Initial seed** parameter, the parameter values do not change unless you manually change them. Such blocks include

- Time-Based Entity Generator
- Event-Based Random Number
- Uniform Random Number
- Random Number
- Blocks in the Routing library

To make **Initial seed** parameters unique among blocks in the currently selected system, enter the following code in the MATLAB Command Window.

```
% Generate a vector of large odd numbers.
newseed = (50001 : 2 : 59999);

% Randomly permute the numbers to avoid always using the same set of seeds.
perm = randperm(length(newseed));

paramname = {'initialseed', 'seed'}; % Parameter names to consider
np = length(paramname);
idx = 1;
```

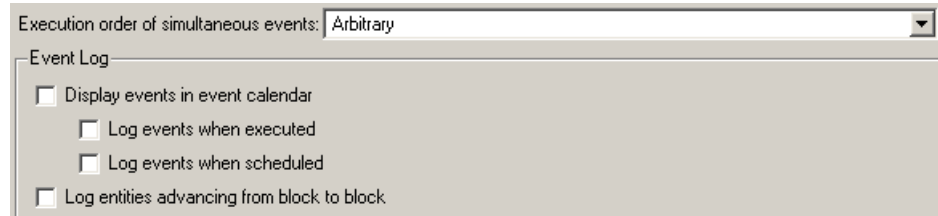


```
for jj=1:np
    % Find blocks that have the parameter with a numerical value
    % (not a variable or other expression).
    blocks = find_system(bdroot,'RegExp','on','LookUnderMasks','all',...
        paramname{jj},'\d+');

    % Replace initial seed parameter values with numbers from
    % the vector.
    for kk = 1:length(blocks)
        newseedparamvalue = num2str(newseed(perm(idx)));
        idx = idx + 1;
        set_param(blocks{kk},paramname{jj},newseedparamvalue);
        disp(['Setting parameter to ' newseedparamvalue ' in ' ...
            strep(blocks{kk},char(10),' ') '.']);
    end
end
end
```

Configuration Parameters for SimEvents Models

When a model contains at least one SimEvents block, the model's Configuration Parameters dialog box has a tab with parameters specific to discrete-event simulation.



Execution order of simultaneous events

If you select **Arbitrary**, an internal algorithm determines the sequence for processing simultaneous events having equal priorities. If you select **Randomized**, all possible sequences have equal probability. In either case, the processing sequence might be different from the sequence in which the events were scheduled on the event calendar. For more information, see “Events with Equal Priorities” on page 2-16.

Seed for event randomization

The initial seed of the random number generator used to determine the sequence for processing simultaneous events having equal priorities. For a given seed, the generator's output is repeatable. This field appears only if you set **Execution order of simultaneous events** to **Randomized**.

Display events in event calendar

If you select this option, the MATLAB Command Window displays a message and the list of events in the event calendar, each time an event is either scheduled or processed. For more information, see “Logging the List of Events” on page 13-5.

Log events when executed

If you select this option, the MATLAB Command Window displays a message each time an event is processed. For more information, see “Logging the Processing of Events” on page 13-3.

Log events when scheduled

If you select this option, the MATLAB Command Window displays a message each time an event is scheduled on the event calendar. For more information, see “Logging the Scheduling of Events” on page 13-4.

Log entities advancing from block to block

If you select this option, the MATLAB Command Window displays information about entities advancing from block to block. For more information, see “Viewing Entity Locations” on page 13-9.

How SimEvents Works

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 both time-based and event-based processing.

Notifications and Queries Among Blocks (p. 14-2)

When and why SimEvents blocks interact with each other, and the impact on simulation behavior

Notifying, Monitoring, and Reactive Ports (p. 14-4)

Types of signal input ports in SimEvents blocks

Interleaving of Block Operations (p. 14-8)

Sequence of block operations and the impact on simulation behavior

Zero-Duration Values and Time-Based Blocks (p. 14-17)

Caveats and techniques for working with multivalued signals

Notifications and Queries Among Blocks

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 are essential to the proper functioning of a discrete-event simulation. The interactions occur automatically without being reported to you explicitly.

This section gives examples of several types of notifications and queries. The topics are

- “Querying Whether a Subsequent Block Can Accept an Entity” on page 14-2
- “Notifying Blocks About Status Changes” on page 14-3

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.
- When 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.

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,

- When an entity departs from a Single Server block, it notifies the preceding block that the server's entity input port has changed from unavailable to available.
- When an entity departs from a queue that was full to capacity, the queue notifies the preceding block that the queue's entity input port has changed from unavailable to available.
- 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 notifies 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.

Notifying, Monitoring, and Reactive Ports

- “Notifying Ports” on page 14-4
- “Monitoring Ports” on page 14-5
- “Reactive Ports” on page 14-6

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 3-4
- “Using Data Sets to Create Event-Based Signals” on page 3-9

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.

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

List of Notifying Ports (Continued)

Signal Input Port	Block	Generate New Output Value Upon
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 rv port
Input port corresponding to Discrete Event Inport block in subsystem	Discrete Event Subsystem	Sample time hit at that input port

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

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. The topics are

- “How Interleaving of Block Operations Occurs” on page 14-8
- “Storage and Nonstorage Blocks” on page 14-9
- “Example: Sequence of Departures and Statistical Updates” on page 14-10
- “Example: Using the Event Calendar to Prevent Interleaving” on page 14-14

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 a subsequent storage block detects the entity’s arrival. 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.

Furthermore, entity advancement is not an atomic operation and the application might process other block operations between portions of the entity-advancement processing. Such interleaving of block operations can be undesirable in some circumstances, especially in models containing feedback loops.

To change the sequence of block operations, you might want to use one or more of these techniques:

- Insert storage blocks in key locations along entity paths in your model to change the sequence of block operations, as illustrated in “Example: Sequence of Departures and Statistical Updates” on page 14-10.
- Use the event calendar to defer operations until an entity’s departure processing is complete, as illustrated in “Example: Using the Event Calendar to Prevent Interleaving” on page 14-14.

Storage and Nonstorage Blocks

The lists of storage and nonstorage blocks in SimEvents are as follows.

Storage Blocks

- 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
- Entity Sink block
- Attribute Scope, X-Y Attribute Scope, and Instantaneous Entity Counting Scope blocks when configured as a sink, that is, without an entity output port

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.

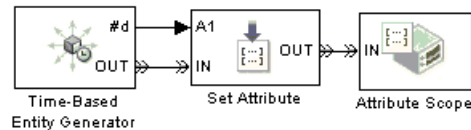
Nonstorage Blocks

- 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
- Attribute Scope, X-Y Attribute Scope, and Instantaneous Entity Counting Scope blocks when configured with an entity output port
- Entity Departure Event to Function-Call Event block
- Entity-Based Function-Call Event Generator block

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.

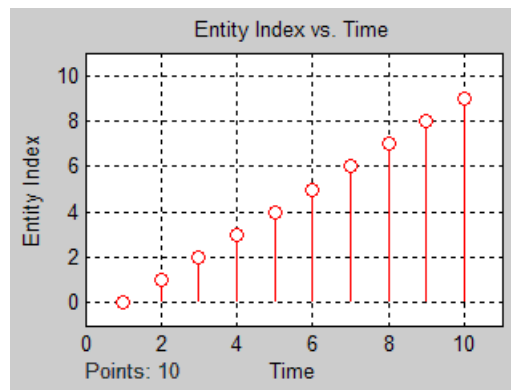


At each time $T = 1, 2, 3, \dots, 10$, Simulink processes the following operations in the order listed:

Order	Operation	Block
1	Entity generation	Time-Based Entity Generator
2	Entity departure	Time-Based Entity Generator
3	Arrival at nonstorage block	Set Attribute
4	Assignment of attribute using value at A1 signal input port	Set Attribute
5	Entity departure	Set Attribute
6	Arrival at storage block	Attribute Scope

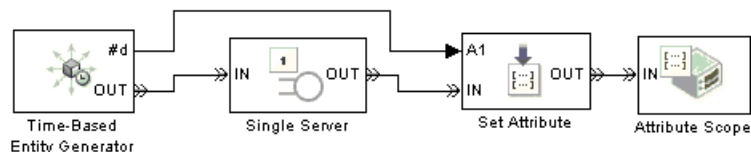
Order	Operation	Block
7	Update of plot	Attribute Scope
8	Update of signal at #d signal output port	Time-Based Entity Generator

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.

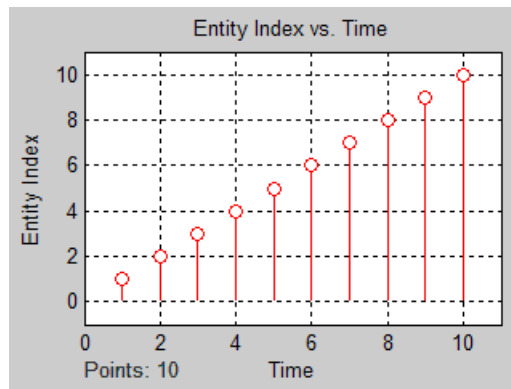


Altering the Processing Sequence

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 departure	Time-Based Entity Generator
3	Arrival at storage block	Single Server
4	Update of signal at #d signal output port	Time-Based Entity Generator
5	Service completion	Single Server
6	Entity departure	Single Server
7	Arrival at nonstorage block	Set Attribute
8	Assignment of attribute using value at A1 signal input port	Set Attribute
9	Entity departure	Set Attribute
10	Arrival at storage block	Attribute Scope
11	Update of plot	Attribute Scope

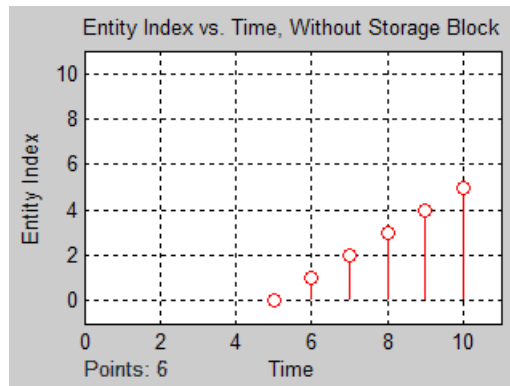
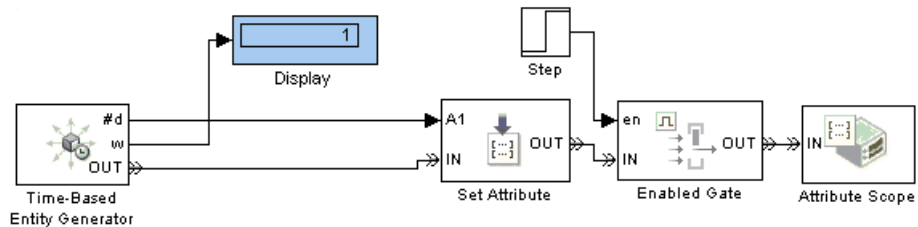


Consequences of Inserting a Storage Block

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

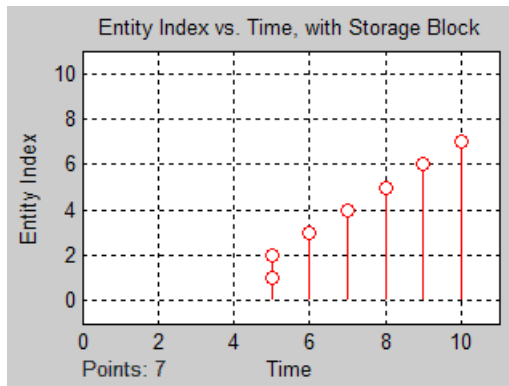
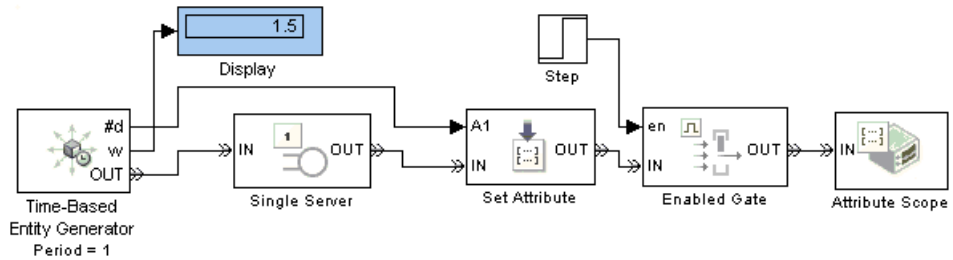
should consider the impact of either inserting or not inserting the storage block.

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

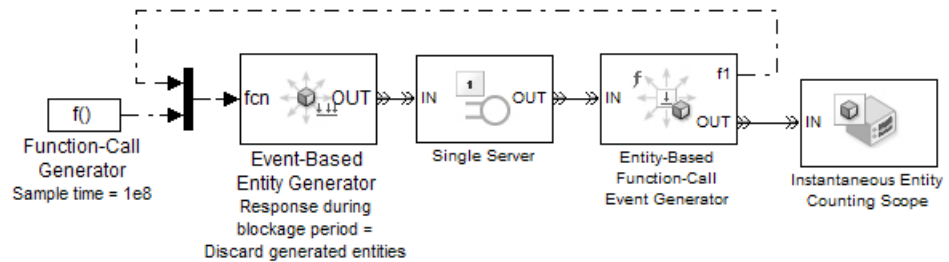
When a storage block is present, the first pending entity stays there instead of in the entity generator. The earlier departure of the first entity from the entity generator increases the value of the w signal.



Model with Gate and Storage Block

Example: Using the Event Calendar to Prevent Interleaving

This example illustrates how putting an event on the event calendar can prevent an undesirable interleaving of operations. The model below aims to follow the departure of each entity from the server with the generation of a new entity. The Entity-Based Function-Call Event Generator has the **Generate function call** parameter set to After entity departure.



The Function-Call Generator causes generation of the first entity at $T=0$. When this entity completes its service, these operations occur simultaneously in sequence:

- 1 The Single Server block starts processing the departure of the entity.
- 2 The Instantaneous Entity Counting Scope block, configured as a sink, detects the arrival of the entity.
- 3 The Entity-Based Function-Call Event Generator block generates a function call.
- 4 The Event-Based Entity Generator block, with default parameter values, responds to the function call by generating an entity immediately. The entity finds the server unavailable, and the generator responds to the blockage by discarding the entity.
- 5 The Single Server block completes its processing of the departure of the first entity. At this point, the server is available but the entity intended to enter the server has already been discarded.

Note The reason the server is unavailable is that entity advancement is not an atomic operation and the generation of the function call becomes interleaved between portions of the entity-advancement processing. The feedback loop in this model makes such interleaving undesirable.

To prevent the Event-Based Entity Generator block from generating the entity too soon, you can put the generation event on the event calendar by selecting **Resolve simultaneous signal updates according to event**

priority in the generator block. The simulation now exhibits this sequence of simultaneous events when the first entity completes its service:

- 1** The Instantaneous Entity Counting Scope block detects the arrival of the entity, although the Single Server block has not yet completed its processing of the departure of that entity.
- 2** The Entity-Based Function-Call Event Generator block generates a function call.
- 3** The Event-Based Entity Generator block reacts to the function call by scheduling an event on the event calendar for the current time.
- 4** The Single Server block completes its processing of the departure of the entity.
- 5** The event calendar causes the Event-Based Entity Generator block to generate an entity. The entity finds the server available.
- 6** The entity advances to the server.

Zero-Duration Values and Time-Based Blocks

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.

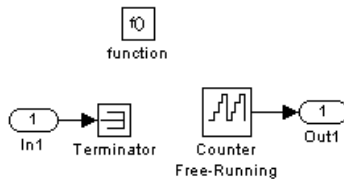
Time-Based Block	Block or Technique for Working with Multivalued Signals
Scope	Signal Scope in the SimEvents Sinks library
To Workspace	Discrete Event Signal to Workspace in the SimEvents Sinks library. Alternatively, put the To Workspace block in a discrete event subsystem.
Triggered Subsystem	Discrete 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 10-16.

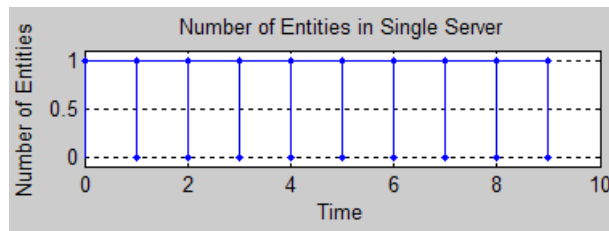
Example: Using a #n Signal as a Trigger

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.)

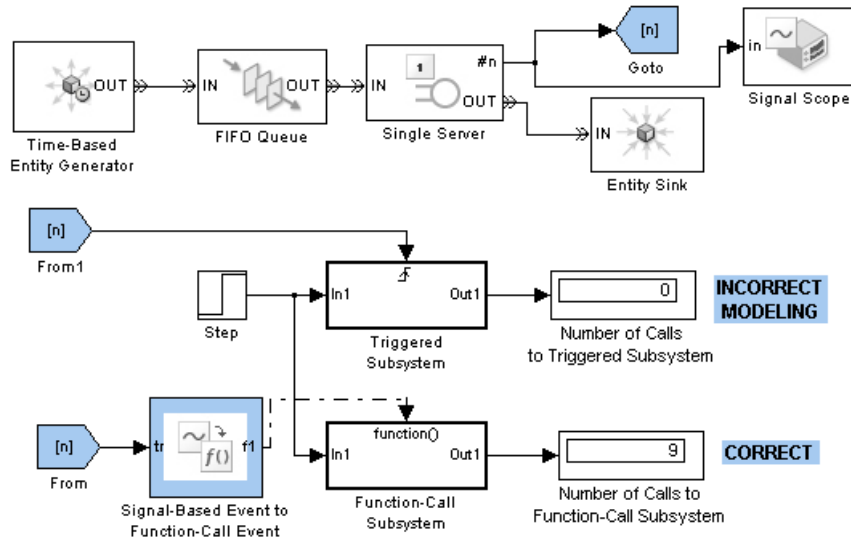


The discrete-event portion of the simulation involves a D/D/1 queuing system in which the server is never idle for a nonzero period of time. As a result, the #n signal exhibits many zero-duration values, shown in the plot below.



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.



Examples

Use this list to find examples in the documentation.

Attributes of Entities

“Example: Setting Attributes” on page 1-14

“When to Use Attributes” on page 1-16

Counting Entities

“Example: Counting Simultaneous Departures from a Server” on page 1-21

“Example: Resetting a Counter After a Transient Period” on page 1-22

Working with Events

“Example: Comparing Types of Signal-Based Events” on page 2-4

“Example: Race Conditions at a Switch” on page 2-25

“Events On and Off the Event Calendar” on page 2-31

“Example: Observing Service Completions” on page 2-38

“Example: Detecting Collisions by Comparing Events” on page 2-40

“Example: Opening a Gate Upon Random Events” on page 2-45

“Example: Counting Events from Multiple Sources” on page 2-48

Queuing Systems

“Example: Event Calendar for a Queue-Server Model” on page 2-17

“Example: Waiting Time in LIFO Queue” on page 4-2

“Example: Serving Preferred Customers First” on page 4-7

“Example: Preemption by High-Priority Entities” on page 4-11

“Example: M/M/5 Queuing System” on page 4-13

“Example: Using Servers in Shifts” on page 6-11

Working with Signals

“Example: Creating a Random Signal for Switching” on page 3-5

“Example: Resampling a Signal Based on Events” on page 3-28

“Example: Sending Queue Length to the Workspace” on page 3-31

Server States

“Example: Failure and Repair of a Server” on page 4-17

“Example: Adding a Warmup Phase” on page 4-19

Routing Entities

“Example: Cascaded Switches with Skewed Distribution” on page 5-6

“Example: Compound Switching Logic” on page 5-7

“Example: Choosing the Shortest Queue” on page 6-3

Batching

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

Gates

“Example: Controlling Joint Availability of Two Servers” on page 7-4

“Example: Synchronizing Service Start Times with the Clock” on page 7-6

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

“Example: First Entity as a Special Case” on page 7-10

Timeouts

- “Basic Example Using Timeouts” on page 8-3
- “Defining Entity Paths on Which Timeouts Apply” on page 8-7
- “Example: Dropped and Timed-Out Packets” on page 8-11
- “Example: Rerouting Timed-Out Entities to Expedite Handling” on page 8-12
- “Example: Limiting the Time Until Service Completion” on page 8-14

Discrete Event Subsystems

- “Example: Comparing the Lengths of Two Queues” on page 9-17
- “Example: Normalizing a Statistic to Use for Routing” on page 9-18
- “Example: Using Event-Based Timing for a Statistical Computation” on page 9-20
- “Example: Ending the Simulation Upon an Event” on page 9-21
- “Example: Sending Unrepeated Data to the MATLAB Workspace” on page 9-22
- “Example: Focusing on Events, Not Values” on page 9-23
- “Example: Detecting Changes from Empty to Nonempty” on page 9-24
- “Example: Logging Data About the First Entity on a Path” on page 9-25
- “Example: Using Entity-Based Timing for Choosing a Port” on page 9-30
- “Example: Performing a Computation on Selected Entity Paths” on page 9-32

Troubleshooting

- “Example: Plotting Entity Departures to Verify Timing” on page 10-10
- “Example: Plotting Event Counts to Check for Simultaneity” on page 10-14
- “Example: Event Logging” on page 13-6
- “Example: Entity Logging” on page 13-10
- “Example: Time-Based Addition of Event-Based Signals” on page 13-16
- “Example: Intergeneration Time of Zero at Simulation Start” on page 13-21
- “Example: Absence of Sample Time Hit at Simulation Start” on page 13-21
- “Example: Faulty Logic in Feedback Loop” on page 13-22

- “Example: Deadlock Resulting from Loop in Entity Path” on page 13-23
- “Example: Invalid Connection of Event-Based Random Number Generator” on page 13-25
- “Example: Sequence of Departures and Statistical Updates” on page 14-10
- “Example: Using the Event Calendar to Prevent Interleaving” on page 14-14
- “Example: Using a #n Signal as a Trigger” on page 14-18

Statistics

- “Example: Fraction of Dropped Messages” on page 11-8
- “Example: Computing a Time Average of a Signal” on page 11-10
- “Example: Resetting an Average Periodically” on page 11-12
- “Example: Running a Simulation Repeatedly to Gather Results” on page 11-28
- “Example: Running a Simulation and Varying a Parameter” on page 11-30

Timers

- “Basic Example Using Timer Blocks” on page 11-19
- “Timing Multiple Entity Paths with One Timer” on page 11-21
- “Restarting a Timer from Zero” on page 11-23
- “Timing Multiple Processes Independently” on page 11-24

A

- arbitrary event sequence 2-16
 - troubleshooting 13-14
- ARQ
 - Stateflow charts 12-5
- attributes of entities
 - combining 1-24
 - plots 10-2
 - reading values 1-19
 - setting values 1-13
 - usage 1-16
- autoscaling 10-8
- averaging signals
 - over samples 11-12
 - over time 11-10
- axis limits 10-5

B

- block-to-block interactions 14-2

C

- caching 10-6
- cascading switch blocks
 - random 5-6
- combining entities 1-24
- combining events 2-47
- conditional events 2-52
- Configuration Parameters dialog box 13-28
- copying entities 1-30
- counting entities
 - cumulative 1-20
 - instantaneous 1-20
 - reset 1-22
 - storing in attribute 1-23
- counting events 10-2
 - simultaneity 10-14

D

- data history 10-6
- data types 3-3
- delays
 - signal updates 3-25
- Discrete Event Subsystem block
 - building subsystems 9-11
- discrete event subsystems 9-7
 - blocks inside 9-10
 - building
 - entity departures 9-29
 - function calls 9-34
 - signal-based events 9-11
 - combinations of events 9-33
 - entity departures 9-27
 - events in terminated signals 9-23
 - function calls 9-33
 - multiple inputs 9-15
 - need for 9-2
 - sequence of events 9-8
 - signal-based events 9-14
- discrete state plots 10-4
- discrete-event plots 10-4
 - compared to time-based plots 10-16
 - customizing 10-8
 - saving FIG-file 10-8
 - troubleshooting using 10-9
- dropped messages 11-8
 - timeouts 8-11

E

- Embedded MATLAB Function blocks 6-3
- enabled gates 7-4
- entities
 - combining 1-24
 - counting 1-20
 - event-based generation 1-2
 - logging 13-9
 - replicating 1-30

- synchronizing 1-24
- timeouts 8-1
- entity collisions 2-40
- entity data
 - combining 1-24
 - plots 10-2
 - reading values 1-19
 - setting values 1-13
 - usage 1-16
- entity generation
 - changes in signal value 1-4
 - event-based 1-2
 - function calls 1-7
 - trigger edges 1-5
 - updates in signal value 1-2
 - vector of times 1-11
- entity logging 13-9
- entity paths
 - timeouts 8-7
- entity-departure subsystems 9-27
- equal event priorities 2-16
 - troubleshooting 13-14
- event calendar 2-9
 - events on 2-9
 - events on/off 2-31
 - example 2-17
 - logging 13-2
- event logging 13-2
- event-based sequences 3-9
- event-based signals
 - data sets 3-9
 - deferring reactions 3-13
 - description 3-2
 - feedback loops 13-21
 - initial conditions 3-27
 - integrating 9-3
 - latency 13-17
 - manipulating 3-27
 - MATLAB workspace 3-31
 - random 3-4
 - resampling 3-28
 - resolving updates 3-17
 - troubleshooting 13-15
 - unrepeated values to workspace 9-22
 - update sequence 3-18
- events
 - conditionalizing 2-52
 - generating 2-43
 - manipulating 2-46
 - observing 2-36
 - on/off event calendar 2-31
 - priorities 2-15
 - reacting to signal updates 3-13
 - resolving signal updates 3-17
 - sequence 2-11
 - modeling approaches 2-14
 - supported types 2-2
 - timeout 8-1
 - translating 2-50
 - troubleshooting 13-14
 - union 2-47

F

- failure modeling
 - conditional events 2-53
 - gates 4-15
 - Stateflow 4-16
- feedback entity paths
 - troubleshooting 13-23
- feedback loops
 - troubleshooting 13-21
- first-order-hold plots 10-4
- function calls 2-7
 - generating 2-43
- function-call subsystems
 - discrete event 9-33

G

- gates 7-1
 - combinations 7-9
 - enabled 7-4
 - entity departures 7-7
 - release 7-6
 - role in modeling 7-2
 - types 7-3

I

- independent replications 11-26
- initial conditions 3-27
 - feedback loops 13-21
- initial port selection
 - switching based on signal 5-2
- initial seeds 11-26
- input signals
 - deferring reactions to updates 3-13
 - resolving updates 3-17
- instantaneous gate openings 7-6
- integration
 - event-based signals 9-3
- intergeneration times
 - event generation 2-45
- interleaved operations 14-8

L

- latency
 - interleaved operations 14-8
 - signal updates 3-25
 - switching based on signal 5-2
 - troubleshooting 13-17
- LIFO queues 4-2
- livelock detection 2-12
- logging entities 13-9
- logging events 13-2
- logic
 - block diagrams 6-10

- MATLAB code 6-3

- usage in discrete-event simulations 6-2

- loops in entity paths 13-23

M

- M/M/5 queuing systems 4-13
- monitoring ports 14-5

N

- n-servers 4-13
- nonstorage blocks 14-9
- notifying ports 14-4

O

- Output Switch block
 - signal-based routing 5-2

P

- plots 10-1
 - customizing 10-8
 - troubleshooting using 10-9
 - zero-duration values 3-22
- ports
 - monitoring 14-5
 - notifying 14-4
 - reactive 14-6
- preemption in servers 4-10
- priorities, entity
 - priority queues with preemptive servers 4-11
 - queue sequence 4-4
 - server preemption 4-10
- priorities, event 2-15
 - reacting to signal updates 3-13
 - resolving signal updates 3-17
 - troubleshooting 13-14

Q

queues

- choosing shortest using logic blocks 6-16
- choosing shortest using MATLAB code 6-3
- LIFO vs. FIFO 4-2
- preemptive servers 4-11
- priority 4-4

queuing systems

- M/M/5 4-13

R

race conditions 2-25

random

- signals 3-4

random event sequence 2-16

- troubleshooting 13-14

random numbers

- event-based 3-4
- switch selection 3-5
- time-based 3-6

reactive ports 14-6

recursion 2-12

release gates 7-6

reneging in queuing 8-3

repeating simulations 11-28

replication of entities 1-30

replications

- independent 11-26

resampling signals 3-28

resetting averages 11-12

residual service time 4-10

resolving simultaneous updates of signals 3-17

running simulations

- repeatedly 11-28
- varying parameters 11-30

S

sample means 11-12

sample time 3-2

scatter plots 10-4

scope blocks 10-1

- zero-duration values 3-22

seed of random number generator

- independent replications 11-26

server states 4-15

servers

- failure states 4-15
- multiple 4-13
- preemption 4-10

signal-based events

- comparison 2-4
- definition 2-3

signals

- deferring reactions to updates 3-13
- event-based 3-2
- event-based data 3-9
- random 3-4
- resolving updates 3-17

simulation parameters

- varying in repeated runs 11-30

simultaneous events

- discrete event subsystems 9-8
- event priorities 2-15
- input signal updates 3-17
- interleaved operations 14-8
- modeling approaches 2-14
- on/off event calendar 2-31
- output signal updates 3-21
- sequential processing 2-11
- troubleshooting 13-14
- unexpected 13-13

splitting entities 1-24

stack 4-2

stairstep plots 10-4

Stateflow 12-3

Stateflow and SimEvents 4-16

statistics 11-2

- accessing from blocks 11-4

- custom 11-7
- discrete event subsystems 9-3
- interleaved updates 14-8
- latency 3-25
- Statistics tab 11-4
- stem plots 10-4
- stop time 11-33
 - event-based timing 9-2
- storage blocks 14-9
 - changing processing sequence 14-11
 - looped entity paths 13-23
- switching entity paths
 - based on signal 5-2
 - initial port selection 5-2
 - random with cascaded blocks 5-6
 - repeating sequence 9-4
 - storing entity 5-2
- synchronizing entities 7-6

T

- time averages 11-10
- time-based blocks
 - zero-duration values 14-17
- timed-out entities 8-10
 - routing 8-12
- timeout intervals 8-4
- timeout paths 8-7
- timeout tags 8-4
- timeouts of entities 8-1

- role in modeling 8-2
- timer tags 11-21
- timers 11-19
 - combining 1-26
 - independent 11-24
 - multiple entity paths 11-21
 - restarting 11-23
- triggers
 - zero-duration values 14-18

V

- visualization 10-1
 - zero-duration values 3-22

W

- workspace 3-31
 - unrepeated signal values 9-22

Z

- zero-duration values
 - definition 3-21
 - MATLAB workspace 3-24
 - time-based blocks 14-17
 - visualization 3-22
- zero-order hold
 - plots 10-4